Modeling and Testing of Component-Based Systems

Bilal Kanso

▶ **To cite this version:**

Bilal Kanso. Modeling and Testing of Component-Based Systems. Formal Languages and Automata Theory [cs.FL]. Ecole Centrale Paris, 2011. English. NNT : . tel-00650258

HAL Id: tel-00650258
https://theses.hal.science/tel-00650258

Submitted on 9 Dec 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

MODELING AND TESTING OF COMPONENT-BASED SYSTEMS

Bilal KANSO

November 21, 2011

M. Marc AIGUIER	Ecole Centrale Paris (ECP)	Thesis director
M. Frédéric BOULANGER	Engineering School SUPELEC	Co-advisor
M. Farhad ARBAB	Centre for Mathematics and Computer Science (CWI) Amsterdam	Reporter
Mme. Virginie WIELS	Onera Toulouse	Reporter
M. Roland GROZ	Université de Grenoble et Ensimag	Examiner
M. Daniel KROB	Ecole polytechnique	Examiner

Order number: 2011*ECAP*0050

Acknowledgments

I thank my parents for encouraging and supporting me during my whole study.

Abstract

In spite of several decades of research, assuring the quality of software systems still represents a major and serious problem nowadays for the industry with respect to both results and costs. This thesis comes within the scope of a proposal centered on a generic unified framework for both complex software systems modeling and testing.

The contribution of this paper is then twofold: first, it defines a unified framework for modeling generic components, as well as a formalization of integration rules to combine their behaviour. This is based on a coalgebraic definition of components, which is a categorical representation allowing the unification of a large family of formalisms for specifying state-based systems. Second, it studies compositional conformance testing i.e. checking whether an implementation made from correct interacting components combined with integration operators conforms to its specification.

keywords: Component-based system, Integration operators, Trace semantics, Transfer function, Compositional testing, Conformance testing, Coalgebra, Monad, Testing in context.

La thèse s'inscrit dans le domaine de la modélisation et de la validation des systèmes modernes complexes. Les systèmes actuels sont en fait d'une complexité sans cesse croissante et formés de plus en plus de composants de natures différentes. Ceci rend leur processus de conception et de validation coûteux et difficile. Il semble être la simple façon permettant de faire face à cette hétérogénéité et à cette complexité est l'approche orientée composant. Suivant cette approche, le système est une entité formée par un ensemble des composants interconnectés. Les composants définissent une interface qui permet d'abstraire leur modèle interne (boîte noire), ce qui favorise la modularité et la réutilisation des composants. L'interaction entre ces composants se fait conformément à un ensemble des règles pré-établies, permettant ainsi d'avoir une vision globale de comportement du système.

La conception ainsi que la validation des systèmes modernes reste alors problématique à cause de la nécessité de prendre en compte l'hétérogénéité des différents composants. Dans ce cadre, dans un premier temps, nous définirons un cadre formel générique dans lequel une large famille de formalismes de description de systèmes à base d'états peut être naturellement capturée. Ainsi, nous allons définir un ensemble de règles de composition permettant de mettre en correspondance les différents composants et ainsi de constituer un modèle global du système à concevoir. Dans un second temps, nous proposerons une approche de test d'intégration qui permet de valider le comportement d'un système complexe sous l'hypothèse que chaque composant est testé et validé. Cette approche vise à générer automatiquement des cas de test en s'appuyant sur un modèle global décrit dans notre framework du système sous test.

Mots-clés : Systèmes à base de composants, Opérateurs d'intégration, Modèle de traces, Fonctions de transfert, Test de conformité, Test compositionnel, Coalgèbres, Monades.

Contents

1 Contexte général . . . 1
1.1 Modélisation d'un système . . . 1
1.2 Validation et vérification . . . 4
2 Contributions de la thèse . . . 7

I Introduction **11**
1 Context . . . 11
1.1 System modeling . . . 11
1.2 Validation and verification . . . 14
2 Thesis overview . . . 18
2.1 Thesis contributions . . . 18
2.2 Plan of the thesis . . . 20

I Theoretical preliminaries **23**

II Category theory **27**
1 Category . . . 28
1.1 Category definition . . . 28
1.2 Constructions of categories . . . 29
1.3 Properties of arrows . . . 30
2 Universal properties . . . 31
2.1 Commutative diagrams . . . 31
2.2 Initial and terminal objects . . . 32
2.3 Product . . . 32
2.4 Coproduct . . . 33
2.5 Exponents . . . 34
3 Functors and natural Transformations . . . 36
3.1 Functors . . . 36
3.1.1 Powersets . . . 37
3.1.2 Free monoid . . . 37
3.1.3 Polynomial functors and Kripke polynomial functors . . . 37
3.1.4 The category of category . . . 38
3.2 Natural transformations . . . 38
3.3 Heterogeneous Compositions . . . 40
3.3.1 Functor categories . . . 40
3.3.2 Heterogeneous compositions . . . 40
4 Monads in category theory . . . 41
4.1 Definition . . . 42
4.2 A working example . . . 42

4.3 More examples . . . 44
4.3.1 Partial . . . 44
4.3.2 Ordered nondeterminism . . . 45
4.3.3 Exception . . . 45
4.4 Category of Kleisli . . . 46

III Coalgebras **49**
1 Coalgebra definition . . . 50
1.1 Streams . . . 50
1.2 Mealy Machines . . . 51
1.3 Labeled Transition Systems (LTS) . . . 51
1.4 Input-Output Labeled Transition Systems (IOLTS) . . . 52
2 Morphisms . . . 52
3 Bisimulation . . . 54
3.1 Stream . . . 55
3.2 Mealy machines . . . 55
3.3 Labeled transition systems . . . 55
4 Final coalgebras . . . 57
4.1 Streams . . . 58
4.2 Mealy machines . . . 60
4.3 Labeled transition systems . . . 61
4.4 More examples . . . 63
5 Co-induction . . . 64
5.1 Proof by bisimulation . . . 66

II Systems modeling framework **69**

IV Generic components **73**
1 Components as coalgebras . . . 74
1.1 Motivation . . . 74
1.2 Components . . . 74
1.3 Genericity of component definition . . . 77
2 Component traces . . . 80
2.1 Transfer function . . . 80
2.2 Component Traces . . . 81
3 Results . . . 82
3.1 Final model . . . 83
3.2 Minimal component . . . 85
4 Conclusion . . . 88

V Integration of components **89**
1 Basic integration . . . 90
1.1 Cartesian product . . . 90
1.2 Feedback . . . 90
2 Complex operators . . . 99
2.1 Sequential composition . . . 100
2.2 Double sequential composition . . . 102
2.3 Synchronous product . . . 104
2.4 Concurrent composition . . . 105

2.5 Synchronous parallel composition . . . 106
3 Systems and compositionality . . . 108
3.1 Systems . . . 108
3.2 Examples . . . 109
3.3 Compositionality . . . 118
4 Related works . . . 123
5 Conclusion . . . 125

III Validation of component-based systems by testing 127

VI Conformance testing theory: a general overview 131
1 Formal Method in Conformance Testing . . . 132
1.1 General principle . . . 132
1.2 The meaning of conformance . . . 133
1.2.1 Specification model . . . 133
1.2.2 Implementation model . . . 133
1.2.3 Conformance relation . . . 133
1.3 Formal framework for conformance testing . . . 134
1.3.1 Test execution . . . 134
1.3.2 Test case properties . . . 135

VII Testing of components 137
1 Conformance relation . . . 138
1.1 Specification model . . . 138
1.2 Implementation model . . . 138
1.3 Conformance . . . 139
1.3.1 An overview . . . 139
1.3.2 Definition . . . 141
2 Finite computation tree . . . 144
2.1 Formal definition . . . 144
2.2 Unfolding algorithm . . . 146
3 Test Purpose . . . 149
4 Test generation guided by test purposes . . . 150
4.1 Preliminaries . . . 153
4.2 Inferences rules . . . 154
4.3 Example . . . 156
4.4 Properties . . . 158
5 Instantiating of the approach . . . 161

VIII Integration Testing 163
1 Compositional testing . . . 164
1.1 Compositional testing with cioco . . . 164
1.2 Compositionality for cartesian product . . . 168
1.3 Compositionality for feedback operators . . . 168
1.4 Compositionality for complex operator . . . 172
2 Test purposes for sub-systems . . . 173
2.1 Sub-systems and projection . . . 174
2.2 System-based test purposes . . . 175
3 Related works . . . 177

IX Conclusion **181**
1 Summary 181
2 Future research 181

Bibliography **187**

Introduction Générale

Ce chapitre décrit le contexte de cette thèse et la motivation de notre recherche. Il propose une définition d'un système complexe et se demande comment celui-ci peut être modélisé et testé. Il présente aussi la « feuille de route » pour les chapitres à suivre et fournit les concepts et notions nécessaires pour à la compréhension.

1 Contexte général

En dépit de plusieurs décennies de recherche, assurer la qualité des systèmes informatiques représente toujours un problème actuel majeur et épineux pour l'industrie en ce qui concerne à la fois les résultats et les coûts [1]. Cette thèse a pour but de proposer une théorie générique et unifiée pour la modélisation et la validation des systèmes complexes.

Par la suite, nous présenterons les concepts et les notions nécessaires pour atteindre notre objectif.

1.1 Modélisation d'un système

Les travaux que nous proposons dans cette thèse consistent d'abord à définir un formalisme générique permettant de modéliser la notion d'un composant informatique comme une coalgèbre concrète d'un foncteur donné dans la catégorie des ensembles **Set**. Avant d'aller à l'explication de ce formalisme, nous allons d'abord succinctement expliquer :

- Ce que nous endentons par la notion d'un composant et d'un système complexe,
- L'importance d'avoir une description générique d'un composant,
- Pourquoi utiliser la définition du composant proposé par *Barbosa* dans le cadre de la théorie des coalgèbres et celle des monades.

Composants L'approche orientée composant [2, 3, 4] est de plus en plus sollicitée grâce à son potentiel. Elle offre de nombreux avantages tels que la modularité, la ré-utilisabilité, et divers autres avantages. Dans cette approche, les composants sont séparément conçus, développés et testés une seule et unique fois, afin d'être utilisés dans divers domaines.

Les fonctionnalités d'un composant sont alors définies à un niveau d'abstraction plus haut que celui de son implantation. Ainsi, la meilleur façon d'adresser formellement le composant est de le considérer comme une représentation abstraite, en omettant les détails de son implantation, tout en surlignant les principales propriétés liées à ses fonctionnalités. Cette représentation est seulement construite en décrivant comment les sorties sont produites en corrélation à la fois avec les entrées et l'état du composant. On parle classiquement d'une représentation *boîte noire*. La structure interne d'un système (i.e comment il a été mis en œuvre) est ainsi ignorée. Seule la

vue fonctionnelle (ou comportementale) du système (i.e ce que fait un système) est considérée. Toutes les communications entre le système et l'environnement sont décrites via l'interface du système. Un observateur extérieur peut seulement observer le système via cette interface. Une telle vue d'un observateur externe représente le comportement complet du système en termes d'entrées sorties.

Du point de vue de l'environnement, une boîte noire peut alors être considérée comme une fonction qui associe, à tout instant t (qui peut être discret ou continu), une sortie du système $o \in \mathsf{Out}$ après avoir reçu une entrée $i \in \mathsf{In}$. Cette fonction dépend de l'état courant $s \in S$ du système, et prend la forme suivante :

$$y = \mathcal{F}(x, s, t)$$

où x, y et s sont respectivement l'entrée, la sortie et l'état du composant considéré et t est l'instant de temps.

D'un point de vue environnemental, un système peut donc être représenté comme dans la figure suivante.

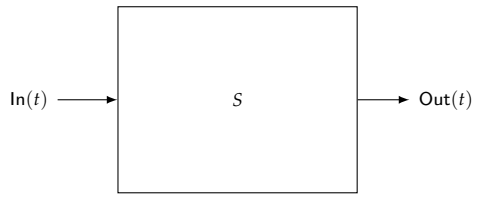


FIGURE 1 – Représentation du système par une boîte noire

Systèmes complexes Jusqu'ici, nous avons vu que l'approche orientée composant pourrait faire face à l'hétérogénéité croissante des composants de différentes natures en cachant plus ou moins de détails liés aux architectures internes des composants selon l'étape du processus de conception considérée. Cela signifie que le comportement global d'un système complexe, serait moins difficile à étudier quand les sous-systèmes sous-adjacents sont considérés comme des représentations boîte noire. De cette façon, une approche puissante pour le développement des systèmes informatiques complexes pourrait se reposer sur l'idée qu'un système complexe est construit en ayant assemblé des composants d'une manière hiérarchique et récursive. Ces sous-systèmes sont ainsi intégrés par le biais de connecteurs architecturaux, qui sont des outils puissants pour décrire les systèmes en termes de composants et de leurs interactions [4].

L'idée de base est que, à un haut niveau d'abstraction, les systèmes complexes peuvent être récursivement et hiérarchiquement constitués d'un ensemble de plusieurs sous-systèmes interconnectés. Chaque sous-système peut être lui-même un système complexe ou un système simple et élémentaire qui peut être traité entièrement. Une telle description hiérarchique formelle d'un système complexe ressemblerait généralement à la construction illustrée dans la Figure I.2.

La composition est utilisée pour assembler les différents sous-systèmes entre eux pour alors former de plus grands systèmes. Une telle composition peut être considérée comme une opération qui prend les composants ainsi que la nature de leurs interactions pour construire un nouveau composant beaucoup plus complexe. Cette représentation récursive d'un système complexe comme une composition de composants élémentaires en utilisant les règles d'intégration, nous permet alors de séparer la réalisation de différents composants. Cela a pour conséquence de rendre le processus de conception plus modulaire. La composition elle-même devient un composant qui peut être encore composé comme si il était lui-même un composant atomique.

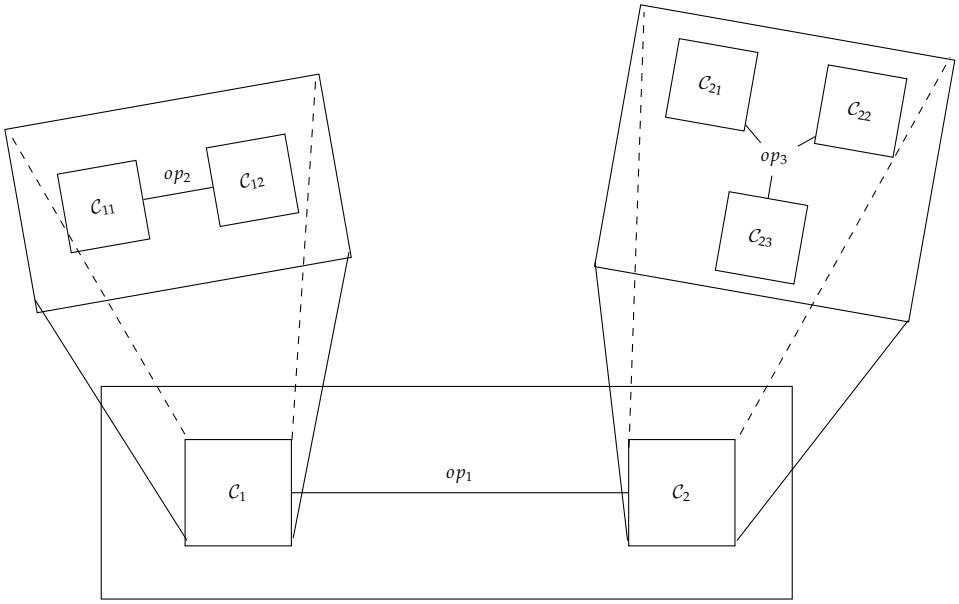


FIGURE 2 – vue compositionnelle d'un système complexe

Vue unifiée des systèmes Les sous-systèmes à partir desquels les systèmes complexes sont conçus, peuvent en effet être modélisés, en utilisant des formalismes de spécification qui diffèrent principalement selon les disciplines scientifiques et les modèles de calcul utilisés pour spécifier l'interaction entre les différents éléments du système. A l'heure actuelle, il n'existe toujours pas de formalisme unifié qui pourrait supporter la conception et la validation des systèmes modernes dans leur généralité. Néanmoins, on peut observer que la plupart des systèmes modernes peuvent être considérés, d'un point de vue théorique et à un haut niveau d'abstraction, comme des systèmes à base d'états dont le comportement peut être seulement observé au travers de leurs interfaces.

Cependant, il s'avère de plus en plus difficile, voire impossible, de construire par de formalismes de description classiques tels que les logiques du premier ordre ou équationnelle, le comportement de ces systèmes dont les structures sont intrinsèquement dynamiques. En effet, les caractéristiques des systèmes actuels tendent à être vues non plus du point de vue de la façon dont ils ont été construits, mais de celui des résultats qu'ils produisent. La sémantique de ces systèmes est alors essentiellement observationnelle. Tout ce qui peut être capturé dans leur évolution est leur interaction avec l'environnement, c'est-à-dire les échanges possibles entre des entrées et des sorties avec le monde extérieur. De ce point de vue, l'approche coalgèbrique peut être perçue comme l'approche duale de l'approche algébrique. Au lieu de construire les objets, on les observe. Dans ce contexte, la théorie des catégories et celle des coalgèbres fournissent tous les concepts nécessaires pour gérer abstraitement le comportement observable de tels systèmes.

Approches coalgèbriques pour la modélisation des systèmes Les coalgèbres se montrent ainsi de plus en plus adaptées pour définir la sémantique des systèmes réactifs. Elles sont considérées aujourd'hui comme une abstraction de formalismes à base d'états fournissant un cadre formel unifié et générique dans lequel une large variété des systèmes à base d'états pourrait être décrite tels que les systèmes de transition, les automates et les classes dans les langages orientés objet [5, 6, 7, 8]. Une des contributions majeures dans la modélisation coalgèbrique d'un composant est le travail proposé par *Barbosa* [9, 10, 11, 12, 13]. Les coalgèbres ont été utilisées

comme un modèle sémantique pour les composants informatiques pour un endofoncteur dans la catégorie des ensembles. Un composant est ainsi décrit comme une machine de *Mealy* étendue par paramétrisation par une monade T, comme il est d'usage dans la programmation fonctionnelle [14]. La monade T agit comme un modèle de comportement qui peut gérer les calculs usuels tels que le déterminisme, le non-déterminisme, les situations de blocage, les exceptions et beaucoup d'autres encore [15, 16].

La notion d'un composant comme il a été introduit par *Barbosa* est la pierre angulaire de notre modélisation.

1.2 Validation et vérification

Correction des systèmes et son importance Avec la complexité croissante des systèmes informatiques modernes, les techniques de vérification et de validation gagnent en importance. Assurer le bon fonctionnement des systèmes modernes est devenu l'un des défis majeurs de nos jours. Des conséquences dramatiques en termes de vies humaines, de pertes économiques, de problèmes écologiques, etc. sont causées lorsque une panne (ou défaillance) dans le système survient. Les pannes des systèmes sont en effet omniprésentes à tel point qu'elles sont si familières pour nous que nous avons pris l'habitude de les oublier. Certaines d'entre elles ont peu d'impact dans notre vie quotidienne, par exemple, lorsque notre téléphone mobile ne fonctionne pas correctement ou notre enregistreur vidéo réagit de façon inattendue et à tort, à des commandes. Toutefois, d'autres erreurs ont un énorme impact. Dans l'histoire, il y a une longue liste de bugs typiques qui ont causé des lourdes pertes en termes de vies humaines tels que *Therac-25*, ou en terme d'argent tels que le *crash d'Ariane 5* (qui a coûté environ 500 millions de dollars américains) et le *bug de la division du Pentium* (environ 500 millions de dollars américains), etc. [17, 18]. Ces exemples font que la correction des systèmes informatiques constitue aujourd'hui un vrai défi (en termes de temps et d'argent) autant pour la conception que pour la validation et la vérification de leur comportement.

Test et vérification La validation d'un système se fait par la mise en œuvre de méthodes utilisant une représentation formelle du comportement du système. Dans ce cadre, plusieurs techniques ont été proposées telles que la preuve, la vérification de modèles « *model checking* » ou encore le test. La preuve, telle que la logique de *Hoare* [19], consiste à exprimer formellement les propriétés attendues du système et à démontrer que ces propriétés sont vraies par déduction à partir d'un ensemble d'axiomes et de règles d'inférences. Les techniques de *model checking* [20, 21] consistent à construire, à partir d'une description formelle d'un système, l'espace des états atteignables puis, en parcourant l'ensemble de ces états, à vérifier d'une manière automatique le respect des propriétés attendues du système. Le test [22, 23] vise à vérifier par expérimentation que le système sous test répond bien aux critères déjà définis (robuste, fonctionnalités correctement implémentées, etc.).

Ces trois techniques sont complémentaires dans le sens où aucune d'entre elles ne suffit à elle-même pour établir complètement la correction d'un système. En effet, la mise en œuvre de la technique par la preuve est très difficile et nécessite généralement de fortes compétences mathématiques de la part de l'utilisateur. De plus, elle n'est pas complètement automatisée. Par contraste, bien que les techniques de *model checking* soient exhaustives et automatiques, elles se heurtent rapidement au problème de l'explosion combinatoire du nombre des états du système. Enfin, le test peut servir à détecter automatiquement les erreurs du système mais n'établit jamais leur absence [24]. Il est indispensable pour nous renseigner sur la correction d'un système, surtout dans le cas où la complexité des modèles est importante.

Dans ce travail, nous intéresserons plus particulièrement aux techniques de tests. Nous proposerons de nous intéresser à la définition d'une théorie du test. Nous justifions de porter notre attention sur le test au détriment de la preuve pour deux raisons :

1. Le premier est que, nous avons pour ambition de définir un formalisme, permettant de modéliser des systèmes informatiques complexes dont l'une des caractéristiques est leur grande taille. Or nous venons de voir précédemment que pour de tels systèmes, les méthodes de vérification étaient limitées.

2. La seconde raison de notre choix est plus pratique. En effet, les méthodes de vérification ont pour but essentiel de montrer que les systèmes vérifient un certain nombre de propriétés formelles. Ces propriétés formelles sont exprimées dans une logique. Donc, pour aborder la vérification dans notre cadre, il aurait fallu définir une logique (temporelle) au-dessus de notre formalisme et établir un certain nombre de propriétés sur cette logique. Il faudrait par exemple définir un calcul et montrer que ce dernier est correct et complet, montrer que la logique est stable pour la bisimulation, étudier la préservation des propriétés au-travers des opérateurs d'intégration, etc. Or tout ceci est souvent un travail de longue haleine, difficilement abordable en même temps que la définition du formalisme lui-même. Cependant, la définition d'une telle logique est primordiale. Elle fait donc naturellement partie des perspectives de la thèse.

Techniques du test Le test est l'activité du génie logiciel qui vise à renforcer la qualité des systèmes à l'aide d'expérimentations dans l'intention d'y trouver des erreurs. Il peut être utilisé pour révéler la présence d'erreurs, mais ne peut pas établir leur absence [22, 24, 25]. Dans la littérature, on trouve une multitude de techniques de test qui différent principalement selon les trois critères suivants:

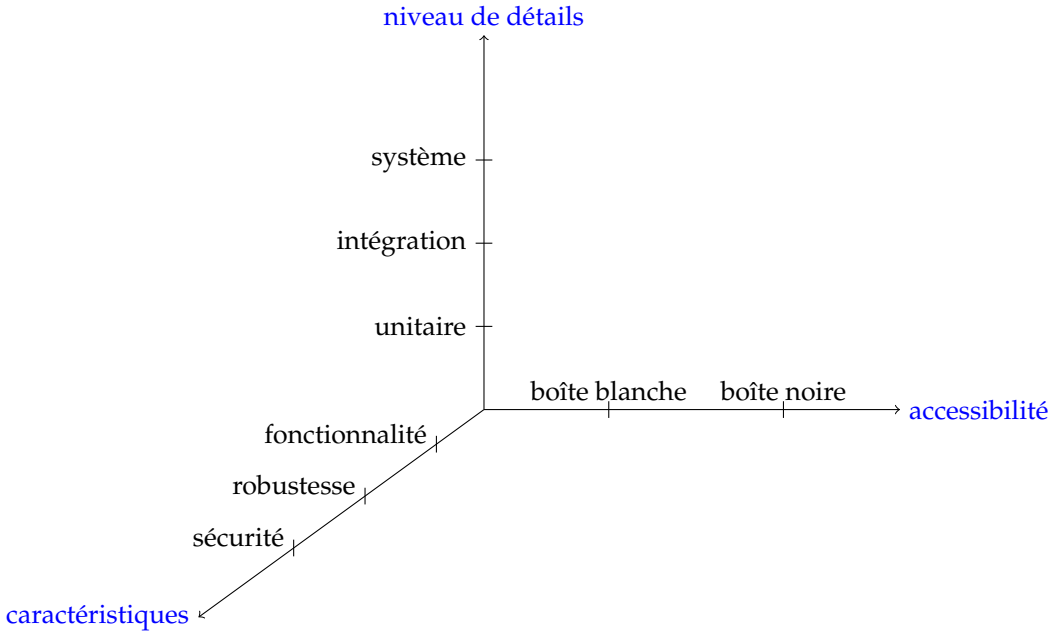


FIGURE 3 – Classification des techniques de test

1. **niveau d'accessibilité du code** : deux techniques de test sont généralement distinguées. Dans le *test structurel* (appelé aussi test *boîte blanche*) [23], la structure interne du système est connue. Le testeur a le droit d'accéder à des informations comme la structure de code, les valeurs possibles de certaines variables du programme, etc. Dans le *test fonctionnel* (appelé aussi test *boîte noire*) [26], la structure interne du système sous test est cachée. Le testeur ne dispose pas du code source du système sous test. Il ne peut accéder qu'aux descriptions des fonctionnalités du système.

2. **niveau d'abstraction** : trois méthodes de test sont communément distinguées. Pour le *test unitaire*, les systèmes examinés ne peuvent pas être subdivisés en d'autres sous-systèmes. Pour le *test d'intégration*, un système construit comme une combinaison d'un ensemble de sous-systèmes (or composants) est examiné. Pour le *test du système*, le système complet est testé.

3. **l'aspect que l'on veut tester** : il y a multitude d'aspects différents que l'on pourrait utiliser pour tester un système. Par exemple, le *test de conformité* consiste à tester le comportement de implémentation pour s'assurer que celle-ci va se comporter comme prévu (par rapport à une référence de correction disponible représentée par un cahier des charges). Le *test de robustesse* vise à tester si l'implémentation réagit bien au comportement non-spécifié, incertain ou malveillant d'environnement, etc.

Dans le cadre de cette thèse, nous allons bien entendu nous intéresser aux techniques de test unitaire et d'intégration qui sont fonctionnelles et fondées sur le test de conformité.

Test de conformité Le test de conformité est une technique de test de type boîte noire, qui a pour objectif de vérifier la conformité d'une réalisation concrète du système (*implémentation*) à une description formelle de son comportement (*spécification*) [27, 26]. Il vise à réaliser des expérimentations sur l'implantation afin de vérifier la correction de son comportement vis-à-vis de sa spécification. L'idée consiste alors à dériver automatiquement une suite de cas de tests de la spécification suivant un algorithme de dérivation de tests. L'exécution de chacun des tests générés sur l'implantation à l'aide d'un testeur (qui peut être un être humain ou un programme) mène à un verdict : Pass ou Fail. Le verdict Pass indique que l'implantation est conforme à la spécification pour ce test, tandis que le verdict Fail signifie la non-conformité.

Test d'intégration A l'heure actuelle, la complexité des systèmes, qui ne cesse de croître, mène naturellement à des difficultés, voire une impossibilité dans certains cas, à utiliser les techniques standards de validation et vérification dans la pratique. Il s'avère que des aspects importants tels que l'hétérogénéité, la décentralisation et la distribution des applications sont difficilement pris en compte par les techniques actuelles de vérification et de validation. Ceci est particulièrement dû au fait que ces techniques ne permettent de considérer que des systèmes de taille réelle maniable, pris comme un seul bloc. Ceux-ci sont généralement traités en de temps raisonnable. Cependant, dès que les systèmes sont de complexité rédhibitoire, ce qui est le cas aujourd'hui avec les systèmes informatiques, ces techniques deviennent difficile à être mises en place. Dans ce cadre, les approches basées sur le raisonnement compositionnel [28, 29, 30] semblent être une des directions les plus prometteuses pour combler le fossé entre la complexité réaliste et croissante des systèmes et les limites des méthodes de vérification et de test.

L'idée directrice de ces approches émergentes, est d'utiliser le principe « diviser pour mieux régner » consistant à décomposer la correction d'un système de taille réaliste en petites tâches. Celles-ci ont ainsi des complexités maniables et leur correction peut être validée localement

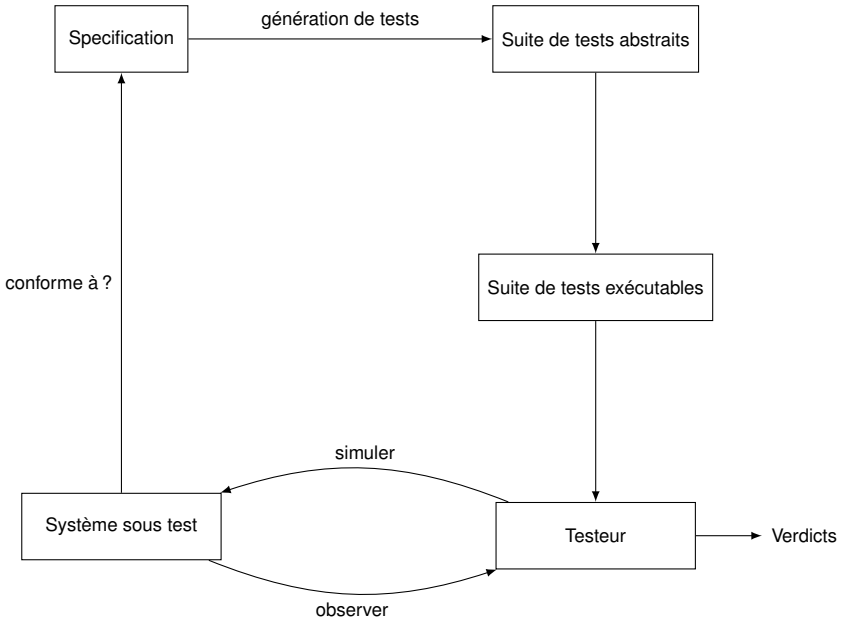


FIGURE 4 – Illustration du test de conformité

au niveau des composants. La correction du système entier est alors élaborée en combinant, suivant certaines règles de composition, les résultats de la vérification des sous-tâches sans vérifier l'ensemble du système.

La vérification compositionnelle a pour but de déduire les propriétés globales d'un système complexe à partir des propriétés de ses composants. Soient n composants dont les modèles de comportement sont $\mathcal{M}_1, \ldots, \mathcal{M}_n$ satisfaisant respectivement les propriétés locales $\phi_1, \ldots, \phi_n$ et op est l'opérateur de composition de comportement. La vérification compositionnelle consiste alors à vérifier si le système $op(\mathcal{M}_1, \ldots, \mathcal{M}_n)$ résultant de la composition de $\mathcal{M}_1, \ldots, \mathcal{M}_2$ satisfait une propriété globale ϕ. De cette façon, les techniques de vérification compositionnelle peuvent éviter l'explosion combinatoire, en décomposant les systèmes en petits sous-systèmes sur lesquels les techniques de vérification classiques peuvent être appliquées directement.

En ce sens, la théorie du test d'intégration vise à vérifier si la correction du système $\mathcal{C} = op(\mathcal{C}_1, \ldots, \mathcal{C}_2)$ est établie en utilisant la correction de chaque composant $\mathcal{C}_i$ où op est l'opérateur d'intégration considéré. Le problème de test d'intégration peut être alors vu comme suivant : étant données les implantations $\mathsf{iut}_1, \ldots, \mathsf{iut}_n$, les spécifications $\mathsf{spec}_1, \ldots, \mathsf{spec}_n$, un opérateur d'intégration op et une relation de conformité **ref** telles que $\mathsf{iut}_1, \ldots, \mathsf{iut}_n$ ont été testées pour être conforme à leurs spécifications $\mathsf{spec}_1, \ldots, \mathsf{spec}_n$ respectivement, par rapport à la relation **ref**, pouvons-nous conclure que leur composition $op(\mathsf{iut}_1, \ldots, \mathsf{iut}_n)$ est aussi conforme à la spécification globale $op(\mathsf{spec}_1, \ldots, \mathsf{spec}_n)$, par rapport à la relation **ref** ?

2 Contributions de la thèse

Comme indiqué dans l'introduction de ce chapitre, le travail que nous présentons a comme objectif la proposition d'une théorie générique pour la modélisation et la validation des systèmes informatiques complexes. Cette thèse est constituée de deux parties principales : la partie « modélisation » et la partie « validation par la technique du test de conformité ». La première partie consiste à proposer un cadre générique unifié pour la modélisation de systèmes standards à base d'états, en particulier, ceux qui sont dédiés à la génération de cas de tests de conformité.

Elle vise aussi à définir un ensemble minimaliste d'opérateurs d'intégration pour combiner des sous-systèmes. La deuxième partie consiste à proposer une théorie du test d'intégration visant à être en mesure de valider les systèmes informatiques complexes.

Nous nous intéressons plus particulièrement aux thèmes suivants :

Cadre formel unifié pour la modélisation des systèmes à base d'états

Basé sur l'approche proposée [9, 31] par *Barbosa* pour la modélisation de composants, nous proposons un cadre formel générique unifié pour la modélisation de systèmes à base d'états. Un tel système est alors représenté par une coalgèbre

$$(S, init, \alpha : S \longrightarrow T(\mathsf{Out} \times S)^{\mathsf{In}})$$

du foncteur F défini par $FX = H(\mathsf{Out} \times X)^{\mathsf{In}}$ dans la catégorie des ensembles **Set**, avec T une monade, et In et Out deux ensembles d'éléments qui désignent respectivement les entrées et les sorties du composant. Cette représentation coalgébrique des composants nous permettra :

- de considérer d'une manière générique une large famille de calculs usuels tels que le déterminisme, le non-déterminisme, la partialité, et d'autres. Ceci est dû à l'introduction de la notion d'une monade T dans la définition du composant qui nous permet de définir les composants d'une manière indépendante de toute structure de calcul [32, 14].

- de fournir un cadre formel générique dans lequel une large famille de formalismes de description peut être naturellement capturée. En particulier, il permet d'englober les formalismes standards dédiés à la génération de test de conformité tels que les automates de *Mealy* [33, 34], les systèmes de transitions étiquetées (LTS) [35, 36], les systèmes de transitions étiquetées à entrées et sorties (IOLTS) [37, 38, 39, 40], etc.

Sémantique unifiée des traces pour les systèmes à base d'états

Cette manière de décrire le comportement des composants nous permettra, suivant les travaux de *Rutten* [41], de définir un modèle de traces pour les composants par des fonctions causales. Ces fonctions sont des transformations de flux de données de la forme :

$$y = f(x, s, t)$$

où x, y et s sont respectivement l'entrée, la sortie et l'état du composant et t est le temps discret.

Ainsi, en prenant le bénéfice de définir le comportement des composants comme des fonctions de transfert causales, la définition d'un modèle de traces à partir de fonctions causales nous permettra de montrer l'existence d'une coalgèbre finale dans la catégorie des coalgèbres du foncteur $H(\mathsf{Out} \times \_)^{\mathsf{In}}$ en imposant certaines conditions suffisantes sur la monade T. Cette dernière est en effet utile pour définir les opérateurs d'intégration et rendre plus facile la preuve des théorèmes tout-au-long de la thèse.

Cette représentation du comportement du système forme la première étape vers un cadre unifié qui capte non seulement l'hétérogénéité des calculs usuels, mais aussi celle du temps (i.e. le temps discret et continu). En effet, dans cette thèse, nous nous restreignons au cas du temps discret pour lequel nous adresserons seulement l'hétérogénéité de formalismes utilisés pour la modélisation des composants. Cependant, il y a d'autres travaux réalisés dans le cadre de thèse de *B. Golden* [42] qui étendent notre cadre pour être en mesure de prendre en compte à la fois, le temps continu et le temps discret en utilisant les techniques de l'analyse non standard.

Opérateurs d'intégration

Nous proposons un ensemble d'opérateurs d'intégration pour construire de grands composants à partir de plus petits. Nous définissons ainsi deux opérateurs d'intégration de base : le *produit cartésien* et la *rétroaction*, et nous défendrons l'idée que les opérateurs d'intégration standards tels que les opérateurs de la composition séquentielle, concurrente ou parallèle et le produit synchronisé peuvent être obtenus par la composition de produit et de rétroaction. Cela nous conduira à définir inductivement des opérateurs d'intégration plus complexes, dont la sémantique sera considérée comme des foncteurs partiels dans la catégories des composants. Ainsi, un système sera construit d'une manière récursive et hiérarchique à l'aide de nos deux opérateurs d'intégration de bases à partir de systèmes élémentaires.

Théorie de test de conformité générique

De la généricité du formalisme développé dans cette thèse, nous proposons de définir une théorie du test de conformité générique pour les composants. Celle-ci sera *de facto* applicable à l'ensemble des formalismes à base d'états, instances de notre cadre tels que ceux dédiés à la génération des cas de tests. Il existe en effet plusieurs théorie du test de conformité dans la littérature [39, 43, 44, 45, 46, 47] qui se différencient par la relation de conformité considérée et les algorithmes de génération mis en place. Bien que la plupart de ces théories pourraient être adaptées à notre formalisme, nous proposons ici d'étendre l'approche définie dans [44] développée dans le cadre des IOLTS. L'intérêt de la théorie du test de conformité proposée dans [44] est qu'elle est fondée sur la relation de conformité *ioco* qui a montré son adaptation pour le test formel ainsi que pour la dérivation automatique de cas de test. De plus, les algorithmes de génération proposés dans [44] sont simples dans leur implantation et efficaces dans leur exécution.

Nous proposerons une technique du test de conformité basée sur la notion de l'objectif de test. Les objectifs de test seront définis comme des sous-arbres de l'arbre de l'exécution construit à partir de notre modèle de trace pour les composants. Les chemins dans ces sous-arbres caractérisent les comportements à tester. Ensuite, nous définirons un algorithme de génération de cas de tests à partir de l'objectif de tests comme [44]. Cet algorithme sera formalisé par un ensemble de règles d'inférence. Chaque règle est dédiée pour gérer une observation du système sous test (iut) ou une stimulation envoyée par le cas de test à l'iut. Ce processus de test produit un verdict. Nous trouverons les verdicts Pass, Fail, Inconc et WeakPass.

Cette théorie du test de conformité générique constitue la première étape vers le test de systèmes informatiques complexes définis comme un assemblage d'un ensemble des composants.

Théorie de test d'intégration

Nous proposons aussi de définir une théorie du test d'intégration qui vise à tester un système en supposant que ses composants sous-jacents ont déjà été testés séparément et sont corrects [30]. Le problème que nous adressons peut alors être vu comme suit. Si les composants d'un système sont conformes à leurs spécifications, que pouvons nous conclure sur la conformité du système par rapport à sa spécification ? Nous montrerons qu'une réponse positive à cette question ne peut être obtenue sans aucune hypothèse sur les spécifications et les implémentations.

Renforcer la qualité des composants par utilisation de techniques de projection

Nous proposons une approche pour renforcer la qualité des composants en tenant compte de leur implication dans le système global. L'idée sous-jacente consiste à montrer comment renforcer la correction de chaque composant impliqué dans le système global, en générant des cas de test plus appropriés pour eux. Ceci sera effectué en définissant un mécanisme de projection

qui, à partir d'un comportement du système global, aidera à se concentrer sur les comportements de sous-systèmes qui sont activés dans l'exécution du système [48].

Chapter I

Introduction

This chapter provides the context of the thesis and gives the motivation of the research presented in it: what are complex software systems and how can they be modeled and tested? It briefly introduces the "roadmap" for the chapters to follow by giving concepts and notions needed to answer this question and then outlines the contributions and the structure of the thesis.

1 Context

In spite of several decades of research, assuring the quality of software systems still represents a major and serious problem nowadays for the industry with respect to both results and costs [1]. This thesis comes within the scope of a proposal centered on a generic unified framework for both complex software systems modeling and testing.

In the following, we outline concepts and notions needed to achieve our goal.

1.1 System modeling

The work of this thesis proposes first to define a generic abstract complex formalism that models software components as concrete coalgebras for some **Set** endofunctors. Before concretely addressing the definition of this formalism, we will first succinctly explain:

1. What do we mean by components and complex systems?
2. The need of a generic formal description of component
3. Why use *Barbosa*'s component definition based on coalgebras and monads?

Components Components are receiving increasing attention as a level of design thanks to the great advantages they offer: modularity, re-usability, cost-effective solution for increasing heterogeneity and complexity, etc. Components are then designed, developed and validated separately in order to be widely used.

Several definitions of a component have been proposed in literature. *Szyperski* defines components as "binary units of independent production, acquisition, and deployment that interact to form a functioning system" [2]. *D'Souza and Wills* define a component as "a reusable part of software, which is independently developed, and can be brought together with other components to build larger units. It may be adapted but may not be modified" [3]. Despite the lack of a

unifying definition of what a component is, the design community agrees that any component definition should have the following characteristic properties [4]:

- It is a unit of encapsulation. It has internal state space, acting as the memory of the component;
- It is abstract enough to encapsulate a number of services through a public interface. There is no internal observable state, the only way to access its content is via its public interface;
- It persists and evolves in time, according to some given semantics;
- It can be deployed easily and independently;
- It is a unit of third-party composition, i.e. there is a possibility of interaction with other components during the overall computation of the whole system;
- It is equipped with input and output observation universes to ensure the flow of data during its execution.

Formal models of components The functionality of a component should then be defined at a higher level than the implemented code. The suitable way to formally address components is to consider them as an abstract representation omitting implementation details and only describing properties relevant to their functionalities. Such a representation is only made by specifying how inputs drive changes in component states and how outputs are produced. We classically talk about *black box* representation. Thus, the internal structure of a system (i.e. how it has been implemented) is ignored, only the functional or behavioral view of the system (i.e. what its requirements are) is considered. All possible communications between the system and the environment are then described via interface. An external observer can observe the system only through this interface. Such an external observer's view represents the complete input-output behaviour of the system. From the point of view of the environment, a black box can be better seen as a function achieving at a different time t (that can be discrete or continuous) some given system functional requirement which makes the inputs In of the system correspond to its outputs Out. This function depends on the current internal state $s \in S$ of the system, and then of the form $y = \mathcal{F}(x, s, t)$ where x, y and s are respectively the input, output and state of the component under consideration, and t is an instant of time.

A environmental view of a system can therefore be represented in Figure I.1.

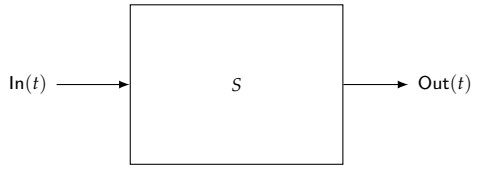


Figure I.1 – Black box view of a system

Complex systems So far we have seen that the component-based approach is considered a cost-effective solution for increased modularity and re-usability of system designs. Black box representations can cope with the increasing heterogeneity of components of different natures

disregarding underlying non-functional details of the components and their internal architectures. This means the global behaviour of a complex system would be less difficult to study when the underlying subsystems are considered as black box representations. In this way, a powerful approach to developing complex software systems could be to describe them in a hierarchical recursive way as interconnections of sub-systems (i.e. components). These sub-systems are then integrated through architectural connectors that are powerful tools to describe systems in terms of components and their interactions. Each subsystem can be then either a complex system itself or simple and elementary enough to be handled entirely.

Hence, the basic idea for modeling complex systems is that, at a high-level of abstraction, complex systems can be recursively decomposed into a set of subsystems, arriving at subsystems that can be completely handled. Such a formal hierarchical description of a complex system usually looks like the construction illustrated in Figure I.2.

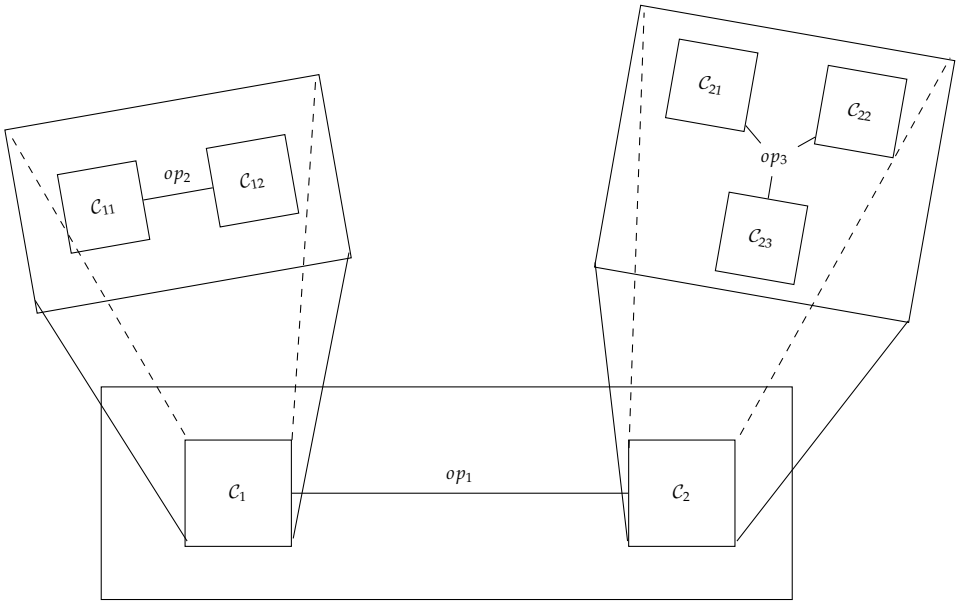


Figure I.2 – Compositional view of complex system

Composition is then used for assembling different sub-systems and then forming larger ones. Such a composition can be seen as an operation taking components as well as the interactive nature between them to provide a new more complex component. Hence, this recursive representation of a complex system as a composition of elementary components using integration rules allows us to separate the realization of different components and subsystems, and then makes the conception process more modular, i.e. the composition itself becomes a component that can be further composed as if it were itself an atomic component.

Suitable unified view of systems Sub-systems from which complex systems are made, may indeed be modeled using different specification formalisms depending especially on the scientific disciplines and the model of computations used to specify the interaction between the different elements of the system. At the moment, there are neither unified models nor unified tools that can be used to deal with such systems in all their generality. Nevertheless, one can observe that a common characteristic of sub-systems of most modern systems is that they can be considered, from a theoretical point of view and at a higher level of abstraction, as state-based

components whose behaviour can only be observed along their interface. Hence, state-based systems seem to be a natural formal representation of most concrete modern systems such as digital hardware components, software programs and distributed systems. It turns out it is becoming more and more difficult to express these systems as algebraic representations i.e. in terms of a set of complete constructions. Their characteristics, indeed, tend to be seen as observable entities rather than definable ones. Hence, the semantics of such systems is essentially *observational*. All that can be captured in their evolution is their *interaction* with the environment, that is to say, the possible input/output between them and the outside world. Inputs are indeed the signals or data received by the system, and outputs are the signals or data sent from it. In this setting, coalgebra theory and category theory provide all the necessary concepts to handle abstractly the observable behaviour of such systems.

Coalgebraic approach to modeling systems Due to this observational view of systems, coalgebras are increasingly used as an appropriate abstract model of state-based dynamical systems, looking for a unified definition of a model from which a great variety state-based models could be deduced such as: transition systems, automata, process calculi and classes in object-oriented languages [5, 6, 7, 8]. One of the major contributions in component coalgebraic modeling are *Barbosa*'s works [9, 10, 11, 12, 13]. Coalgebras have been used as a semantic model for software components for some endofunctor on **Set**. A component is then presented as an extended Mealy machine parametrized by a monad T, as is customary in functional programing [14], acting as a behaviour model. The monad T can indeed handle the different usual computational effects such as determinism, non-determinism, possible deadlock states, or exceptions and many more [15, 16]. *Barbosa*'s approach will be the cornerstone of our modeling. We will detail it in Section 1.2 of Chapter IV.

1.2 Validation and verification

Correctness and its importance With the increasing complexity of modern software systems, verification and validation techniques are becoming more and more important. Ensuring the functional correctness of a system's behaviour is becoming one of the major challenges nowadays in view of dramatic consequences in terms of human lives, economic loss, ecological problems, etc. caused when the faulty behaviour of a system occurs. System failures are indeed everywhere to the point that they are so familiar to us that we usually forget them. Some of them have little impact in our daily life, for example, when our mobile phone is malfunctioning or our video recorder reacts unexpectedly and wrongly to commands via the remote control. However, other errors have a huge impact. In history, there is a long list of typical software bugs that have caused catastrophes in terms of loss of human lives [17, 18] such as *Therac-25*, or loss of money such as *Ariane 5 crash* (costs about 500 million US dollars) and *Intel's Pentium floating-point division* (a loss of about 500 million dollars), etc. All these examples make verification and validation phases more expensive (in terms of time and money) than construction, in most designs.

Testing vs verification To reduce, as much as possible, the risk that a system fails, and so increase the level of confidence as well as decrease the gap between requirements specifying the functions that a system is expected to perform and the real implementation of the system, verification and validation techniques seem to be the best method. These techniques should be implemented automatically. Manual techniques such as *peer reviewing* have shown their unsuitability even impossibility in validating system functionalities as it is stated by *Wolper* in [49]:

"manual verification is at least as likely to be wrong as the program itself". On the other hand, automatic techniques are widely used in practice and well-accepted in industrial fields.

Three important techniques are mainly used: formal proofs, model checking and testing. *Formal proofs*, such as *Hoare logic* [19], consists in formally expressing the expected properties of the system and proving that these properties are correct by deduction from a set of axioms and inference rules. *Model checking* [20, 21] consists in automatically and algorithmically verifying whether system properties such as the absence of deadlocks (described in some appropriate logical formalism such as temporal logic) are satisfied by the system (described as a finite state model). *Testing* [22, 23] consists in running the system under test by providing it well-chosen input values (called tests), observing the value of its outputs, and then by comparing the observed behaviour with that desired, deducing whether the system is correct or not.

When talking about these techniques, the question "which technique is more effective?" directly arises. This question can be answered according mainly to both characteristics and complexity of the system under validation/verification. In general, verification and testing are best considered as complementary techniques. In practice, it turns out that they complement each other, and in most cases, there is a need to apply both to get the desired system quality. Indeed, though formal proof techniques are based on formal methods and considered to establish system correctness exhaustively, they are too hard and tedious to use in practice. In most cases, they cannot be automatically implemented, and need human help that renders the proof potentially incorrect. Though model checking intends to verify automatically that a system is free from errors, a close look at reality, however, reveals that it has its own weakness as we state below:

- it is only as good as the model of the system. In fact, it enables one to check exhaustively the correctness of a model of the system, but not the real system itself. The fact that a model has certain properties does not guarantee that the final realization also has the same properties;
- only desired or well-chosen properties are checked: there is no guarantee of the completeness of all system properties;
- it requires some expertise to be used (for example, some knowledge in logical temporal formalisms);
- it is hard to be used for systems of realistic size;
- it is impossible to use it in some cases, for instance, when there is no formal model of the system.

On the other hand, testing techniques can be applied directly to the real implementation, contrary to model checking or formal proofs that are based on mathematical models rather than on the real system. This advantage makes testing techniques more used in practice than other techniques, especially when verification methods seem impossible to be used due to the complexity of the system or its nature: there is no possibility to build a formal model of the real system (e.g physics devices, or it is proprietary). However, testing can never be complete as stated by *Dijkstra's* in [24]: "Testing can only show the presence of errors, never their absence". This is especially due to the too large set of all possible inputs to be submitted to the system. For example, suppose that a *calculator* only does both addition and subtraction operations for numbers ranging from 0 to 20. To test this calculator, it is required to execute both addition and subtraction operations on all possible combinations of integers in this range. This will require a total of $\Sigma_{i=1}^{20} 2^i \times 21^{i+1} (= 2 \times 21^2 \times \frac{1-(42)^{20}}{1-42})$ executions. Then, assuming that testing is done

on a computer that will take $10^{-7}s$ to input a subset of integers ranging from 0 to 20, execute a calculator operation, and check if the output (*i.e.* result of the requested operation) is correct, the testing execution process will take approximately $2 \times 21^2 \times \frac{1-(42)^{20}}{1-42} \times 10^{-7}s$ which is an order of years.

Thus, neither verification (both mathematical proofs and model checking) nor testing appear to be the perfect technique for proving the correctness of programs. They are often used as two related complementary activities and guide each other.

In this thesis, we are interested in testing, more precisely in defining a testing theory. Our reasons for focusing our attention on testing techniques to the detriment of verification ones are twofold:

1. First of all, our ambition is to define a mathematical framework to model complex software systems whose important characteristic is their large size. Yet, we have just seen that, verification methods are limited in their use for such systems.

2. The second reason for our choice is more practical. Indeed, verification methods essentially aim to prove that the systems verify a certain number of formal properties. These formal properties are expressed in a logic. Then, to address verification in our framework, we would have had to define a logic (temporal) over our formalism, and then establish a certain number of properties of this logic such as defining a calculus and proving that this latter is correct and complete, showing that the logic is adequate with respect to bisimulation (i.e. every couple of bisimilar states satisfy the same set of properties), studying conditions to preserve properties along integration operators, etc. However, all of this is often a long-term job that is difficult to address at the same time as the formalism definition. Nevertheless, the definition of such a logic is primordial, it is therefore naturally a part of future thesis work.

Software testing Software testing is a process which aims to strengthen the quality of systems through experimentations with the intent of finding errors in them. It can be used to reveal the presence of errors, but never their absence [22, 24, 25]. Many kinds of testing have been proposed that can be mainly classified in accordance with the following three characteristics (see Figure I.3):

– **level of code accessibility**: two testing techniques are commonly distinguished: *structural testing* (also known as *white box*) [23] in which system code is examined as to whether it works as expected, and *functional testing* (also known as *black box*) [26] in which system functionalities are examined without requiring any knowledge of the internal structure of the system.

– **level of abstraction**: three testing methods are commonly distinguished: *unit testing* in which software or hardware components that cannot be subdivided into other components are examined, *integration testing* in which a larger component built as a combination of a set of basic components is examined, and *system testing* in which the complete system is tested.

– **aspects we want to test**: there are different aspects used to test a system, for example *conformance testing* in which the behaviour of the implementation is tested to check whether it conforms to the specified behaviour, *robustness testing* in which an implementation reacts to unspecified, or "abnormal" environments, etc.

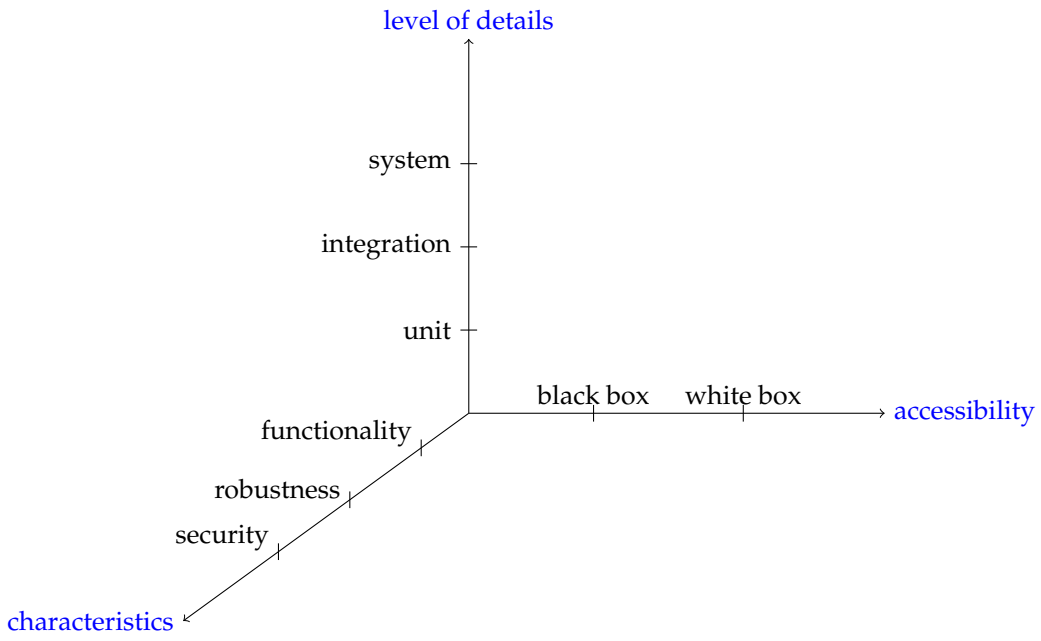


Figure I.3 – Classification of testing techniques

Since our goal, in this thesis, is to verify the correctness of behaviour of component-based systems, testing techniques we address here come then within the scope of both unit and integration functional testing based on conformance testing.

Conformance testing theory Conformance testing is a black box technique for checking correctness implementation against its specification by means of experimentations [27, 26]. It aims to check if the observable behaviour of the system under test conforms to a specification with respect to a particular conformance relation. The underlying idea consists in automatically generating test cases from a formal model a so-called *specification* of the system under test. These test cases are then executed by an external tester on the real system, and based on observed results, a verdict is generated indicating if the system was successful or not on the test of interest (or if the test was inconclusive).

Compositional testing As a matter of fact, the exponentially growing complexity and heterogeneity of today's systems give rise naturally to difficulties even the impossibility, in some cases, of using actual validation and verification methods in practice. It turns out important aspects for software systems such as heterogeneity, decentralized and networked applications, etc. are not well-supported by actual modeling and both verification and testing techniques. This is especially due to the fact that these techniques are limited to scalability of the complexity of actual software systems that are not only large but are also growing dramatically. As in a state-based components approach, compositional reasoning approaches [28, 29, 30] about system correctness is viewed as one of the most promising directions to bridge the gap between the increasing complexity of systems and actual verification and testing method limits.

The underlying idea behind these emergent approaches is to use "divide-and-conquer" approaches consisting in breaking down the correction of a complex system into smaller tasks of lower complexity that involve the correctness of its components. The conclusion for the whole system correctness is then drawn by combining the results from the verification of the subtasks

following certain compositional reasoning rules without verifying the whole system.

On the first hand, compositional verification aims to infer global properties of complex systems from properties of their components. It then consists in verifying, given n components with behaviour models $\mathcal{M}_1, \ldots, \mathcal{M}_n$ that satisfy local properties $\phi_1, \ldots, \phi_n$ respectively, and op is some composition behaviour operators, if the system $op(\mathcal{M}_1, \ldots, \mathcal{M}_n)$ resulting from the composition of $\mathcal{M}_1, \ldots, \mathcal{M}_n$ will satisfy a global property ϕ. In this way, compositional verification techniques avoid combinatorial explosion by decomposing systems into smaller subsystems to which classical verification techniques can be directly applied.

In this sense, compositional testing theory aims to check whether the correctness of the whole system $\mathcal{C} = op(\mathcal{C}_1, \ldots, \mathcal{C}_n)$ is established using the correctness of each components $\mathcal{C}_i$ where **op** is the integration operator of interest. Hence, the problem of compositional testing can be seen as follows: given implementation models $\mathsf{iut}_1, \ldots, \mathsf{iut}_n$, specifications $\mathsf{spec}_1, \ldots, \mathsf{spec}_n$, an integration operator **op** and a conformance relation **rel** such as $\mathsf{iut}_1, \ldots, \mathsf{iut}_n$ have been tested to be **rel**-correct according to their specifications $\mathsf{spec}_1, \ldots, \mathsf{spec}_n$ respectively, may we conclude that their composition $\mathbf{op}(\mathsf{iut}_1, \ldots, \mathsf{iut}_n)$ also **rel**-correct to the integrated specification $\mathbf{op}(\mathsf{spec}_1, \ldots, \mathsf{spec}_n)$?

Since, in this thesis, we propose a compositional method for the testing of component-based systems, we will only address the compositional testing approach.

2 Thesis overview

2.1 Thesis contributions

As stated in the introduction of this chapter, this thesis intends to contribute to two central topics: "modeling" and "testing" complex software systems. It then consists of two main parts. The first part, the *modeling*, intends to propose a generic unified framework from which most standard state-based systems, especially those which are dedicated to test case generation, can be deduced, and to define a minimalist set of operators to combine components. The second part, the *testing*, intends to propose generic compositional testing aimed at being able to validate complex software systems.

In particular, the contributions of this research work include:

Unifying formal framework for modeling state-based systems

Based on *Barbosa*'s approach [9, 31] for component modeling, we propose a formal generic unified framework for modeling state-based systems. Systems are then modeled as concrete coalgebras over the endofunctor $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$ on the category of sets where T is a monad, and In and Out are two sets of elements which denote respectively inputs and outputs of the component. Such coalgebraic models will allow us:

1. to abstract away computation situations such as determinism or non-determinism. Indeed, monads have been introduced in [32, 14] to consider in a generic way a wide range of computation structures such as partiality, non-determinism, etc. Hence, such a component representation allows us to define components independently of any computation structure.

2. to unify in a single framework a large family of state-based formalisms encompassing most standard formalisms dedicated to conformance test generation such as Mealy au-

tomata [33, 34], Labeled Transition Systems (LTS) [35, 36], Input-Output Labeled Transition Systems (IOLTS) [37, 38, 39, 40], etc.

Unifying trace semantics of state-based systems

This way of modeling component behaviour allows us, following *Rutten*'s works [41], to define a trace model over components by causal transfer functions. Such functions are dataflow transformations of the form: $y = \mathcal{F}(x, q, t)$ where x, y and q are respectively the input, output and state of the component under consideration, and t is discrete time.

Taking advantage of the definition of components behaviour as transfer functions, defining a trace model from causal functions allows us to show the existence of a final coalgebra in the category of coalgebras over *Barbosa*'s signature $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$ under some sufficient conditions of the monad T. This final coalgebra is indeed useful when defining the integration operators and makes easier theorem proofs throughout the thesis.

This representation of system behaviour then forms the first step towards a unified framework that captures not only different usual computations, but also time heterogeneity (i.e. both discrete and continuous times). Indeed, in this thesis we restrict ourselves to discrete time. Only formalism heterogeneity and component nature are addressed. However, there is other current work done in *B. Golden*'s thesis extending our framework to be able to take into account continuous time using non-standard analysis [42].

Calculus of operators

We propose a calculus of standard operators used to build larger components from smaller ones. We then define two basic integration operators, *product* and *feedback*, and defend the idea that most standard integration operators such as sequential, concurrent and parallel composition operators and synchronous product can be obtained by composition of product and feedback. This will lead us to define inductively more complex integration operators, the semantics of which will be partial functors over categories of components. Hence, a system will be built by a recursive hierarchical process through these integration operators from elementary systems or basic components.

Generic conformance testing theory

From the genericity of the formalism developed in this thesis, we propose to define a generic conformance testing theory for components. This testing theory will be applicable *de facto* to all state-based formalisms, instances of our framework. There are several conformance testing theories in literature [39, 43, 44, 45, 46, 47] that differ by the considered conformance relation and algorithms used to generate test cases. Although most of these theories could be adapted to our formalism, we propose here to extend the approach defined in [44] in the context of *IOSTS* formalism. The advantage of the testing theory proposed in [44] is that it is based on the conformance relation *ioco* that received much attention by the community of formal testing because it has shown its suitability for conformance testing and automatic test derivation. Furthermore, test generation algorithms proposed in [44] are simple in their implementation and efficient in their execution. Hence, test purposes will be defined as some particular subtrees of the execution tree built from our trace model for components. We will then define an algorithm which will generate test cases from test purposes. As in [44], this algorithm will be given by a set of inference rules. Each rule is dedicated to handle an observation from the system under test iut or a stimulation sent by the test case to the iut. This testing process leads to a verdict about implementation correctness with respect to its associated specification.

This generic conformance testing theory is the first step toward the testing of complex software systems made of interacting components.

Compositional testing theory

We further propose to define a compositional testing theory that aims to test an integrated system assuming that its underlying components have already been tested in isolation and are correct [30]. The problem that we address can be seen as follows: if single components of a system conform to their specifications, what can be said concerning conformance of the whole system in accordance with its specification? We will show that a positive answer to this question cannot be obtained without any assumption about both specifications and implementations.

Strengthening components quality by means of projection mechanism

We propose a component-based approach to strengthen the quality of components taking into account their involvement in the global system that encapsulates them. The underlying idea consists in showing how to re-enforce the correctness of each component involved in a global system by generating suitable test cases for them. This will be done by defining a projection mechanism that, from a behaviour of the global system, will help to focus on behaviours of sub-systems that typically occur in the whole system [48].

2.2 Plan of the thesis

The rest of the thesis is split into three parts organized as follows.

Part I provides all theoretical notions we use in this thesis. It contains two chapters where we introduce basic notions of both category and coalgebra theories.

In Chapter II we describe the different basic concepts and notions of category theory that serves us as formal background in the remaining chapters.

In Chapter III we describe the basic concepts and notions of coalgebras that will be useful in this thesis.

Part II is the core of the modeling part. It contains two chapters where we introduce a formal framework to model and unify state-based systems, and a calculus of operators to combine components in order to build larger components.

In Chapter IV we define a generic formal framework to define components and their trace models by means of causal transfer functions. We also present some theoretical results regarding the existence and uniqueness of a final model in the category of components.

In Chapter V we describe the basic integration operators: cartesian product and both relaxed and synchronous feedback as well as how to combine them to build more complex integration operators. We also define the notion of a system that will be the result of the composition of basic components using complex integration operators.

Part III is the core of the testing part. It contains three chapters where we introduce a brief overview of black-box conformance testing, our generic conformance testing for components defined in Part II, and our compositional testing for component-based systems.

In Chapter VI we describe the formal testing background used in the remaining chapters of the thesis. We then introduce the testing concepts presented in the *international standard* IS-9646: "Conformance Testing Methodology and Framework" [50, 51] and their formalization introduced in the setting of "Formal Methods in Conformance Testing" (*FMCT*) project [52].

In Chapter VII we present our generic conformance testing theory for components defined in Chapter IV. The work presented in this chapter is mainly inspired from the conformance testing theory developed in [44, 53]. In our theory of conformance testing, behaviors of specifications and implementations under test are modeled as components over the signature $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$, and the conformance relation is defined as a partial inclusion of their traces. We also present an algorithm for test case generation from the specification. This algorithm uses test purposes to eliminate the part of the specification which is not of interest for testing. The basic idea is to generate a finite computation tree from the specification of the system whose set of paths embodies the set of all behaviours of the specification. Test purposes are then used to tag a finite set of paths of interest of testing in the finite computation tree. Thereby, test cases will be generated by exploring the tagged finite computation tree starting from the initial state, and switching between sending stimuli to the implementation and waiting for output of the implementation according to certain inference rules as long as a verdict is not reached. At the end of the chapter, we show both correctness and completeness of generated test cases with respect to specifications and test purposes.

In Chapter VIII we present a compositional method for the testing of component-based systems described in Part II. The main idea is to apply "divide-and-conquer" approaches to global behaviour of a system from behaviours of its subsystems whose complexity is manageable. Instead of entirely testing the global system, the compositional testing approach we propose first decomposes the system under test into small subsystems and then tests each of them separately. The size of a subsystem is then smaller than the size of the whole system, and thus the risk of explosion of state space is significantly decreased.

We also propose a method for test purposes derivation for a given component of an integrated system from the behaviours of the components that constitute it. This last work is based on projection mechanisms that were first developed in [48].

Part I

Theoretical preliminaries

We introduce in this part the basic concepts of both category and coalgebra theories which will be useful throughout the thesis. We will rely on concrete examples to illustrate the expressive power of these two theories in systems modeling. Our aim therefore is to simply show that using these two theories, a unified formal framework for modeling state-based systems can be provided. For more interested readers, we refer to [5, 6, 54, 55, 56, 57, 58].

Chapter II

Category theory

1 Category 28
1.1 Category definition 28
1.2 Constructions of categories 29
1.3 Properties of arrows 30
2 Universal properties 31
2.1 Commutative diagrams 31
2.2 Initial and terminal objects 32
2.3 Product 32
2.4 Coproduct 33
2.5 Exponents 34
3 Functors and natural Transformations 36
3.1 Functors 36
3.2 Natural transformations 38
3.3 Heterogeneous Compositions 40
4 Monads in category theory 41
4.1 Definition 42
4.2 A working example 42
4.3 More examples 44
4.4 Category of Kleisli 46

Category theory is a branch of mathematics that was developed by *MacLane* and *Barr* [57, 58] in 1940. Since its appearance, it has been used as a powerful tool allowing the generalization of the concept of algebraic structures such as vectorial spaces, groups, topological spaces, etc., and the relations between them. In this chapter we describe the most fundamental concepts of category theory that will be used to define our formal framework.

1 Category

1.1 Category definition

Definition 1.1 *(**Category**) A **category** $\mathbb{C}$ consists of:*

- *A collection* $\mathsf{Obj}(\mathbb{C})$ *of objects;*
- *For each pair of objects* $A, B \in \mathsf{Obj}(\mathbb{C})$*, a collection* $\mathsf{Hom}(A, B)$ *of arrows (or morphisms or also maps)* $f : A \longrightarrow B$ *from* A *to* B*;*
- A *is the domain and* B *is the codomain of* $f : A \longrightarrow B$*;*
- *For each object* $A \in \mathsf{Obj}(\mathbb{C})$*, an identity arrow* $\mathsf{id}_A : A \longrightarrow A$*;*
- *For each pair of arrows* $f : A \longrightarrow B$ *and* $g : B \longrightarrow C$*, a composite arrow* $g \circ f : A \longrightarrow C$*.*

These data have to satisfy the following laws:

- *Associativity: if* $A \xrightarrow{f} B \xrightarrow{g} C \xrightarrow{h} D$*, then*

$$(h \circ g) \circ f = h \circ (g \circ f)$$

- *Identity composition: if* $f : A \longrightarrow B$*, then*

$$f \circ \mathsf{id}_A = \mathsf{id}_B \circ f$$

Remark: in case of ambiguity, the operations of composition, the identities and the set of arrows are denoted by the name of the corresponding category i.e. we write: $\mathsf{id}^{\mathbb{C}}$, $f \circ^{\mathbb{C}} g$ and $\mathsf{Hom}^{\mathbb{C}}$.

Example 1.1 *As a concrete example, we consider a set of objects* $\mathsf{Obj} = \{A, B\}$ *and a set of arrows* $\mathsf{Hom} = \{f : A \longrightarrow B, h : B \longrightarrow A, g : B \longrightarrow A, \mathsf{id}_A, \mathsf{id}_B\}$ *that are depicted in Figure II.1 (on the left side) such that:*

$$g \circ f = h \circ f = \mathsf{id}_A \quad \textit{and} \quad f \circ h = f \circ g = \mathsf{id}_B$$

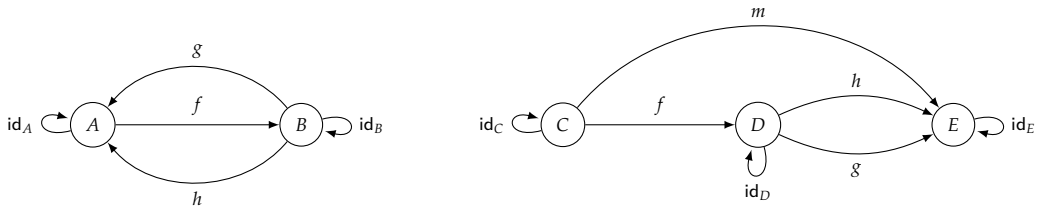


Figure II.1 – Examples of categories

These objects and arrows do not form a category. That is due to the fact that the property of associativity is not verified: there are three arrows f_1, f_2 *and* f_3 *such that the couples* (f_1, f_2) *and* (f_2, f_3) *can be composed and* $(f_3 \circ f_2) \circ f_1 \neq f_3 \circ (f_2 \circ f_1)$*. For example:*

$$\begin{aligned} (h \circ f) \circ g &= \mathsf{id}_A \circ g \\ &= g \end{aligned}$$

$$\begin{aligned} h \circ (f \circ g) &= h \circ \mathsf{id}_B \\ &= h \end{aligned}$$

Similarly to above, it is not hard to check that the objects and arrows that are depicted in Figure II.1 (on the right side) and satisfy $(h \circ f = g \circ f = m)$ *form a category.*

Example 1.2 *Table II.1 shows some mathematical structures that can be perceived as special types of examples of categories. All of these examples are categories whose objects are sets with particular mathematical structure and whose arrows are functions preserving that structure, so-called* morphisms.

Category	Objects	Arrows
Set	sets	applications
Poset	ordered sets	monotone applications
Grp	groups	homomorphisms
Top	topological spaces	continuous applications
$\mathbf{Vect_k}$	vectorial K-spaces	linear applications

Table II.1 – Examples of categories

A category is **small** if its objects and arrows constitute sets; otherwise it is **large**.

1.2 Constructions of categories

In this subsection, we describe some usual constructions of categories.

Definition 1.2 *(**Subcategory**) Let $\mathbb{C}$ and $\mathbb{D}$ be two categories. We say that $\mathbb{C}$ is a **subcategory** of $\mathbb{D}$ if:*

- *All the objects of $\mathbb{C}$ are objects of $\mathbb{D}$: $\mathsf{Obj}(\mathbb{C}) \subseteq \mathsf{Obj}(\mathbb{D})$;*
- *All the arrows of $\mathbb{C}$ are arrows of $\mathbb{D}$: for each pair of objects $A, B \in \mathsf{Obj}(\mathbb{C}), \mathsf{Hom}^{\mathbb{C}}(A,B) \subseteq \mathsf{Hom}^{\mathbb{D}}(A,B)$;*
- *If A is an object of $\mathbb{C}$ then its identity id_A in $\mathbb{C}$ is in $\mathbb{D}$: for each object $A \in \mathbb{C}, \mathsf{id}_A^{\mathbb{C}} = \mathsf{id}_A^{\mathbb{D}}$;*
- *If $f : A \longrightarrow B$ and $g : B \longrightarrow C$ in $\mathbb{C}$, then the composite in $\mathbb{C}$ $f \circ g$ is in $\mathbb{D}$ and is the composite in $\mathbb{D}$: $f \circ^{\mathbb{C}} g = f \circ^{\mathbb{D}} g$.*

We say that $\mathbb{C}$ is a *full subcategory* of $\mathbb{D}$ if $\mathbb{C}$ is a subcategory of $\mathbb{D}$ such that for each pair of objects $A, B \in \mathsf{Obj}(\mathbb{C}), \mathsf{Hom}^{\mathbb{C}}(A,B) = \mathsf{Hom}^{\mathbb{D}}(A,B)$.

From any category $\mathbb{C}$, it is possible to construct another category called dual of $\mathbb{C}$ by reversing all the arrows. That is to say, the source and the target of each arrow of $\mathbb{C}$ have to be reversed.

Definition 1.3 *(**The dual of a category**) Let $\mathbb{C}$ be a category. The **dual** (or **opposite**) of $\mathbb{C}$, noted $\mathbb{C}^{op}$, is the category whose objects and arrows are the objects and arrows of $\mathbb{C}$, but the domain and the codomain of each arrow have been reversed. Then we have:*

$$\begin{array}{lll}
\mathsf{domain}(f) = A \text{ in } \mathbb{C}^{op} & \text{if} & \mathsf{codomain}(f) = A \text{ in } \mathbb{C} \\
\mathsf{codomain}(f) = A \text{ in } \mathbb{C}^{op} & \text{if} & \mathsf{domain}(f) = A \text{ in } \mathbb{C} \\
f = \mathsf{id}_A \text{ in } \mathbb{C}^{op} & \text{if} & f = \mathsf{id}_A \text{ in } \mathbb{C} \\
h = g \circ f \text{ in } \mathbb{C}^{op} & \text{if} & h = f \circ g \text{ in } \mathbb{C}
\end{array}$$

Example 1.3 *If P is a poset, then the dual of the category P is the category determined by a poset P^{op}: if $(x,y) \in \mathsf{Hom}^{P}$, then $(y,x) \in \mathsf{Hom}^{P^{op}}$. For instance, the dual of the poset $(\mathbb{Z}, \leq)$ is $(\mathbb{Z}, \geq)$.*

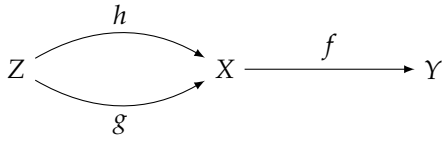
1.3 Properties of arrows

One of the principle characteristics of the theory of categories is its powerful unifying concepts across many branches of mathematics. This characteristic often leads to categorical definitions that do not involve the objects of a category in the sense that the property of one object is entirely defined in terms of the external interactions of that object with other objects.

In the following, we give the categorical definitions of some concepts that are standard in the set theory such as injectivity (monomorphism), surjectivity (epimorphism) and bijectivity (isomorphism). These definitions will be only defined with the concept of arrows of a category, without involving its objects (i.e. objects make no sense of such definitions).

Monomorphisms. A function $f : X \longrightarrow Y$ in **Set** is *injective* if for any element y in Y, there is at most one element x in X such that $y = f(x)$. This concept of injective function can be easily redefined in the category theory without using elements of X or Y. This is done by replacing the elements of X by arbitrary arrows into X.

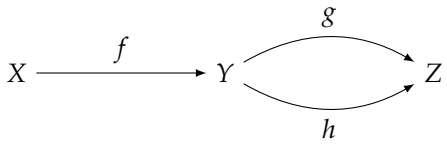
Definition 1.4 *(**Monomorphism**) Let $\mathbb{C}$ be a category and $f : X \longrightarrow Y$ be a arrow of $\mathbb{C}$. We say that f is a **monomorphism (or monic)** if for any object $Z \in \mathsf{Obj}(\mathbb{C})$ and any arrows $g, h : Z \longrightarrow X \in \mathsf{Hom}^{\mathbb{C}}(Z, X)$: if $f \circ g = f \circ h$, then $g = h$.*



We also say that f is left-cancellative.

Epimorphisms. A function $f : X \longrightarrow Y$ in **Set** is *surjective* if for any element in Y, there is at least one antecedent. Like injective functions, surjective functions can also only be redefined in terms of arrows as follows:

Definition 1.5 *(**Epimorphism**) Let $\mathbb{C}$ be a category and $f : X \longrightarrow Y$ be a morphism of $\mathbb{C}$. We say that f is a **epimorphism (or epic)** if for any object $Z \in \mathsf{Obj}(\mathbb{C})$ and any arrows $g, h : Y \longrightarrow Z \in \mathsf{Hom}^{\mathbb{C}}(Y, Z)$: if $g \circ f = h \circ f$, then $g = h$.*



We also say that f is right-cancellative.

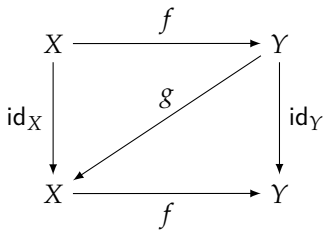
Isomorphism. The word "isomorphic" is usually used in mathematics to express distinction of objects in form, but not in number. Before stating its categorical definition, we give the following definition:

Definition 1.6 *(**Section and retraction**) Let $\mathbb{C}$ be a category and $f : X \longrightarrow Y \in \mathsf{Hom}^{\mathbb{C}}(X, Y)$:*

- *We say that an arrow $r : Y \longrightarrow X$ is a **section** (or **right inverse**) of f, if $f \circ r = \mathsf{id}_Y$;*
- *We say that an arrow $l : Y \longrightarrow X$ is a **retraction** (or **left inverse**) of f, if $l \circ f = \mathsf{id}_X$.*

Definition 1.7 *(**Isomorphism**) Let $\mathbb{C}$ be a category and $f : X \longrightarrow Y$ be an arrow of $\mathbb{C}$. We say that f is an **isomorphism** if there exists an arrow $g : Y \longrightarrow X \in \mathsf{Hom}^{\mathbb{C}}(Y, X)$ such that g is both a section and a retraction of f i.e. :*

$$f \circ g = \mathsf{id}_Y \text{ and } g \circ f = \mathsf{id}_X$$



A function $f : X \longrightarrow Y$ in **Set** is *bijective* if for every element $y \in Y$, there is exactly one element $x \in X$ such that $f(x) = y$. Alternatively, a function f is *bijective* if and only if there exists function $f^{-1} : Y \longrightarrow X$ such that their compositions $f^{-1} \circ f = \mathsf{id}_Y$ and $f \circ f^{-1} = \mathsf{id}_X$. Then, bijective functions in **Set** are obviously isomorphisms. However, a morphism which is both a monomorphism and an epimorphism is not necessarily an isomorphism (it is a bimorphism) unlike a bijection function which is both injective and surjective.

Example 1.4 *The inclusion of $\mathbb{Z}$ into $\mathbb{Q}$ in the category of* abelian groups **Ab** *is both a monomorphism and an epimorphism, but it is not isomorphism.*

If there is such an isomorphism from X to Y, one often writes $X \cong Y$.

2 Universal properties

Universal properties are the properties that ensure existence and uniqueness of a given construction under some conditions. They are generally formalized as follows:

for any ... there is a unique ... such that ...

2.1 Commutative diagrams

Categorical properties are often expressed in terms of commuting diagrams. Informally, a diagram in a category $\mathbb{C}$ is an oriented graph whose nodes are labeled with objects of $\mathbb{C}$ and whose edges are labeled with arrows in $\mathbb{C}$ in such a way that source and target nodes of an edge are labeled with source and target objects of the labeling arrow.

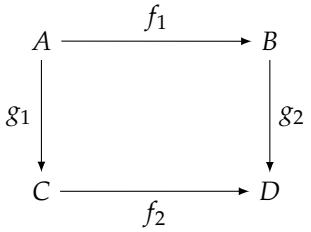
Definition 2.1 *(**Category diagram**) A **diagram of a category** $\mathbb{C}$ is an oriented graph $G = (V, E)$ where:*

- *$V \subseteq \mathsf{Obj}(\mathbb{C})$ and V has a finite cardinality;*
- *E is a set of directed edges. The edges of an object$X$ to an object Y are arrows from X to Y and are finite in number.*

Definition 2.2 *(**Commutative diagram**) Let $\mathbb{C}$ be a category and $G = (V, E)$ be a diagram in $\mathbb{C}$.*

- *A path in G is a non-empty sequence of edges such that the target node of each edge is the source node of the next edge in the sequence;*
- *G is said to **commute** if, for every pair of nodes $X, Y \in V$, all paths between them lead to the same result by composition.*

Hence, verifying in a category $\mathbb{C}$ that the diagram below is commutative is equivalent to verifying that $g_2 \circ f_1 = f_2 \circ g_1$.



2.2 Initial and terminal objects

Definition 2.3 *(**Initial et terminal objects**) Let $\mathbb{C}$ be a category. An object I of $\mathbb{C}$ is called **initial** if there is exactly one arrow $I \longrightarrow X$ for each object X of $\mathbb{C}$. An object T of $\mathbb{C}$ is called **terminal** if there is exactly one arrow $X \longrightarrow T$ for each object X of $\mathbb{C}$. An object that is both initial and terminal is called a **zero**.*

Proposition 2.1

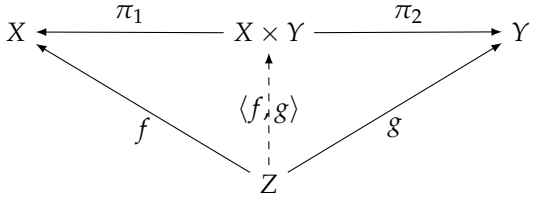
1. *If a category has an initial object, it is unique up to isomorphism.*
2. *If a category has a terminal object, it is unique up to isomorphism.*

Note: The expression "up to isomorphism" means that if we have two constructions of such an object, there is one and only way to convert one into the other and vice-versa.

Example 2.1 *In* **Set**, *the initial object is the empty set while any singleton set is a final object. As already stated, when reasoning at the level of a category $\mathbb{C}$, its objects are abstracted away and their internal structures are not available. Hence, equality between the objects of a category makes no sense, the real criterion to distinguish them is the isomorphism. Then, objects that cannot be distinguished in such a category are called isomorphic. For instance, the sets $\{\mathsf{Hello}\}, \{1000\}, \{*\}$ or $\{\mathsf{Ok}\}$ are all isomorphic: just renaming all elements of a set does not really give us another set. From this point of view, the symbol $\mathbf{1} = \{*\}$ has been chosen to denote the isomorphism class of all singletons. In other words, $\mathbf{1}$ is the final object in* **Set**.

2.3 Product

Definition 2.4 *(**Product**) Let $\mathbb{C}$ be a category. The **product of two objects** $X, Y \in \mathsf{Obj}(\mathbb{C})$ is a new object $X \times Y \in \mathsf{Obj}(\mathbb{C})$ with two projection morphisms $\pi_1 : X \times Y \longrightarrow X$ and $\pi_2 : X \times Y \longrightarrow Y$ which are universal: for each pair of morphisms $f : Z \longrightarrow X$ and $g : Z \longrightarrow Y$ in $\mathbb{C}$, there is a unique morphism $\langle f, g \rangle : Z \longrightarrow X \times Y$ in $\mathbb{C}$, making the following diagram commute*[1]*:*



1. The dashed notation is used to express the uniqueness of morphisms.

Proposition 2.2 *Let $\mathbb{C}$ be a category. Let $X, Y \in \mathsf{Obj}(\mathbb{C})$. If A is an object of $\mathbb{C}$, $\pi_1 : A \longrightarrow X$ and $\pi_2 : A \longrightarrow Y$ two arrows of $\mathbb{C}$, and $(f, g) \mapsto \langle f, g \rangle$ a function that associates to each pair of arrows $f : Z \longrightarrow X$ and $g : Z \longrightarrow Y$ the arrow $Z \longrightarrow A$ such that the following equations:*

- $\pi_1 \circ \langle f, g \rangle = f$
- $\pi_2 \circ \langle f, g \rangle = g$
- $\langle \pi_1 \circ \pi_2 \rangle = \mathsf{id}_A$
- $\langle f, g \rangle \circ \varphi = \langle f \circ \varphi, g \circ \varphi \rangle$

are satisfied for every objects Z and Z', and for every arrows $f : Z \longrightarrow X, g : Z \longrightarrow Y$ and $\varphi : Z' \longrightarrow Z$, then (A, π_1, π_2) is a product of X and Y in $\mathbb{C}$.

Example 2.2 *The* cartesian product *of two sets X and Y is generally defined by:*

$$X \times Y = \{(x, y) \mid x \in X \text{ and } y \in Y\}$$

This product can be easily redefined as an instance of the general notion of product of two objects of such a category. It is therefore defined by a triplet $(X \times Y, \pi_1, \pi_2)$ such that π_1 and π_2 are projection functions defined by:

$$\begin{array}{rccc} \pi_1 : & X \times Y & \longrightarrow & X \\ & (x, y) & \mapsto & x \end{array} \qquad \begin{array}{rccc} \pi_2 : & X \times Y & \longrightarrow & Y \\ & (x, y) & \mapsto & y \end{array}$$

Given a set Z and two applications $f : Z \longrightarrow X$ and $g : Z \longrightarrow Y$. It is not hard to see that there is a unique function $\langle f, g \rangle : Z \longrightarrow X \times Y$ such that $\pi_1 \circ \langle f, g \rangle = f, \pi_2 \circ \langle f, g \rangle = g, \langle \pi_1 \circ \pi_2 \rangle = \mathsf{id}_{X \times Y}$ and $\langle f, g \rangle \circ h = \langle f \circ h, g \circ h \rangle : Z' \longrightarrow X \times Y$ where $h : Z' \longrightarrow X \times Y$.

The product does not only apply to sets, but also to functions. For functions $f : X \longrightarrow X'$ and $g : Y \longrightarrow Y'$, one has:

$$f \times g : X \times X' \longrightarrow Y \times Y' \text{ given by } (x, y) \mapsto (f(x), g(x))$$

This can also be differently defined in terms of projection functions as follows:$f \times g = \langle f \circ \pi_1, g \circ \pi_2 \rangle$. It is easily verified that the operation $\times$ on functions satisfies:

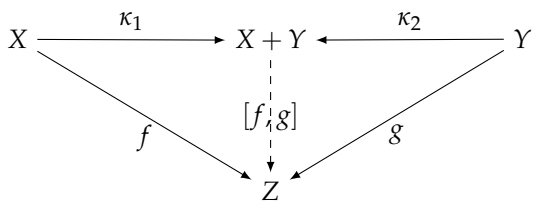
$$\mathsf{id}_X \times \mathsf{id}_Y = \mathsf{id}_{X \times Y} \text{ and } (f \circ h) \times (g \circ k) = (f \times g) \circ (h \times k)$$

Therefore, $\times$ applies both sets and functions, and preserves domains, identities and composition.

2.4 Coproduct

Every notion in category theory has its dual. Hence, the dual of product is *coproduct*.

Definition 2.5 ***(Coproduct)*** *Let $\mathbb{C}$ be a category. The* ***coproduct of two objects*** *X and Y of $\mathsf{Obj}(\mathbb{C})$ is a new object $X + Y \in \mathsf{Obj}(\mathbb{C})$ with two coprojection morphisms $\kappa_1 : X \longrightarrow X + Y$ and $\kappa_2 : Y \longrightarrow X + Y$ which are universal: for each pair of arrows $f : X \longrightarrow Z$ and $g : Y \longrightarrow Z$ in $\mathbb{C}$, there is a unique morphism $[f, g] : X + Y \to Z$ in $\mathbb{C}$, making the following diagram commute:*



Proposition 2.3 *Let $\mathbb{C}$ be a category. Let $X, Y \in \mathsf{Obj}(\mathbb{C})$. If A is an object of $\mathbb{C}$, $\kappa_1 : X \longrightarrow A$ and $\kappa_2 : Y \longrightarrow A$ two arrows of $\mathbb{C}$, and $(f, g) \mapsto [f, g]$ a function that associates to each pair of arrows $f : X \longrightarrow Z$ and $g : Y \longrightarrow Z$, the arrow $A \longrightarrow Z$ such that the following equations:*

- $[f, g] \circ \kappa_1 = f$
- $[f, g] \circ \kappa_2 = g$
- $[\kappa_1 \circ \kappa_2] = \mathsf{id}_A$
- $\varphi \circ [f, g] = [\varphi \circ f, \varphi \circ g]$

are satisfied for every objects Z and Z', and for every arrows $f : X \longrightarrow Z, g : Y \longrightarrow Z$ and $\varphi : Z \longrightarrow Z'$, then (A, κ_1, κ_2) is a coproduct of X and Y in $\mathbb{C}$.

Example 2.3 *The* coproduct *(or* sum*, or* disjoint union*) of two sets X and Y is generally defined by:*

$$X + Y = \{\langle 0, x\rangle \mid x \in X\} \cup \{\langle 1, y\rangle \mid y \in Y\}$$

The components 0 and 1 are useful to force this union to be disjoint. They can be considered as tags used to recognize the elements of X and Y in $X + Y$. Like the cartesian product, the coproduct of two sets can be seen as an instance of the general definition of coproduct described above. It is therefore defined by a triple $(X + Y, \kappa_1, \kappa_2)$ such that κ_1 and κ_2 are coprojection functions defined by:

$$\begin{array}{rccc} \kappa_1 : & X & \longrightarrow & X + Y \\ & x & \mapsto & \langle 0, x\rangle \end{array} \qquad \begin{array}{rccc} \kappa_2 : & Y & \longrightarrow & X + Y \\ & Y & \mapsto & \langle 1, y\rangle \end{array}$$

Given a set Z and two applications $f : X \longrightarrow Z$ and $g : Y \longrightarrow Z$. It is not hard to verify that there is a unique function $[f, g] : X + Y \longrightarrow Z$ such that $[f, g] \circ \kappa_1 = f, [f, g] \circ \kappa_2 = g, [\kappa_1 \circ \kappa_2] = \mathsf{id}_{X+Y}$ and $h \circ [f, g] = [h \circ f, h \circ g] : Z' \to X + Y$ where $h : Z \longrightarrow Z'$.

The coproduct of two functions $f : X \longrightarrow X'$ and $g : Y \longrightarrow Y'$ is $f + g : X + X' \longrightarrow Y + Y'$ with:

$$(f + g)(u) = \begin{cases} \langle 0, f(x)\rangle & \text{if} \quad u = \langle 0, x\rangle \\ \langle 1, g(y)\rangle & \text{if} \quad u = \langle 1, y\rangle \end{cases}$$

This can also be differently defined in terms of coprojection functions as follows: $f + g = [\kappa_1 \circ f, \kappa_2 \circ g]$. It is easy to verify that the operation $+$ on functions preserves identities and composition:

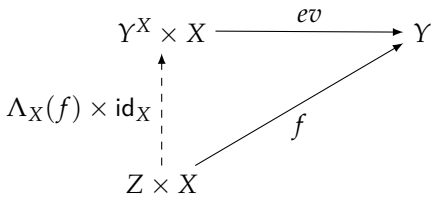
$$\mathsf{id}_X + \mathsf{id}_Y = \mathsf{id}_{X+Y} \text{ and } (f \circ h) + (g \circ k) = (f + g) \circ (h + k)$$

2.5 Exponents

Definition 2.6 *(**Exponent**) Let $\mathbb{C}$ be a category with products $\times$. The **exponent**[2] of two objects $X, Y \in \mathsf{Obj}(\mathbb{C})$ is a new object $Y^X \in \mathsf{Obj}(\mathbb{C})$ with an evaluation morphism:*

$$ev : Y^X \times X \longrightarrow Y$$

such that: for each morphism (sometimes called currying) $f : Z \times X \longrightarrow Y$ in $\mathbb{C}$, there is a unique abstraction morphism $\Lambda_X(f) : Z \longrightarrow Y^X$, making the diagram commute:

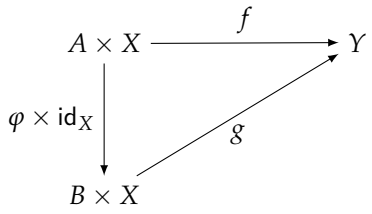


2. The object Y^X is read: *Y is exponent of X.*

Let us note here that the arrow $\Lambda_X : Z \to Y^X$ forms the terminal object in a category whose objects are the diagrams of the form:

$$A \times X \xrightarrow{f} Y$$

and the arrows from an object $A \times X \xrightarrow{f} Y$ to an object $B \times X \xrightarrow{g} Y$ are the arrows $\varphi : A \longrightarrow B$ such that the following diagram commutes:



Proposition 2.4 *Let $\mathbb{C}$ be a category with products $\times$. Let $X, Y \in \mathsf{Obj}(\mathbb{C})$. If Z is an object of $\mathbb{C}$, $ev : Z \times X \longrightarrow Y$ an arrow of $\mathbb{C}$ and $f \mapsto \Lambda_X(f)$ a function that associates to every arrow $f : A \times X \longrightarrow Y$, an arrow $A \longrightarrow Z$ such that the following equations:*

- $ev \circ (\Lambda_X(f) \times \mathsf{id}_X) = f$
- $\Lambda_X(ev) = \mathsf{id}_Z$
- $\Lambda_X(f) \circ \varphi = \Lambda_X(f \circ (\varphi \times \mathsf{id}_X))$

are satisfied for every object A and B, and for every arrow $f : A \times X \longrightarrow Y$ and $\varphi : B \longrightarrow A$, then (Z, ev) is an exponent of Y and X in $\mathbb{C}$.

Example 2.4 *Let us define the exponent of two sets X and Y as an instance of the general categorical definition of exponents given above. So, given two sets X and Y, the set of functions from X to Y can embody in the object Y^X. This set is defined by:*

$$Y^X = \{f \mid f : X \longrightarrow Y \text{ is a total function}\}$$

The evaluation function ev can be considered as the following application:

$$\begin{array}{rccc} ev : & Y^X \times X & \longrightarrow & Y \\ & (f, x) & \mapsto & f(x) \end{array}$$

which sends each pair (f, x) to $f(x)$, the value of f for x.

Now, let us consider the function $f : Z \times X \longrightarrow Y$ and then define the abstraction function $\Lambda_X(f)$:

$$\begin{array}{rccc} \Lambda_X(f) : & Z & \longrightarrow & Y^X \\ & z & \mapsto & (x \mapsto f(z, x)) \end{array}$$

which sends $z \in Z$ to the function $x \mapsto f(z, x)$ that maps $x \in X$ to $f(z, x) \in Y$.

Let us verify that the exponent of two sets X and Y is defined by the triple (Y^X, ev, Λ_X). For this, we need to prove the following equations:

$$ev \circ (\Lambda_X(f) \times \mathsf{id}_X) = f \tag{II.1}$$

$$\Lambda_X(ev) = \mathsf{id}_{Y^X} \tag{II.2}$$

$$\Lambda_X(f) \circ \varphi = \Lambda_X(f \circ (\varphi \times \mathsf{id}_X)) \tag{II.3}$$

hold for every arrow $f : Z \times X \longrightarrow Y$ *and* $\varphi : Z' \longrightarrow Z$.

For Equation II.1, one has: $ev \circ (\Lambda_X(f) \times \mathsf{id}_X)(z, x) = ev((x \mapsto f(z, x)), x) = f(z, x)$

For Equation II.2, one has: $\Lambda_X(ev)(f) = (x \mapsto ev(f, x)) = f(x)$

For Equation II.3, one has:
$\Lambda_X(f) \circ \varphi(z') = \Lambda_X(f)(\varphi(z')) = (x \mapsto f(\varphi(z'), x))$
and $\Lambda_X(f \circ (\varphi \times \mathsf{id}_X))(z') = (x \mapsto f \circ (\varphi \times \mathsf{id}_X))(z', x) = (x \mapsto f(\varphi(z'), x))$

Thus, the exponent of two sets X *and* Y *is defined by the triple* (Y^X, ev, Λ_X).

3 Functors and natural Transformations

3.1 Functors

The notion of functors has been defined as a generalization of functions in category theory. Hence, functors are structure-preserving maps between categories. They transpose the objects and arrows of a category to another one. A functor F from a category $\mathbb{C}$ to a category $\mathbb{D}$ then associates to each object $X \in \mathsf{Obj}(\mathbb{C})$ an object $Y \in \mathsf{Obj}(\mathbb{D})$ and to each arrow $f : X \longrightarrow Y \in \mathsf{Hom}^{\mathbb{C}}(X, Y)$ an arrow $F(f) \in \mathsf{Hom}^{\mathbb{D}}(F(X), F(Y))$, while preserving identity and composition of arrows.

Definition 3.1 *(**Functor**) Let* $\mathbb{C}$ *and* $\mathbb{D}$ *be two categories. A* ***functor*** $F : \mathbb{C} \longrightarrow \mathbb{D}$ *consists of:*

- *A function* $F_o : \mathsf{Obj}(\mathbb{C}) \longrightarrow \mathsf{Obj}(\mathbb{D})$, *called* object function, *that associates to every object* $X \in \mathsf{Obj}(\mathbb{C})$, *an object* $F_o(X) \in \mathsf{Obj}(\mathbb{D})$ *and*
- *A family of functions* $F_{X,Y} : \mathsf{Hom}(X, Y) \longrightarrow \mathsf{Hom}(F_o(X), F_o(Y))$ *indexed by couples* (X, Y) *of objects in* $\mathsf{Obj}(\mathbb{C})$. $F_{X,Y}$ *is called* arrow function *and associates to every arrow* $f : X \longrightarrow Y \in \mathsf{Hom}^{\mathbb{C}}(X, Y)$ *an arrow* $F_{X,Y}(f) : F_o(X) \longrightarrow F_o(Y)$ *in* $\mathsf{Hom}^{\mathbb{D}}(F_o(X), F_o(Y))$.

Both identity and composition properties have to be satisfied:

- *For every object* $X \in \mathsf{Obj}(\mathbb{C})$, $F_{X,X}(\mathsf{id}^{\mathbb{C}}_X) = \mathsf{id}^{\mathbb{D}}_{F_o(X)}$;
- *For every pair of morphisms* $f : X \longrightarrow Y$ *and* $g : Y \longrightarrow Z$, $F_{X,Z}(g \circ^{\mathbb{C}} f) = F_{Y,Z}(g) \circ^{\mathbb{D}} F_{X,Y}(f)$.

A functor is called endofunctor *if it maps a category to itself.*

In the following, the indexes o and $_{X,Y}$ of F_o and $F_{X,Y}$ will be clarified when this is necessary in order to avoid any complications.

Example 3.1 *(**Forgetful functors**) Forgetful functors are functors which send objects of a category to objects of another category by forgetting certain properties of objects. For example, the functor* F : **Grp** $\longrightarrow$ **Set** *which maps a group to its underlying set while forgetting its mathematical structure and a group homomorphism to its underlying function of sets, is a forgetful functor.*

Example 3.2 *(**Product, coproduct, exponent**) The product* $\times$ *(respectively the coproduct* $+$*) defined in Subsection 2.3 (respectively in Subsection 2.4) give rises to a functor* $\times$: **Set** $\times$ **Set** $\longrightarrow$ **Set**, *from the product category (respectively the coproduct category) of* **Set** $\times$ **Set** *of* **Set** *with itself, to* **Set**. *The exponent (defined in Subsection 2.5) is also a functor* $\mathbf{Set}^{op} \times \mathbf{Set} \longrightarrow \mathbf{Set}$ *which involves a dual category for its first argument.*

In the following, we give three examples of functors:

3.1.1 Powersets

The *powerset* functor $\mathcal{P} : \mathbf{Set} \longrightarrow \mathbf{Set}$ maps any set X to its powerset i.e. the set of all subsets of X:

$$\mathcal{P}(X) = \{U \mid U \subseteq X\}$$

and any function $f : X \longrightarrow Y$ to the function:

$$\begin{array}{rccl} \mathcal{P}(f): & \mathcal{P}(X) & \longrightarrow & \mathcal{P}(Y) \\ & U \subseteq X & \mapsto & \{f(x) \mid x \in U\} = \{y \in Y \mid \exists x \in X, f(x) = y \ \wedge \ x \in U\} \end{array}$$

The powerset operation is indeed a functor because it preserves identities and composition:

$$\begin{array}{rcl} \mathcal{P}(\mathsf{id}_X)(U) & = & \{\mathsf{id}_X(x) \mid x \in U\} \\ & = & \{y \in X \mid \exists x \in X, \mathsf{id}_X(x) = y \ \wedge \ x \in U\} \\ & = & \{y \in X \mid \exists x \in X, x = y \ \wedge \ x \in U\} \\ & = & \{x \in X \mid x \in U\} \\ & = & U \end{array}$$

$$\begin{array}{rcl} (\mathcal{P}(g) \circ \mathcal{P}(f))(U) & = & \{z \in Z \mid \exists y \in Y, g(z) = z, \text{ and } \exists x \in X, f(x) = y \text{ and } x \in U\} \\ & = & \{z \in Z \mid \exists y \in Y, g(f(x)) = z \text{ and } x \in U\} \\ & = & \{z \in Z \mid \exists y \in Y, g \circ f(x) = z \text{ and } x \in U\} \\ & = & \mathcal{P}(g \circ f)(U) \end{array}$$

Finally, the *finite powerset* of a set X will be denoted by $\mathcal{P}_{\text{fin}}(X) = \{U \mid U \subseteq X \wedge \ U \text{ finite}\}$.

3.1.2 Free monoid

The *free monoid* functor $\mathsf{Mon} : \mathbf{Set} \longrightarrow \mathbf{Set}$ associates to any set X the set of all finite sequences over X (including the empty one):

$$\mathsf{Mon}(X) = X^* = \{\langle x_0, x_1, \ldots, x_n \rangle \mid \forall 0 \leq i \leq n, x_i \in X\}$$

and to any function $f : X \longrightarrow Y$ the function:

$$\begin{array}{rccl} \mathsf{Mon}(f) = f^*: & \mathsf{Mon}(X) & \to & \mathsf{Mon}(Y) \\ & \langle x_0, x_1, \ldots, x_n \rangle & \mapsto & \langle f(x_0), f(x_1), \ldots, f(x_n) \rangle \end{array}$$

3.1.3 Polynomial functors and Kripke polynomial functors

Polynomial functors are functors $\mathbf{Set} \longrightarrow \mathbf{Set}$ built up inductively from simple basic functors, using products, coproducts and exponents for forming new functors.

Definition 3.2 *(**Polynomial Functors**) The class of **polynomial functors** is inductively defined as the least collection of functors* $\mathbf{Set} \longrightarrow \mathbf{Set}$ *satisfying the following clauses:*

1. *The identity functor* $\mathsf{id} : \mathbf{Set} \longrightarrow \mathbf{Set}$ *is a polynomial functor;*
2. *For any set K, the constant functor* $K : \mathbf{Set} \longrightarrow \mathbf{Set}$ *is a polynomial functor. This functor maps every set X to K, and every function $f : X \longrightarrow Y$ to the identity* $\mathsf{id}_K : K \longrightarrow K$*;*
3. *If F_1 and F_2 are polynomial functors, then their product $F_1 \times F_2$ is also a polynomial functor. This functor*[3] *maps every set X to the product $F_1(X) \times F_2(X)$ and every function f to the product $F_1(f) \times F_2(f)$;*

3. See Sections 2.3, 2.4 and 2.5 for the definitions of product, coproduct and exponent respectively.

4. *If F_1 and F_2 are polynomial functors, then their coproduct $F_1 + F_2$ is also a polynomial functor. This functor maps every set X to the coproduct $F_1(X) + F_2(X)$ and every function f to the product $F_1(f) + F_2(f)$;*
5. *For any set E, if F is a polynomial functor, then the exponent F^E is also a polynomial functor. This functor maps every set X to the exponent $F(X)^E$ and every function $f : X \longrightarrow Y$ to the function $F(f)^E = F(f)^{\mathsf{id}_E}$ which maps $h : E \longrightarrow F(X)$ to $F(f) \circ h : E \longrightarrow F(Y)$.*

If we add both the power set functor and the free monoid functor, we obtain *Kripke polynomial functors*:

Definition 3.3 *(**Kripke polynomial functors**) The class of **Kripke polynomial functors** is the superset of polynomial functors obtained by the rules defined in Definition 3.2, with two additional rules:*

1. *If F is a Kripke polynomial functor, then the powerset $\mathcal{P}(F)$ is also[4] Kripke. This functor maps every set X to $\mathcal{P}(F(X))$, and every function f to $\mathcal{P}(F(f)(X))$;*
2. *If F is a Kripke polynomial functor, then the free monoid $\mathsf{Mon}(F)$ is also Kripke polynomial functor. This functor maps every set X to $\mathsf{Mon}(F(X))$, and every function f to $\mathsf{Mon}(F(f)(X))$.*

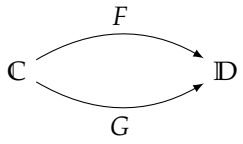
Occasionally, for technical reasons, we will need to restrict ourselves to the finite Kripke polynomial functors that are Kripke polynomial functors where all powersets $\mathcal{P}(\_)$ are finite powersets $\mathcal{P}_{\text{fin}}(\_)$.

3.1.4 The category of category

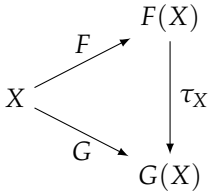
The category of category, noted **Cat**, has all small categories as objects and all functors between such categories as arrows. The composition of two functors $F : \mathbb{A} \longrightarrow \mathbb{B}$ and $G : \mathbb{B} \longrightarrow \mathbb{C}$ is $G \circ F : \mathbb{A} \longrightarrow \mathbb{C}$ defined for any object A of $\mathbb{A}$ by $G \circ F(A) = G(F(A))$ and for any arrow f of $\mathbb{A}$ by $G \circ F(f) = G(F(f))$.

3.2 Natural transformations

A natural transformation provides a way to transform one functor into another while respecting the structure of the categories involved. Given two parallel functors i.e. with the same domain and codomain:



For each object X in $\mathbb{C}$, we can associate two objects $F(X)$ and $G(X)$ in $\mathbb{D}$ that should be the source and the target of an arrow $\tau_X : F(X) \longrightarrow G(X)$ in $\mathbb{D}$:



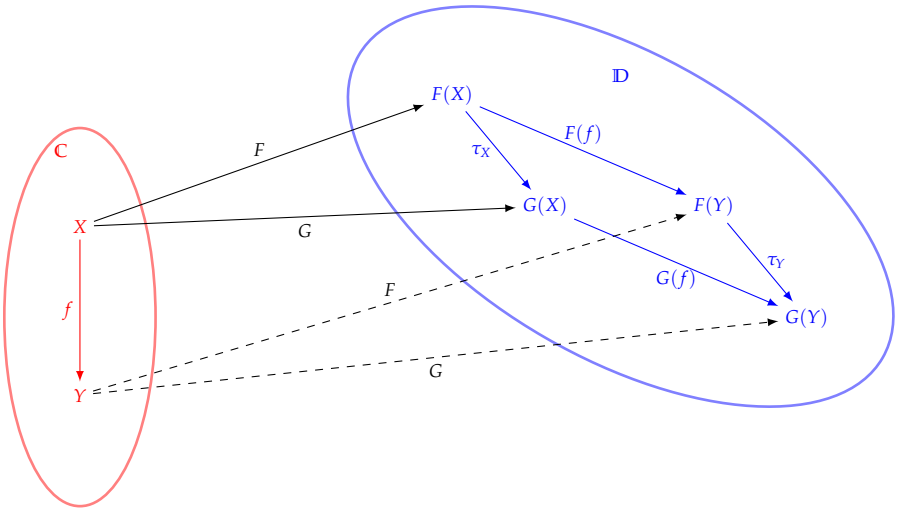
Thus, we have a collection of arrows $\tau_X : F(X) \longrightarrow G(X) \in \mathbb{D}$, indexed by objects $X \in \mathsf{Obj}(\mathbb{C})$. This collection is called a *natural transformation* from F to G. Hence, natural transformations define morphisms between functors.

4. See Subsections 3.1.1 and 3.1.2 for the definitions of powerset and monoid respectively.

Definition 3.4 *(**Natural transformation**) Let $\mathbb{C}, \mathbb{D}$ be two categories. Let $F, G : \mathbb{C} \longrightarrow \mathbb{D}$ be two parallel functors between $\mathbb{C}$ and $\mathbb{D}$. A **natural transformation** from F to G is a $\mathbb{C}$-object indexed family of morphisms in $\mathbb{D}$ $\tau = (\tau_X)_{X \in \mathsf{obj}(\mathbb{C})} : F \Longrightarrow G$ such that:*

$$\tau_Y \circ F(f) = G(f) \circ \tau_X$$

for every arrow $f : X \longrightarrow Y$ in $\mathbb{C}$.

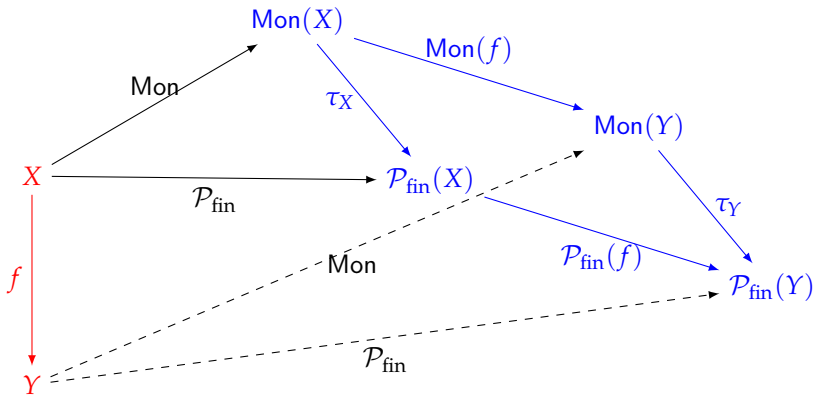


Note: τ_X is called the *component* at X of the natural transformation τ.

Example 3.3 *Consider the finite powerset functor $\mathcal{P}_{\text{fin}}$: **Set** $\longrightarrow$ **Set** and the free monoid functor* Mon : **Set** $\longrightarrow$ **Set**. *There is a natural transformation τ : Mon $\Longrightarrow \mathcal{P}_{\text{fin}}$ that associates to every set X a function τ_X : Mon$(X) \longrightarrow \mathcal{P}_{\text{fin}}(X)$ which is defined for every sequence $\langle x_0, x_1, \ldots, x_n \rangle \in$ Mon(X) by:*

$$\tau_X(\langle x_0, x_1, \ldots, x_n \rangle) = \{x_0, x_1, \ldots, x_n\}$$

This operation is indeed natural because the following diagram commutes for any function $f : X \longrightarrow Y$:



That is to say, for each sequence $\langle x_0, x_1, \ldots, x_n \rangle \in$ Mon(X), one has:

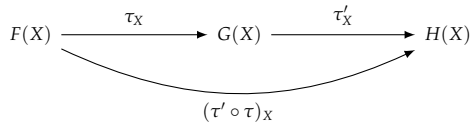
$$(\tau_Y \circ \mathsf{Mon}(f))(\langle x_0, x_1, \ldots, x_n \rangle) = (\mathcal{P}_{\text{fin}}(f) \circ \tau_X)(\langle x_0, x_1, \ldots, x_n \rangle)$$

$$
\begin{aligned}
(\tau_Y \circ \mathsf{Mon}(f))(\langle x_0, \ldots, x_n \rangle) &= \tau_Y(\mathsf{Mon}(f)(\langle x_0, \ldots, x_n \rangle)) \\
&= \tau_Y(\langle f(x_0), \ldots, f(x_n) \rangle) \\
&= \{f(x_0), \ldots, f(x_n)\} \\
(\mathcal{P}_{\text{fin}}(f) \circ \tau_X)(\langle x_0, \ldots, x_n \rangle) &= \mathcal{P}_{\text{fin}}(f)(\tau_X(\langle x_0, \ldots, x_n \rangle)) \\
&= \mathcal{P}_{\text{fin}}(f)(\{x_0, \ldots, x_n\}) \\
&= \{f(x_0), \ldots, f(x_n)\}
\end{aligned}
$$

3.3 Heterogeneous Compositions

3.3.1 Functor categories

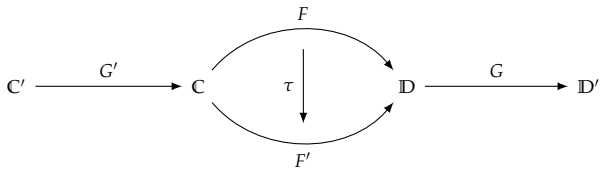
Given three parallel functors: F, G and H from $\mathbb{C}$ to $\mathbb{D}$ and two natural transformations between them: $\tau : F \Longrightarrow G$ and $\tau' : G \Longrightarrow H$ such that for each $X \in \mathsf{Obj}(\mathbb{C})$, one has $(\tau \circ \tau')_X = \tau_X \circ \tau'_X$:



We can form a new category called *functors category* and noted $\mathbb{C}^{\mathbb{D}}$ having as objects all functors from $\mathbb{C}$ to $\mathbb{D}$ and as arrows the natural transformations between those functors. It is obvious that the composition of natural transformations is a natural transformation, that this composition is associative, and that any identity natural transformation acts as a unit.

3.3.2 Heterogeneous compositions

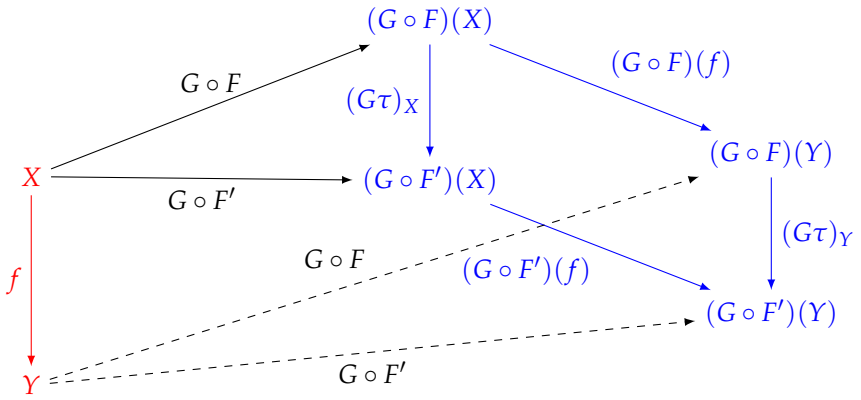
Natural transformations may be composed with functors to get new natural transformations. Let $\mathbb{C}, \mathbb{D}, \mathbb{C}'$ and $\mathbb{D}'$ be categories, let $F, F' : \mathbb{C} \longrightarrow \mathbb{D}$, $G : \mathbb{C}' \longrightarrow \mathbb{C}$ and $G' : \mathbb{D} \longrightarrow \mathbb{D}'$ be functors and let $\tau : F \Longrightarrow F'$ be a natural transformation as the following diagram shows:



There are two ways to compose natural transformations and functors:

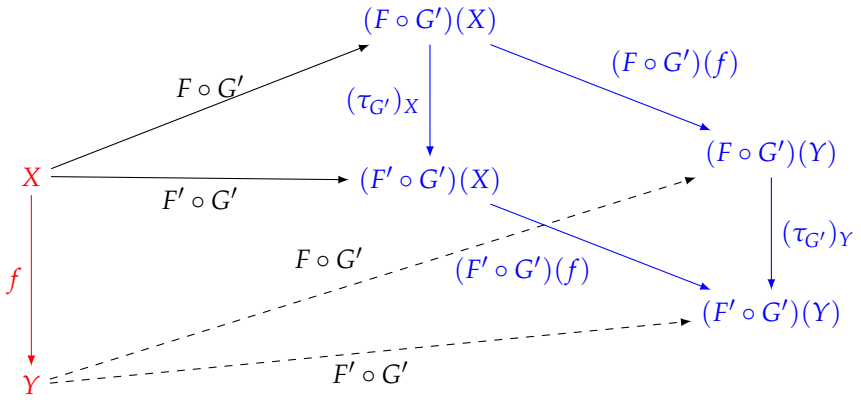
1. Composing G with τ leads to a new natural transformation from $G \circ F$ to $G \circ F'$ that is usually noted $G\tau$ (not $G \circ \tau$) in order to distinguish it from the usual composition. This is a composition of heterogeneous objects. It is not hard to see that $G\tau$ makes the following

diagram commute:



Let us note here that the functor G impacts only arrows. That is to say that the composition is only made between the "arrow function" of F and the natural transformation τ.

2. Composing τ with G' leads to a new natural transformation from $F \circ G'$ to $F' \circ G'$ making the following diagram commutes:



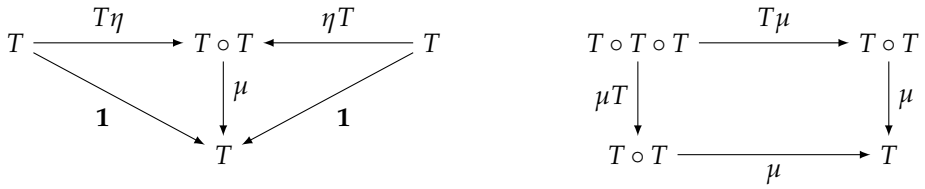
Let us note here that the functor G' impacts only on objects. That is to say the composition is made between the natural transformation τ and the "object function" of G'. For this reason, we write $\tau_{G'}$ instead of $\tau G'$ (i.e. τ is indexed by G').

4 Monads in category theory

Monads are a categorical tool that was first introduced in the 80′*s* by *Moggi* in order to develop a categorical semantics of computations for programming languages [16, 15]. Later, in the 90′*s*, monads received much attention from the community of functional programming languages because they have shown their suitability for sequentially combining computations into more complex computations [32, 14]. They were then used to introduce aspects of imperative programming languages such as input/output (I/O) operations, state updating, exceptions, nondeterminism, etc., in functional programming languages. More precisely, they allow the explicit addition, to functional programming languages, of concepts which they lack such as side effects. Consequently, monads can then sequentially build severals kinds of computation effects such as non-determinism, partiality, exception, etc., flexibly.

4.1 Definition

Definition 4.1 *(**Monad**) Let $\mathbb{C}$ be a category. A **monad in** $\mathbb{C}$ consists of an endofunctor $T : \mathbb{C} \longrightarrow \mathbb{C}$ equipped with two natural transformations $\eta : \mathbf{1} \Longrightarrow T$ and $\mu : T \circ T \Longrightarrow T$ (with $\mathbf{1} : \mathbb{C} \longrightarrow \mathbb{C}$ is the identity functor) which satisfy the conditions $\mu \circ T\eta = \mu \circ \eta T = \mathbf{1}$ and $\mu \circ T\mu = \mu \circ \mu T$:*



η is called the *unit* of the monad. Its components map objects in $\mathbb{C}$ to their naturally structured counterpart. μ is the *multiplication* of the monad. Its components map objects with two levels of structure to objects with only one level of structure. The first condition states that a doubly structured object $\eta_{T(X)}(x)$ built by η from a structured object x, is flattened by μ to the same structured object as a structured object $T(\eta_X)(x)$ made of structured objects built by η. The second condition states that when flattening two levels of structure, we get the same result by flattening the outer structure first (with $\mu_{T(X)}$) or the inner structure first (with $T(\mu_X)$).
In the following, we will often simply write T^2 for $T \circ T$ and T^3 for $T \circ T \circ T$.

4.2 A working example

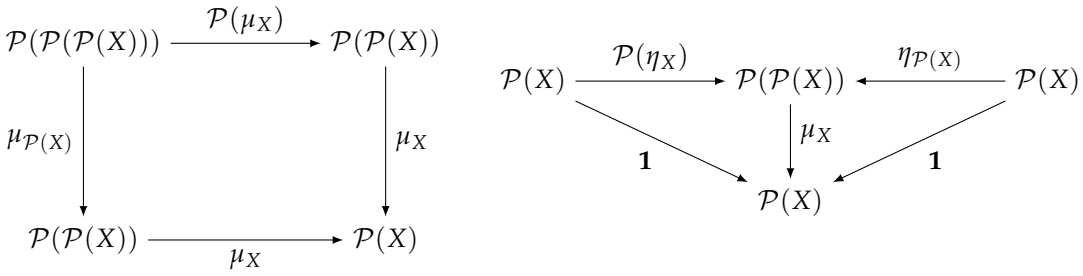
Let us consider a monad built on the powerset functor $\mathcal{P}$: **Set** $\rightarrow$ **Set**. For every set X, the component $\eta_X : X \longrightarrow \mathcal{P}(X)$ of the unit of this monad has to build a set of elements from an element of X. We can choose $\eta_X : x \mapsto \{x\}$ that maps every element $x \in X$ to the singleton $\{x\}$. On the other hand, for every set X, the component $\mu_X : \mathcal{P}(\mathcal{P}(X)) \longrightarrow \mathcal{P}(X)$ of the multiplication of this monad has to flatten a set of sets of elements of X into a set of elements. In order to define μ, we need to understand the nature of $\mathcal{P}^2(X)$. Indeed, $\mathcal{P}^2$ is the set of all subsets not formed by elements of X, but by elements of $\mathcal{P}(X)$. There is a set with two levels of nesting. For instance, x, y and z are elements of X, the sets $\{x, y\}, \{z\}$ and $\{x\}$ are elements of $\mathcal{P}(X)$, and the sets $\{\{x, y\}, \{z\}\}$ and $\{\{x\}\}$ are elements of $\mathcal{P}^2(X)$. The multiplication of the monad simply consists in transforming a set of subsets of X into a set of X i.e. it flattens all subsets of X into one set by omitting a nesting level. For instance:

$$\mu_X(\{\{x, y\}, \{z\}\}) = \{x, y, z\}$$

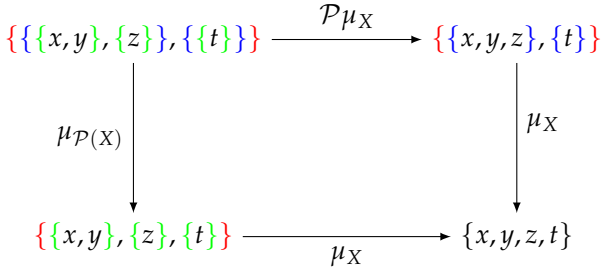
Formally, μ is defined by:

$$\begin{array}{rccc} \mu : & \mathcal{P}^2(X) & \rightarrow & \mathcal{P}(X) \\ & U & \mapsto & \bigcup\limits_{u \in U} (u) \end{array}$$

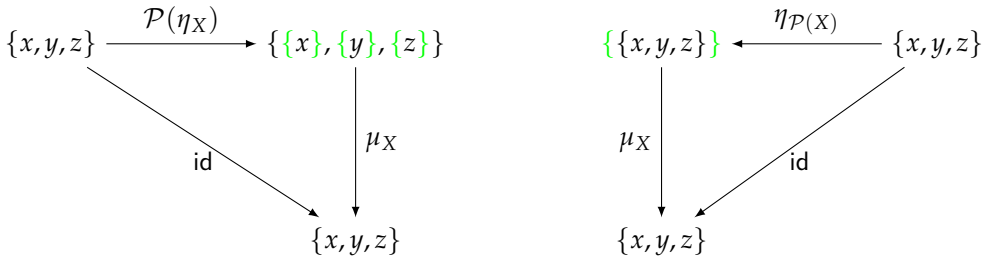
To check that η and μ are natural transformations, it is enough to show that the following diagrams commute.



The diagram depicted on the left side can be read as follows: given a set of three levels of parentheses, remove the inner parentheses, then the intermediate parentheses (that become inners after the first operation), or remove the intermediate parentheses, then the inner parentheses will get the same result. For example, for a set $X = \{x, y, z, t\}$, one has:



Similarly, the diagram depicted on the right side can be read as follows: given a set of elements $\{x_0, \ldots, x_n\}$ of a set X, first return the singleton containing the set $\{x_0, \ldots, x_n\}$ (i.e. add a parenthesis from the outside to $\{x_0, \ldots, x_n\}$) and then reduce the obtained singleton $\{\{x_0, \ldots, x_n\}\}$ by μ_X, or first return the set $\{\{x_0\}, \ldots, \{x_n\}\}$ obtained after applying η_X to every element of $\{x_0, \ldots, x_n\}$ and then reduce the obtained result $\{\{x_0\}, \ldots, \{x_n\}\}$ by μ_X will lead to the same result which is $\{x_0, \ldots, x_n\}$. For example, for a set $X = \{x, y, z\}$, one has:



Let us now verify that the above diagrams commute: given that for any morphism $f : X \longrightarrow Y$, $\mathcal{P}(f) : \mathcal{P}(X) \longrightarrow \mathcal{P}(Y)$ is the morphism which maps each part of X to its image by f. We must check that:

$$\mu \circ \eta\mathcal{P} = \mu \circ \mathcal{P}\eta = \mathbf{1} \tag{II.4}$$

$$\mu \circ \mu\mathcal{P} = \mu \circ \mathcal{P}\mu \tag{II.5}$$

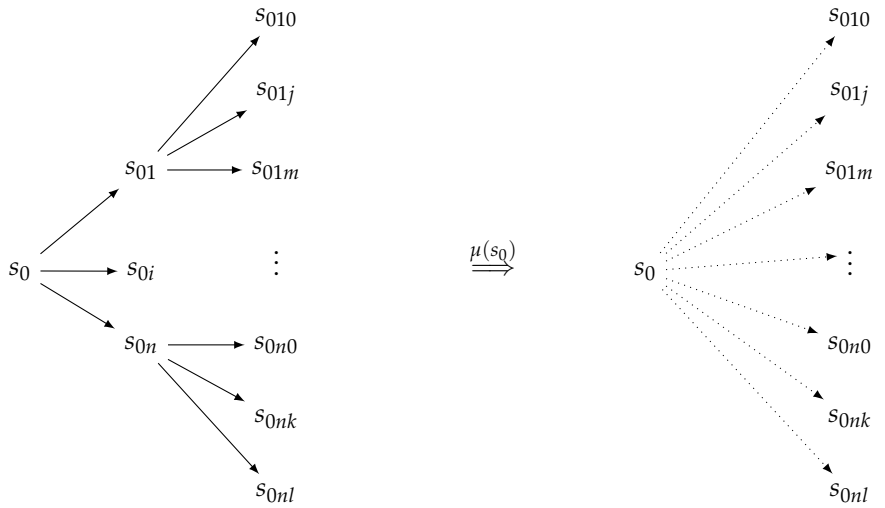
For Equation II.4, we have:

$$\begin{array}{lccccc}
\mu_X \circ \eta_{\mathcal{P}(X)}: & \mathcal{P}(X) & \xrightarrow{\eta_{\mathcal{P}(X)}} & \mathcal{P}(\mathcal{P}(X)) & \xrightarrow{\mu_X} & \mathcal{P}(X) \\
& \{x_1, \ldots, x_i \ldots\} & \longmapsto & \{\{x_1, \ldots, x_i \ldots\}\} & \longmapsto & \{x_1, \ldots, x_i \ldots\} \\
\mu_X \circ \mathcal{P}(\eta_X): & \mathcal{P}(X) & \xrightarrow{\mathcal{P}(\eta_X)} & \mathcal{P}(\mathcal{P}(X)) & \xrightarrow{\mu_X} & \mathcal{P}(X) \\
& \{x_1, \ldots, x_i \ldots\} & \longmapsto & \{\{x_1\}, \ldots, \{x_i\}, \ldots\} & \longmapsto & \{x_1, \ldots x_i, \ldots\}
\end{array}$$

For Equation II.5, we have:

$$\begin{array}{lccccc}
\mu_X \circ \mu_{\mathcal{P}(X)}: & \mathcal{P}^3(X) & \xrightarrow{\mu_{\mathcal{P}(X)}} & \mathcal{P}^2(X) & \xrightarrow{\mu_X} & \mathcal{P}(X) \\
& \{\{\{x_{11}\},\ldots\{x_{1j}\},\ldots\}, & & \{\{x_{11}\},\ldots\{x_{1j}\},\ldots, & & \\
& \{\{x_{21}\},\ldots\{x_{2j}\},\ldots\}, & \longmapsto & \{x_{21}\},\ldots\{x_{2j}\},\ldots & \longmapsto & \{x_{1i},x_{1j},\ldots,x_{2i},x_{2j},\ldots\} \\
& \ldots \quad \ldots \quad \} & & \ldots \quad \ldots \quad \} & & \\
\mu_X \circ \mathcal{P}(\mu_X): & \mathcal{P}^3(X) & \xrightarrow{\mathcal{P}(\mu_X)} & \mathcal{P}^2(X) & \xrightarrow{\mu_X} & \mathcal{P}(X) \\
& \{\{\{x_{11}\},\ldots\{x_{1j}\},\ldots\}, & & \{\{x_{11},\ldots x_{1j},\ldots\} & & \\
& \{\{x_{21}\},\ldots\{x_{2j}\},\ldots\}, & \longmapsto & \{x_{21},\ldots x_{2j},\ldots\} & \longmapsto & \{x_{1i},x_{1j},\ldots x_{2i},x_{2j},\ldots\} \\
& \ldots \quad \ldots \quad \} & & \ldots \quad \ldots \quad \} & &
\end{array}$$

The powerset monad is usually used to model non-deterministic state machines by replacing the target state of a transition by a set of possible target states. Intuitively, the unit η allows us to add a non-deterministic behaviour to every state as a trivial non-determinism (with only one possibility to choose). The multiplication μ allows us to obtain the successors of the successor states of each state, abstracting away internal states:



4.3 More examples

We describe here a few usual monads.

4.3.1 Partial

The *partial* monad (or *maybe*, or also *deadlock*) is defined by the triplet $(\mathsf{id} + \mathbf{1}, \eta, \mu)$ with:

– $\mathsf{id}+\mathbf{1} : \mathbf{Set} \longrightarrow \mathbf{Set}$ is the functor that maps a set X to $(\mathsf{id}+1)(X) = X_\perp = X \cup \{\perp\}$ and that sends a function $f : X \longrightarrow Y$ to a function :$(\mathsf{id}+1)(f) : X \cup \{\perp\} \longrightarrow Y \cup \{\perp\}$ given by:

$$\perp \mapsto \perp \text{ and } x \mapsto f(x)$$

where $\mathbf{1}$ is the singleton set whose only element is $\perp$.

– For every X, the unit $\eta_X : X \longrightarrow X_\perp$ is the canonical inclusion that maps every element $x \in X$ to itself;

– For every X, the multiplication $\mu_X : X_\perp \longrightarrow X_\perp$ is the application defined as follows:

$$\begin{array}{rccc} \mu_X : & X_\perp & \to & X_\perp \\ & \perp & \mapsto & \perp \\ & x & \mapsto & x \end{array}$$

The monad partial is usually used to model objects that evolve and can be interrupted or disappear at any time. For example, systems that evolve by moving to successor states, and for which we cannot guarantee whether their execution will be normally terminated. There is a deadlock state in which the system runs in unsafe state. Then, the state space X is extended to $X_\perp$ by adding the special symbol $\perp$ to signal explicitly deadlock states.

4.3.2 Ordered nondeterminism

The *ordered nondeterminism* (or *sequence*) is defined by the triplet $(\mathsf{id}^*, \eta, \mu)$ with:

– $\mathsf{id}^* : \mathbf{Set} \longrightarrow \mathbf{Set}$ is the functor that maps a set X to $\mathsf{id}^*(X)$ which is the set of all lists that can be recursively formed with the elements of X and the usual list operations. As well, the functor id^* associates to a function $f : X \longrightarrow Y$ a function:

$$\begin{array}{rccc} \mathsf{id}^*(f) : & \mathsf{id}^*(X) & \to & \mathsf{id}^*(Y) \\ & [x_0, \ldots, x_n] & \mapsto & [f(x_0), \ldots, f(x_n)] \end{array}$$

– For every X, the unit is the singleton list constructor. It is an application $\eta_X : X \longrightarrow \mathsf{id}^*(X)$ mapping every element $x \in X$ to the list $[x]$;

– For every$X$, the multiplication $\mu_X : \mathsf{id}^*(\mathsf{id}^*(X)) \longrightarrow \mathsf{id}^*(X)$ is the application that flattens out a list of lists of elements in list of elements, by removing all inner "square brackets". For example, it transforms $[[x, y, z], [t]]$ into $[x, y, z, t]$. Formally, this application is defined by:

$$\begin{array}{rccc} \mu_X : & \mathsf{id}^*(\mathsf{id}^*(X)) & \to & \mathsf{id}^*(X) \\ & [[x_{11}, \ldots, x_{1n_1}], \ldots, [x_{m1}, \ldots, x_{mn_m}]] & \mapsto & [x_{11}, \ldots, x_{1n_1}, \ldots, x_{m1}, \ldots, x_{mn_m}] \end{array}$$

The ordered nondeterminism monad is usually used in cases where there is a set of possibilities ordered from a certain perspective, and that requires an exhaustive exploration of all these possibilities. For example, a system that has possible input signals at any given time, and only one of them has to be taken. Hence, this entails an ordered view of nondeterminism. A possible choice may be then "all inputs are possible", but their probability decreases (or increases) depending on their lengths.

4.3.3 Exception

The *exception* monad is defined by the triplet $(\mathsf{id} + \mathsf{E}, \eta, \mu)$ with:

- $\mathsf{id} + \mathsf{E} : \mathbf{Set} \longrightarrow \mathbf{Set}$ is the functor that maps a set X to $(\mathsf{id} + \mathsf{E})(X) = X \cup \mathsf{E}$ and a function $f : X \longrightarrow Y$ to $(\mathsf{id} + \mathsf{E})(f) = f + \mathsf{id}_{\mathsf{E}}$;
- For every X, the unit $\eta_X : X \longrightarrow X \cup \mathsf{E}$ is the injective application maping every $x \in X$ to itself;
- For every X, the multiplication $\mu_X : (\mathsf{id} + \mathsf{E})((\mathsf{id} + \mathsf{E})(X)) \longrightarrow (\mathsf{id} + \mathsf{E})(X)$ is the application defined as follows:

$$\begin{array}{rlll} \mu_X : & (\mathsf{id} + \mathsf{E})((\mathsf{id} + \mathsf{E})(X)) & \rightarrow & (\mathsf{id} + \mathsf{E})(X) \\ & e & \mapsto & e \\ & x & \mapsto & x \end{array}$$

The exception monad is usually used to handle exceptions in many programming languages. Programs may indeed terminate "abruptly" because of an exception. Then, from this point of view, monads allow us to model computations that either normally succeed, moving to a successor state from a given state, or fail raising an exception $e \in \mathsf{E}$.

Note first that the partial monad can be seen as a particular instance of the exception monad. It is enough to instantiate E with $\mathbf{1}$. Second, non-termination (deadlock state) is basically different from exceptions. That is: once a computation is blocked, it blocks forever. There is no way to get out of the deadlock state. However, using a suitable exception handler, normal termination can be restored when a system "exceptionally" terminates.

4.4 Category of Kleisli

There is an alternative description of monads to represent computations: *Kleisli triple*. Formally, *Kleisli triples* are defined as follows:

Definition 4.2 *(**Kleisli triple**) A **Kleisli triple** over a category $\mathbb{C}$ is a triple $(T, \eta, \_^*)$ where $T : \mathsf{Obj}(\mathbb{C}) \longrightarrow \mathsf{Obj}(\mathbb{C})$ is an object mapping, $\eta_X : X \longrightarrow TX$ is a $\mathsf{Obj}(\mathbb{C})$-indexed mapping in $X \in \mathsf{Obj}(\mathbb{C})$, and $\_^*$ is an operator that assigns to each function $f : X \longrightarrow TY$ a function $f^* : TX \longrightarrow TY$ such that:*

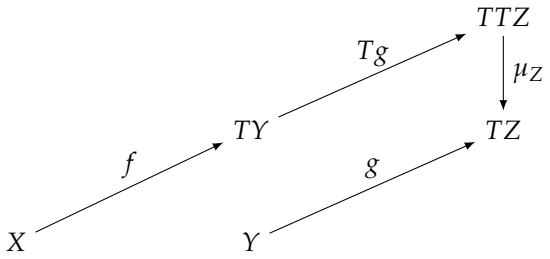
- $\eta_X^* = \mathsf{id}_{TX}$;
- $f^* \circ \eta_X = f$ *for* $f : X \longrightarrow TY$;
- $g^* \circ f^* = (g^* \circ f)^*$ *for* $f : X \longrightarrow TY$ *and* $g : Y \longrightarrow TZ$.

From any *Kleisli triple* $(T, \eta, \_^*)$ over $\mathbb{C}$, we can define a new category called *Kleisli category* and noted $\mathsf{Kl}_T(\mathbb{C})$ [57].

Definition 4.3 *(**Kleisli category**) Given a Kleisli triple $(T, \eta, \_^*)$ over $\mathbb{C}$. The **category of Kleisli** $\mathsf{Kl}_T(\mathbb{C})$ is defined as follows:*

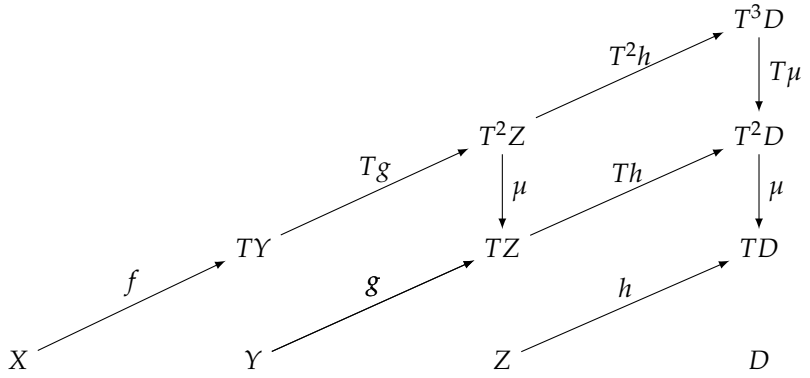
- *The objects of $\mathsf{Kl}_T(\mathbb{C})$ are those of $\mathbb{C}$;*
- *The morphisms $X \longrightarrow Y$ from X to Y in $\mathsf{Kl}_T(\mathbb{C})$ are the morphisms $X \longrightarrow TY$ from X to TY in $\mathbb{C}$;*
- *For every object X, the identity $\mathsf{id}_X : X \longrightarrow X$ in $\mathsf{Kl}_T(\mathbb{C})$ is the morphism $\eta_X : X \longrightarrow TX$ in $\mathbb{C}$;*
- *For every pair of morphisms $f : X \longrightarrow Y$ and $g : Y \longrightarrow Z$ in $\mathsf{Kl}_T(\mathbb{C})$, the composite of $g \circ f : X \longrightarrow Z$ in $\mathsf{Kl}_T(\mathbb{C})$ is defined in $\mathbb{C}$ as follows:*

$$g \circ f : X \xrightarrow{f} TY \xrightarrow{Tg} TTZ \xrightarrow{\mu} TZ$$



It is not hard to see that the data of this definition form a category. Indeed, the following properties are verified:

- Compositionality of arrows: for each pair of arrows $f : X \longrightarrow Y$ and $g : Y \longrightarrow Z$ in $\mathsf{Kl}_T(\mathbb{C})$, there is a composite arrow $g \circ f : X \longrightarrow Z$ in $\mathsf{Kl}_T(\mathbb{C})$. This property is ensured by the operations of the monad T as shown in the last point of the above definition.
- Associativity of the composition: if $X \xrightarrow{f} Y \xrightarrow{g} Z \xrightarrow{h} D$, then $(h \circ g) \circ f = h \circ (g \circ f)$. To verify this property, it is enough to consider the following diagram in $\mathbb{C}$, and show that the two morphisms from X to TD coincide.



Example 4.1 *(**Rel** **Category**) Consider the* **Rel** *category of sets and relations. Its objects are ordinary sets, and its arrows $X \longrightarrow Y$ are binary relations $R \subseteq X \times Y$. The composition of two relations $R_1 : X \longrightarrow Y$ and $R_2 : Y \longrightarrow Z$ is given by:*

$$R_2 \circ R_1 = \{(x, z) \in X \times Z \mid \exists y \in Y \text{ such that } (x, y) \in R_1 \text{ and } (y, z) \in R_2\}$$

The **Rel** *category can be obtained from the* **Set** *category as the Kleisli category $\mathsf{Kl}_T(\mathbb{C})$ for the monad whose functor corresponds to powerset. Then, instantiating the monad T with the powerset monad $\mathcal{P}$, the kleisli category $\mathsf{Kl}_{\mathcal{P}}(\mathbf{Set})$ becomes equivalent to the* **Rel** *category. Indeed, a relation $R \subseteq X \times Y$ can be seen as a function from X to $\mathcal{P}(Y)$. Then, given a function $X \longrightarrow Y$ in* **Set**, *its corresponding in $\mathsf{Kl}_{\mathcal{P}}(\mathbf{Set})$ is the relation $R_f = \{(x, y) \mid y = f(x)\}$. This lifting from* **Set** *to $\mathsf{Kl}_{\mathcal{P}}(\mathbf{Set})$ is therefore done via the functor $\mathsf{Graph} : \mathbf{Set} \longrightarrow \mathsf{Kl}_{\mathcal{P}}(\mathbf{Set})$ which maps a set X to itself $\mathsf{Graph}(X) = X$, and a function $f : X \longrightarrow Y$ to its graph relation $\mathsf{Graph}(f) = \{(x, y) \mid y = f(x)\} \subseteq X \times Y$.*

E.Manes [59] has shown the equivalence between monads and *Kleisli triples*. Given a *Kleisli triple* $(T, \eta, \_^*)$ the corresponding monad is (T, η, μ) where T is the endofunctor over $\mathbb{C}$ that extends the function T and maps any function $f : X \longrightarrow Y$ to $(\eta_Y \circ f)^*$, and the multiplication μ_X acts

as id^*_{TX} for every set X. Conversely, given a monad (T, η, μ), the corresponding *Kleisli triple* is $(T, \eta, \_^*)$ where T is the restriction of the endofunctor T to objects, and $f^* = \mu_Y \circ (Tf)$ for every function $f : X \longrightarrow TY$.

Chapter III

Coalgebras

1 Coalgebra definition . . . 50
1.1 Streams . . . 50
1.2 Mealy Machines . . . 51
1.3 Labeled Transition Systems (LTS) . . . 51
1.4 Input-Output Labeled Transition Systems (IOLTS) . . . 52
2 Morphisms . . . 52
3 Bisimulation . . . 54
3.1 Stream . . . 55
3.2 Mealy machines . . . 55
3.3 Labeled transition systems . . . 55
4 Final coalgebras . . . 57
4.1 Streams . . . 58
4.2 Mealy machines . . . 60
4.3 Labeled transition systems . . . 61
4.4 More examples . . . 63
5 Co-induction . . . 64
5.1 Proof by bisimulation . . . 66

Coalgebra theory was first introduced in computer science during the $80's$ by *Arbib* and *Ernest* in [60, 61] as an abstract formalism for describing state-based systems such as automata (in various forms) and transition systems. Later, coalgebra theory emerged, and rapidly became a powerful tool in different areas of computer science. Indeed, during the $1990's$, a step in the formalization of concepts of classes and objects in object-oriented programming was achieved thanks to this theory [7, 8, 62, 63]. Since then, coalgebras have shown that they are suitable mathematical structures to model and unify state-based dynamic systems. They provide a theoretical framework offering an excellent modeling tool whose effectiveness can describe a large family of state-based systems and prove their properties. In fact, the characteristics of modern systems viewed not from the point of view of how they are built, but of the results they produce, are hardly definable (or even simply not definable) by standard formalisms such as first order or equational logic. From this point of view, the theory of coalgebras can be seen as the dual of that of algebras. Then, algebras are used to build objects, by constructors, that are considered different if differently constructed. On the contrary, coalgebras are used to observe (or decompose) objects, by observers (or destructors), that are considered different when they can be distinguished by observation. So the difference between the two approaches is intuitively

expressed as follows: one shows the construction part of a system, the other shows the observation part of it. For example, given a set E, in the algebraic approach, finite lists over E can be inductively constructed using the two operations: $\mathsf{nil} : 1 \longrightarrow E^*$ and $\mathsf{cons} : E \times E^* \longrightarrow E^*$. The nil operation generates an empty list from nothing and the cons operation adds an element to the list. However, the infinite lists over E cannot be built in this way. One can only observe their elements. Hence, an infinite list can be thought of as a black box with a set of internal states S and two operations $\mathsf{obs} : S \longrightarrow E$ and $\mathsf{next} : S \longrightarrow S$ that associate respectively to each state $s \in S$ an observation and its successor state.

1 Coalgebra definition

Coalgebras can be seen as an abstraction of dynamical and reactive systems, behaviours are well described by transition functions (i.e. automata) of all kinds. A coalgebra consists [1] of a set S of states equipped with a transition function $\alpha : S \longrightarrow FS$ where $F : \mathbf{Set} \longrightarrow \mathbf{Set}$ is an endofunctor on the category of sets, defining the signature of the coalgebra. Hence, α provides the set of states S with some structures. Unlike algebraic operations that enable us to recursively build complex objects from basic objects given by signatures, coalgebra operations are a means to observe system states. More formally, we have:

Definition 1.1 *(**Coalgebra**) Let $F : \mathbf{Set} \longrightarrow \mathbf{Set}$ be a functor called* signature functor. *A **coalgebra** for F, or F-coalgebra is any couple (S, α) where:*

- *S is a set whose elements are called* states*;*
- *$\alpha : S \longrightarrow FS$ is a mapping called* transition function.

In the following, we give some classical examples of formalism semantics which can be defined in terms of coalgebras.

1.1 Streams

Streams are used to model continuous input, behaviour of state-based systems, finite or infinite sequences, etc. They are especially useful in modeling behaviour of dynamical systems. As an example, consider a *deterministic transition system with output* [64] $\mathcal{S}$ that behaves as follows: in state s, it either goes into the next state s' of s, producing an output o as the "observable effect" of the state transition, or it fails and then, its execution is terminated. Such a system is usually considered as a black box which can produce outputs, by moving from one state to another. The behaviour of such a system is called *stream automata* and consists of a set of internal states S with two operations acting on the state space S:

$$\mathsf{obs} : S \longrightarrow \mathsf{Out} \text{ and } \mathsf{next} : S \longrightarrow S$$

The operation obs associates to every state $s \in S$ the corresponding output $o \in \mathsf{Out}$ while the operation $\mathsf{next} : S \to S$ maps every state $s \in S$ to a successor state $s' \in S$. We distinguish two cases:

- If the computation is infinite (the system is running forever), we consider this kind of systems as coalgebras:
$$(S, \langle \mathsf{obs}, \mathsf{next} \rangle : S \to A \times S)$$
of the functor F defined by $FX = A \times X$

1. Note that we restrict ourselves to the category of sets. Hence, we work with coalgebras for an endofunctor F on the category of sets and functions.

– If the computation is finite (the execution is terminated), we consider this kind of systems as coalgebras:

$$(S, \langle \mathsf{obs}, \mathsf{next} \rangle : S \to A \times (S \cup \{\bot\}))$$

of the functor F defined by $FX = A \times (X + \mathbf{1})$.

1.2 Mealy Machines

Mealy machines can be seen as stream automata, which also accept environment input. A Mealy automata is indeed a deterministic automaton with inputs and outputs in which output values are determined both by the current state and the input values [34, 33, 65]. It is usually defined as consisting of a set S of states, a set In of input labels, a set Out of output labels, an output function $\mathsf{obs} : S \times \mathsf{In} \longrightarrow \mathsf{Out}$ associating an output to every state s depending on its current input, and finally a transition function $\mathsf{next} : S \times \mathsf{In} \longrightarrow S$ mapping every state to its successor state. Note the typical feature of state-based systems is that one can observe the state space by means of functions with the state space as domain. In fact, Mealy automata are naturally in coalgebraic setting because the state space S can be considered as a black box and the functions obs and next can be combined using the cartesian product as a single function: $\langle \mathsf{obs}, \mathsf{next} \rangle : S \times \mathsf{In} \longrightarrow \mathsf{Out} \times S$ which can be also written, using the technique of currying, as a single function with domain S:

$$\langle \mathsf{obs}, \mathsf{next} \rangle : S \longrightarrow (\mathsf{Out} \times S)^{\mathsf{In}}$$

Thus, Mealy automata are coalgebras:

$$(S, \langle \mathsf{obs}, \mathsf{next} \rangle : S \to (\mathsf{Out} \times S)^{\mathsf{In}})$$

of the functor F defined by $FX = (\mathsf{Out} \times X)^{\mathsf{In}}$.

1.3 Labeled Transition Systems (LTS)

A labeled transition system (LTS) is simply an automaton labeled by actions, and which has no final state [66, 36]. Formally, it is a tuple (S, A, R) where S is a set of states, $A = \sigma \cup \{\tau\}$ is a set of observable actions Σ and unobservable[2] actions $\{\tau\}$ and R is the transition relation between states. The relation R can be replaced by a function. Such a binary relation $R \subseteq X \times Y$ whose domain is X and codomain is Y can indeed be commonly seen as a part of the cartesian product $X \times Y$. The relation R can be then written as a function sending every element of X to a subset of the powerset $\mathcal{P}(A \times Y)$ of $A \times Y$. Thus, labeled transition systems are coalgebras:

$$(S, \alpha : S \to \mathcal{P}(A \times S))$$

of the functor F defined by $FX = \mathcal{P}(A \times S)$.

For technical reasons, another class of labeled transition systems called *finitely branching transition systems* has been defined. Every finitely branching transition system is bounded. That is for all state $s \in S$, the set

$$\{(a, s') \mid s \xrightarrow{a} s'\} \text{ is finite}$$

Such systems can be identified with coalgebras:

$$(S, \alpha : S \longrightarrow \mathcal{P}_{\text{fin}}(A \times S))$$

where $\mathcal{P}_{\text{fin}}$ is the finite powerset functor.

2. τ is called *internal action*.

1.4 Input-Output Labeled Transition Systems (IOLTS)

Many variants of *LTS*, that make distinction between input and output actions, have been introduced over the years 1990 in order to respond to technical testing requirements that we will address later in Chapter VII. For instance, *Input-Output Automata* (*IOA*) [37], *Input-Output State Machines* (*IOSM*) [38], *Input-Output Transition Systems* (*IOTS*) [27, 45], *Input-Output Labeled Transition Systems* (*IOLTS*) [67]. These models are very similar theoretically. We then consider modeling coalgebraically the *IOLTS* model since it is the most widely used for the purpose of testing. Formally, an *IOLTS* is a variant of *LTS*, where the set of observable actions Σ is partitioned into input and output actions i.e. $\Sigma = \Sigma^! \cup \Sigma^? \cup \{\tau\}$. Classically, in order to distinguish the input actions from the output actions, input actions (resp. output actions) are marked with the symbol ? (respectively the symbol !). Similarly to *LTSs*, *IOLTSs* are coalgebras

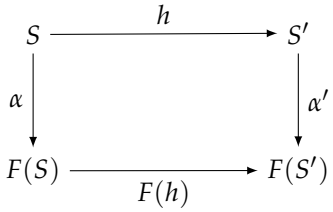
$$(S, \alpha : S \longrightarrow \mathcal{P}((\Sigma^? \cup \Sigma^! \cup \{\tau\}) \times S))$$

of the functor F defined by $FX = \mathcal{P}((\Sigma^? \cup \Sigma^! \cup \{\tau\}) \times S)$.

2 Morphisms

The easiest way to describe the relation between the different coalgebras of a given functor is via morphisms. A *morphism* from one coalgebra to another is an arrow between their state sets which preserves coalgebraic structures. If (S, α) and $(S'\alpha')$ are two coalgebras of a functor F, an application $h : S \longrightarrow S'$ is a morphism of coalgebras if h transforms the states of S into those of S' and if $F(h)$ transforms the transitions of S into those of S' with respect to the following constraint: if there is a transition tr between two states of S, then there must be a transition tr' between the two corresponding states in S'.

Definition 2.1 *(**Morphisms of coalgebras**) Let F : **Set** $\longrightarrow$ **Set** be a functor. Let (S, α) and (S', α') be two F-coalgebras. A **morphism of coalgebras** is a function $h : S \longrightarrow S'$ making the following diagram commute i.e. $\alpha' \circ h = F(h) \circ \alpha$.*



Now, let us consider a concrete example to better illustrate the notion of a morphism of coalgebras. Figure III.1 depicts two label transition systems LTS and LTS$'$ with the same set $A = \{a, b\}$ of labels. As one has already seen in Section 1.3, labeled transition systems are coalgebras of the functor $\mathcal{P}(A \times \_)$ with A as the set of labels. Then, LTS and LTS$'$ are respectively described as coalgebras $\mathcal{C} = (S, \alpha)$ and $\mathcal{C}' = (S', \alpha')$ as shown in Table III.1.
The initial states of LTS and LTS are respectively s_0 and s'_0 that are drawn as circles with a thick border.

A morphism from LTS to LTS$'$ is then a mapping $h : S \longrightarrow S'$ such that the above diagram commutes i.e. $F(h) \circ \alpha = \alpha' \circ h$ where $F(h) : \mathcal{P}(A \times S) \longrightarrow \mathcal{P}(A \times S')$ is defined for each $U \subseteq A \times S$ by:

$$F(h)(U) = \{(a, h(s)) \in A \times S' \mid (a, s) \in U\}$$

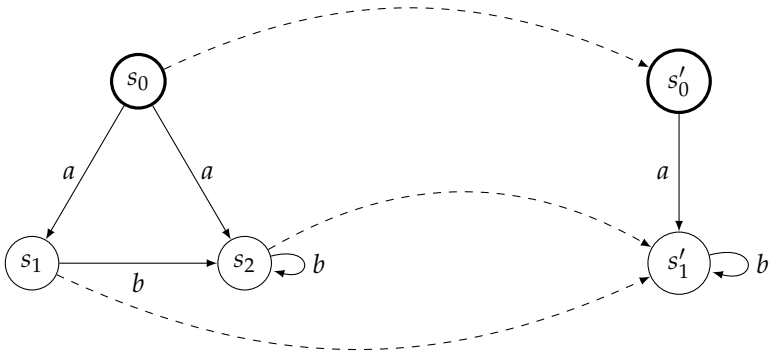


Figure III.1 – Graphical representation of LTS and LTS′

Characteristics \ Model	LTS	LTS′
Set of labels	$A = \{a, b\}$	$A = \{a, b\}$
State space	$S = \{s_0, s_1, s_2\}$	$S' = \{s'_0, s'_1\}$
Transitions Function	$\alpha : S \longrightarrow \mathcal{P}(\{a, b\} \times S)$ $s_0 \mapsto \{(a, s_1), (a, s_2)\}$ $s_1 \mapsto \{(b, s_2)\}$ $s_2 \mapsto \{(b, s_2)\}$	$\alpha' : S' \longrightarrow \mathcal{P}(\{a, b\} \times S')$ $s'_0 \mapsto \{(a, s'_1)\}$ $s'_1 \mapsto \{(b, s'_1)\}$

Table III.1 – LTS and LTS′

In our example, h is the mapping depicted with dotted line in Figure III.1. It is defined as follows:

$$h(s_0) = s'_0 \text{ and } h(s_1) = h(s_2) = s'_1$$

In order to prove that h is a morphism from LTS to LTS′, it is enough to verify that the following diagram is commutative:

$$\begin{array}{ccc} \{s_0, s_1, s_2\} & \xrightarrow{\quad h \quad} & \{s'_0, s'_1\} \\ \Big\downarrow \alpha & & \Big\downarrow \alpha' \\ \mathcal{P}(\{a, b\} \times \{s_0, s_1, s_2\}) & \xrightarrow[\mathcal{P}(\mathsf{id}_A \times h)]{} & \mathcal{P}(\{a, b\} \times \{s'_0, s'_1\}) \end{array}$$

Hence, we must check that for each state $s \in S$, one has $(\alpha' \circ h)(s) = (\mathcal{P}(\mathsf{id}_A \times h) \circ \alpha)(s)$.

$$\begin{array}{rcl}(\alpha' \circ h)(s_0) &=& \alpha'(h(s_0)) \\ &=& \alpha'(s'_0) \\ &=& \{(a, s'_1)\} \\ &=& \mathcal{P}(\mathsf{id}_A \times h)(\{(a, s_1), (a, s_2)\}) \\ &=& \mathcal{P}(\mathsf{id}_A \times h)(\alpha(s_0)) \\ &=& (\mathcal{P}(\mathsf{id}_A \times h) \circ \alpha)(s_0)\end{array}$$

$$\begin{array}{rcl}(\alpha' \circ h)(s_1) &=& \alpha'(h(s_1)) \\ &=& \alpha'(s'_1) \\ &=& \{(b, s'_1)\} \\ &=& \mathcal{P}(\mathsf{id}_A \times h)(\{(b, s_2)\}) \\ &=& \mathcal{P}(\mathsf{id}_A \times h)(\alpha(s_1)) \\ &=& (\mathcal{P}(\mathsf{id}_A \times h) \circ \alpha)(s_1)\end{array}$$

$$\begin{array}{rcl}(\alpha' \circ h)(s_2) &=& \alpha'(h(s_2)) \\ &=& \alpha'(s'_1) \\ &=& \{(b, s'_1)\} \\ &=& \mathcal{P}(\mathsf{id}_A \times h)(\{(b, s_2)\}) \\ &=& \mathcal{P}(\mathsf{id}_A \times h)(\alpha(s_2)) \\ &=& (\mathcal{P}(\mathsf{id}_A \times h) \circ \alpha)(s_2)\end{array}$$

Let us note that F-coalgebras with morphisms between them constitute a category, which noted $\mathbf{CoAlg}(F)$. Indeed, there is a forgetful functor $\mathbf{CoAlg}(F) \longrightarrow \mathbf{Set}$ mapping a F-coalgebra (S, α) to its state space S, and a F-morphism coalgebra h to itself.

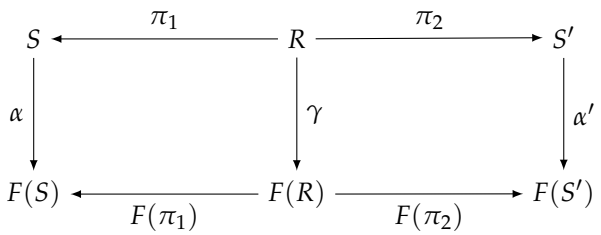
3 Bisimulation

The notion of bisimulation was first introduced by *Milner* for the calculus of communicating systems (CSS) language [35, 68], and it is usually defined as follows: a bisimulation between two labeled transition systems (S, A, T) and (S', A, T') is a relation $R \subseteq S \times S'$ such that for every pair of states $(s, s') \in R$, the two conditions are satisfied:

- For each transition $(s, a, q) \in T$, there is a state $q' \in S'$ such that $(s', a, q') \in T'$ and $(q, q') \in R$, and
- Symmetrically, for each transition $(s', a, q') \in T'$, there is a state $q \in S$ such that $(s, a, q) \in T$ and $(q, q') \in R$.

Given two states $s \in S$ and $s' \in S'$, s is *bisimilar* to s', written $s \sim s'$, if there is a bisimulation R such that (s, s') is in R.

This notion of bisimulation was redefined in terms of coalgebras as follows: Given a functor F and two F-coalgebras (S, α) and (S', α'). A relation $R \subseteq S \times S'$ is a bisimulation between S and S' if there exists a coalgebra $(R, \gamma : R \longrightarrow F(R))$ such that the projections $\pi_1 : R \longrightarrow S$ and $\pi_2 : R \longrightarrow S'$ are morphisms of F-coalgebras.



We detail in the following some examples of bisimulation.

3.1 Stream

A bisimulation $R \subseteq S \times S'$ between two automata streams $(S, \langle \mathsf{obs}, \mathsf{next} \rangle : S \longrightarrow A \times S)$ and $(S', \langle \mathsf{obs}', \mathsf{next}' \rangle : S' \longrightarrow A \times S')$ is a coalgebra $(R, \gamma : R \longrightarrow (A \times R))$ such that the function γ has to satisfy the following propriety: for all pair of states $(s, s') \in S \times S'$, we have:

$$\gamma((s,s')) = (a, (\mathsf{next}(s), \mathsf{next}(s'))) \text{ such that } \begin{cases} \mathsf{obs}(s) = \mathsf{obs}(s') = a \\ (\mathsf{next}(s), \mathsf{next}(s')) \in R \end{cases}$$

In order to verify that R is a stream bisimulation between S and S', we have to prove that the projection π_1 (respectively π_2) is a morphism of coalgebras from (R, γ) to $(S, \langle \mathsf{obs}, \mathsf{next} \rangle)$ (respectively from (R, γ) to $(S', \langle \mathsf{obs}', \mathsf{next}' \rangle)$). Hence, we must check:

$$\langle \mathsf{obs}, \mathsf{next} \rangle \circ \pi_1 = F(\pi_1) \circ \gamma \tag{III.1}$$

$$\langle \mathsf{obs}', \mathsf{next}' \rangle \circ \pi_2 = F(\pi_2) \circ \gamma \tag{III.2}$$

For Equation III.1, for each pair of states $(s, s') \in R$, one has:

$$\begin{array}{lcl} (\langle \mathsf{obs}, \mathsf{next} \rangle \circ \pi_1)((s,s')) & = & \langle \mathsf{obs}, \mathsf{next} \rangle (\pi_1((s,s'))) \\ & = & \langle \mathsf{obs}, \mathsf{next} \rangle (s) \\ & = & (\mathsf{obs}(s), \mathsf{next}(s)) \\ (F(\pi_1) \circ \gamma)((s,s')) & = & (F(\pi_1)(\gamma((s,s'))) \\ & = & F(\pi_1)(\mathsf{obs}(s), (\mathsf{next}(s), \mathsf{next}'(s'))) \\ & = & (\mathsf{obs}(s), \mathsf{next}(s)) \end{array}$$

Similarly we obtain Equation III.2.

3.2 Mealy machines

A bisimulation between two Mealy automata:

$$(S, \langle \mathsf{obs}, \mathsf{next} \rangle : S \longrightarrow (\mathsf{Out} \times S)^{\mathsf{In}}) \text{ and } (S', \langle \mathsf{obs}', \mathsf{next}' \rangle : S' \longrightarrow (\mathsf{Out} \times S')^{\mathsf{In}})$$

is a coalgebra $(R, \gamma : R \longrightarrow (\mathsf{Out} \times R)^{\mathsf{In}})$ such that its transition function γ is defined for each pair of states $(s, s') \in R$ and for each input $i \in \mathsf{In}$ by:

$$\gamma((s,s'), i) = (o, (\mathsf{next}(s)(i), \mathsf{next}'(s')(i)))$$

where $o = \mathsf{obs}(s)(i) = \mathsf{obs}'(s')(i)$ and $(\mathsf{next}(s)(i), \mathsf{next}'(s')(i)) \in R$.

3.3 Labeled transition systems

Two labeled transition systems are bisimilar if their labeling is equivalent, and their behaviour cannot be distinguished. A bisimulation $R \subseteq S \times S'$ between two labeled transition systems (S, α) and (S', α') over $\mathcal{P}(A \times \_)$ where A is the set of labels, is a coalgebra

$$(R, \gamma : R \longrightarrow \mathcal{P}(A \times R))$$

where γ is the function defined for each pair of states $(s, s') \in R$ by:

$$\gamma((s,s')) = \{(a, (q,q')) \mid (a,q) \in \alpha(s),\ (a,q') \in \alpha'(s') \text{ and } (q,q') \in R\}$$

Let us illustrate this definition by a concrete example. Consider again the two labeled transition systems LTS and LTS′ depicted in Figure III.1. In order to prove that LTS and LTS′ are bisimilar we first have to choose an appropriate relation R and then to check that it is indeed a bisimulation. Let us take the coalgebra (R, γ) with $R = \{(s_0, s'_0), (s_1, s'_1), (s_2, s'_1)\}$ is a relation between S and S' and $\gamma : R \longrightarrow \mathcal{P}(A \times R)$ is a function defined as follows:

$$\gamma((s_0, s'_0)) = \{(a, (s_1, s'_1)), (a, (s_2, s'_1))\} \text{ and } \gamma((s_1, s'_1)) = \gamma((s_2, s'_1)) = \{(a, (s_2, s'_1))\}$$

and let us verify that the following diagram commutes:

$$\begin{array}{ccccc}
\{s_0, s_1, s_2\} & \xleftarrow{\pi_1} & \{(s_0, s'_0), (s_1, s'_1), (s_2, s'_1)\} & \xrightarrow{\pi_2} & \{s'_0, s'_1\} \\
\downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle \gamma} & & \downarrow{\scriptstyle \alpha'} \\
\mathcal{P}(A \times S) & \xleftarrow[\mathcal{P}(A \times \pi_1)]{} & \mathcal{P}(A \times R) & \xrightarrow[\mathcal{P}(A \times \pi_2)]{} & \mathcal{P}(A \times S')
\end{array}$$

That is to say, for each pair of states $(s, s') \in R$, one has:

$$(\alpha \circ \pi_1)(s, s') = (\mathcal{P}(A \times \pi_1) \circ \gamma)(s, s') \tag{III.3}$$

$$(\alpha' \circ \pi_2)(s, s') = (\mathcal{P}(A \times \pi_2) \circ \gamma)(s, s') \tag{III.4}$$

For Equation III.3, for (s_0, s'_0) one has:

$$\begin{array}{lcl}
(\alpha \circ \pi_1)(s_0, s'_0) & = & \alpha(\pi_1((s_0, s'_0))) \\
& = & \alpha(s_0) \\
& = & \{(a, s_1), (a, s_2)\} \\
(\mathcal{P}(A \times \pi_1) \circ \gamma)(s_0, s'_0) & = & (\mathcal{P}(A \times \pi_1)(\gamma((s_0, s'_0))) \\
& = & \mathcal{P}(A \times \pi_1)(\{(a, (s_1, s'_1)), (a, (s_2, s'_1))\} \\
& = & \{(a, s_1), (a, s_2)\}
\end{array}$$

For (s_1, s'_1) one has:

$$\begin{array}{lcl}
(\alpha \circ \pi_1)(s_1, s'_1) & = & \alpha(\pi_1((s_1, s'_1))) \\
& = & \alpha(s_1) \\
& = & \{(b, s_2)\} \\
(\mathcal{P}(A \times \pi_1) \circ \gamma)(s_1, s'_1) & = & (\mathcal{P}(A \times \pi_1)(\gamma((s_1, s'_1))) \\
& = & \mathcal{P}(A \times \pi_1)(\{(b, (s_2, s'_1))\} \\
& = & \{(b, s_2)\}
\end{array}$$

For (s_2, s'_1) one has:

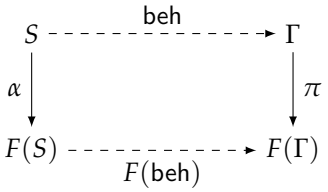
$$\begin{array}{lcl}
(\alpha \circ \pi_1)(s_2, s'_1) & = & \alpha(\pi_1((s_2, s'_1))) \\
& = & \alpha(s_2) \\
& = & \{(b, s_2)\} \\
(\mathcal{P}(A \times \pi_1) \circ \gamma)(s_2, s'_1) & = & (\mathcal{P}(A \times \pi_1)(\gamma((s_2, s'_1))) \\
& = & \mathcal{P}(A \times \pi_1)(\{(b, (s_2, s'_1))\} \\
& = & \{(b, s_2)\}
\end{array}$$

Similarly we obtain Equation III.4.

4 Final coalgebras

In this section, we represent final coalgebras which play a central role in the theory of coalgebras. They provide in fact an abstract model of all possible behaviours of a system.

Definition 4.1 *(**Final coalgebra**) Let F be a functor. A **final** F**-coalgebra** (Γ, π) is a F-coalgebra such that for every F-coalgebra (S, α), there is a unique*[3] *coalgebra morphism* $\mathsf{beh} : (S, \alpha) \longrightarrow (\Gamma, \pi)$ *such that the following diagram commutes:*



A final coalgebra, when it exits, contains every possible observable behaviour. It can be seen as the maximal representation containing all possible observations of a system. Once the final coalgebra for a functor F is defined, we can associate to any state s of an arbitrary F-coalgebra its behaviour $\mathsf{beh}(s)$, that is a state of the final coalgebra which is bisimilar with it.

Theorm 4.1 *(**Gumm [56]**) Let F be a signature functor. If F admits a final coalgebra (Γ, π), then for every F-coalgebra (S, α), and for every state $s \in S$, there is a unique state $u = \mathsf{beh}(s) \in \Gamma$ such that u and s are bisimilar.*

Moreover, it has been shown that states of the final coalgebra coincide with behaviours. Then, for any functor F, bisimilarity implies behavioural equivalence. We have the fundamental result:

Theorm 4.2 *(**Rutten and Turi [69]**) Let F be a functor. Let (S, α) be an arbitrary F-coalgebra, let (Γ, π) be the final F-coalgebra and let $\mathsf{beh} : S \longrightarrow \Gamma$ be the unique morphism from S to Γ. Then, for each pair of states $s, s' \in S$, one has:*

$$s \sim_S s' \Longrightarrow \mathsf{beh}(s) = \mathsf{beh}(s')$$

This result then states that to show equality between two states, it is enough to show that they are mapped to the same state in the final coalgebra.

In most cases, the unique existence of a final coalgebra (Γ, π) for a functor F is what one needs to know. We focus precisely on the uniqueness of the morphism beh which maps a state of an arbitrary F-coalgebra to its behaviour in Γ, rather than on the internal structure of the space state Γ or the form of the transition function π. Hence, we are often interested in defining final coalgebras without building their elements, ensuring their uniqueness up to isomorphism.

Theorm 4.3 *(**Lambek [70]**) Let F be a functor. If F admits a final coalgebra $\pi : \Gamma \longrightarrow F(\Gamma)$, then it is necessary an isomorphism $\pi : \Gamma \xrightarrow{\cong} F(\Gamma)$.*

One class of functors for which a final coalgebra always exists is the class of finite Kripke polynomial functors. As already mentioned, finite Kripke polynomial functors are endofunctors of the category **Set** which include the identity functor, the constant functors, and are closed by product, coproduct, exponent, and finite powerset.

3. The dotted notation is used to express the uniqueness of the morphism.

Theorm 4.4 *Each finite Kripke polynomial functor* **Set** $\longrightarrow$ **Set** *has a final coalgebra.*

In the following, we describe some concrete examples of final coalgebras to become more familiar with them. We refer to [71, 72, 73] for results on the existence and construction of final coalgebras.

4.1 Streams

In this subsection, we describe the final coalgebra for stream automata. Assume we have a state $s \in S$ of such a stream automata $(S, \langle \mathsf{obs}, \mathsf{next} \rangle : S \longrightarrow A \times S)$. Apply $\langle \mathsf{obs}, \mathsf{next} \rangle$ on s yields an output $\mathsf{obs}(s) \in A$ as well as its successor state $s' \in S$. In the same manner, $\langle \mathsf{obs}, \mathsf{next} \rangle$ can be again applied on s', and then produce a new couple $(\mathsf{obs}(s'), \mathsf{next}(s')) \in A \times S$. Hence, in this way, we can get for each state $s \in S$, an infinite sequence of outputs:

$$(\mathsf{obs}(s), \mathsf{obs}(\mathsf{next}(s)), \mathsf{obs}(\mathsf{next}(\mathsf{next}(\mathsf{obs}(s)))), \dots)$$

This sequence is formally obtained by the following definition:

Definition 4.2 ***(Observable behaviour of streams)***
Let $FX = A \times X$ be a functor. Let $\mathcal{C} = (S, \langle \mathsf{obs}_{\mathcal{C}}, \mathsf{next}_{\mathcal{C}} \rangle)$ be a F-coalgebra, let s be a state in S and let $n \in \mathbb{N}$. The ***observable behaviour*** *of s is defined by:*

$$(\mathsf{beh}(s))(n) = \mathsf{obs}_{\mathcal{C}}(\mathsf{next}^n_{\mathcal{C}}(s))$$

where next^n is inductively defined via:

$$\begin{cases} \mathsf{next}^0_{\mathcal{C}}(s) & = & s \\ \mathsf{next}^{n+1}_{\mathcal{C}}(s) & = & \mathsf{next}_{\mathcal{C}}(\mathsf{next}^n_{\mathcal{C}}(s)) \end{cases}$$

The set of states Γ of the final coalgebra is then the set $A^{\mathbb{N}} = \{f : \mathbb{N} \to A\} = \{(a_i)_{i \in \mathbb{N}} \mid a_i \in A\}$ of all infinite sequences $(a_i)_{i \in \mathbb{N}}$ over A, and its transition function π is the cartesian product of the two following functions:

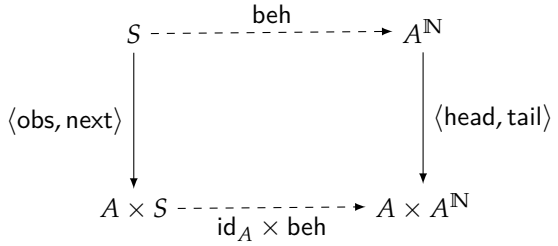
$$\begin{array}{rccc} \langle \mathsf{head}, \mathsf{tail} \rangle : & A^{\mathbb{N}} & \to & A \times A^{\mathbb{N}} \\ & (a_0, a_1, a_2, \dots) & \mapsto & (a_0, (a_1, a_2, \dots)) \end{array}$$

which are defined for any function $f : \mathbb{N} \longrightarrow A$ as follows:

$$\mathsf{head}(f) = f(0) \text{ and } \mathsf{tail}(f) = f' \text{ such that } \forall n \in \mathbb{N}, f'(n) = f(n+1)$$

Thus, everything we can possibly observe about a state $s \in S$ is obtained via the function $\mathsf{beh} : S \longrightarrow A^{\mathbb{N}}$. It assigns to a state $s \in S$, the sequence of outputs $\mathsf{beh}(s) \in A^{\mathbb{N}}$ which is generated on the unique path starting from s.

In order to prove that $(A^{\mathbb{N}}, \langle \mathsf{head}, \mathsf{tail} \rangle : A^{\mathbb{N}} \longrightarrow A \times A^{\mathbb{N}})$ is indeed a final coalgebra of the functor $FX = A \times X$, it suffices to show that the function $\mathsf{beh} : S \longrightarrow A^{\mathbb{N}}$ is the unique morphism making the following diagram commute.



In fact, it leads to two points:

1. verify that beh is a morphism of coalgebras from S to $A^{\mathbb{N}}$;
2. verify that beh is the unique morphism from S to $A^{\mathbb{N}}$.

For the first point, we need to prove that for any $s \in S$, one has:

$$(\mathsf{head} \circ \mathsf{beh})(s) = (\mathsf{id}_A \circ \mathsf{obs})(s) \text{ and } (\mathsf{tail} \circ \mathsf{beh})(s) = (\mathsf{beh} \circ \mathsf{next})(s)$$

$$\begin{array}{rcl}
(\mathsf{head} \circ \mathsf{beh})(s) & = & \mathsf{head}(\mathsf{beh}(s)) \\
& = & \mathsf{head}((\mathsf{obs}(s), \mathsf{obs}(\mathsf{next}(s)), \mathsf{obs}(\mathsf{next}(\mathsf{next}(s))), \dots)) \\
& = & \mathsf{obs}(s) \\
& = & \mathsf{id}_A(\mathsf{obs}(s)) \\
& = & (\mathsf{id}_A \circ \mathsf{obs})(s)
\end{array}$$

$$\begin{array}{rcl}
(\mathsf{tail} \circ \mathsf{beh})(s) & = & \mathsf{tail}(\mathsf{beh}(s)) \\
& = & \mathsf{tail}((\mathsf{obs}(s), \mathsf{obs}(\mathsf{next}(s)), \mathsf{obs}(\mathsf{next}(\mathsf{next}(s))), \dots) \\
& = & (\mathsf{obs}(\mathsf{next}(s)), \mathsf{obs}(\mathsf{next}(\mathsf{next}(s))), \dots) \\
& = & \mathsf{beh}(\mathsf{next}(s)) \\
& = & (\mathsf{beh} \circ \mathsf{next})(s)
\end{array}$$

We still have to show the second point. Let us assume that $f : S \longrightarrow A^{\mathbb{N}}$ is also a morphism of coalgebras which satisfies $(\mathsf{head} \circ f = \mathsf{id}_A \circ \mathsf{obs})$ and $(\mathsf{tail} \circ f = f \circ \mathsf{next})$, and then prove that for any $s \in S$, one has $\mathsf{beh}(s) = f(s)$.
First of all, we need to define an auxiliary function $\mathsf{tail}^n : A^{\mathbb{N}} \longrightarrow A^{\mathbb{N}}$ that takes a sequence $\sigma \in A^{\mathbb{N}}$ and returns the sequence $\sigma' \in A^{\mathbb{N}}$ containing all the elements after the index n in σ. More formally, it is defined for any $\sigma = (a_i)_{i \in \mathbb{N}} \in A^{\mathbb{N}}$ and for any $n \in \mathbb{N}$ by:

$$\mathsf{tail}^n(\sigma) = (a_{i+n})_{i \in \mathbb{N}}$$

Proposition 4.1 *For every morphism $f : S \longrightarrow A^{\mathbb{N}}$ and for any $n \in \mathbb{N}$, one has:*

$$(f(s))(n) = \mathsf{obs}(\mathsf{next}^n(s)) \tag{III.5}$$

Proof *Let us assume $f(s) = (a_0, a_1, a_2, \dots, a_n, \dots)$. On one hand, we have:*

$$\mathsf{head}(\mathsf{tail}^n(f(s))) = \mathsf{head}((a_n, a_{n+1}, \dots)) = a_n = (f(s))(n) \tag{III.6}$$

On the other hand, since f is a morphism, we have $\mathsf{tail} \circ f = f \circ \mathsf{next}$ which implies $\mathsf{tail}^n \circ f = f \circ \mathsf{next}^n$. Then, we have:

$$\begin{array}{rcl}
\mathsf{head}(\mathsf{tail}^n(f(s))) & = & \mathsf{head}(f(\mathsf{next}^n(s)) \\
& = & (\mathsf{id}_A \circ \mathsf{obs})(\mathsf{next}^n(s)) \\
& = & \mathsf{obs}(\mathsf{next}^n(s))
\end{array} \tag{III.7}$$

Consequently, Equation III.5 is obtained as an equality between Equation III.6 and Equation III.7.
End

We have just proved that for any $n \in \mathbb{N}$, $(f(s))(n) = \mathsf{obs}(\mathsf{next}^n(s))$. We have therefore $\mathsf{beh}(s) = f(s)$. Thus, beh is the unique morphism of coalgebras from S to $A^{\mathbb{N}}$.

4.2 Mealy machines

Given a Mealy machine $(S, \langle \mathsf{obs}, \mathsf{next} \rangle : S \times \mathsf{In} \longrightarrow (\mathsf{Out} \times S))$. Its observable behaviour (i.e. a representation in which only inputs and outputs of the machine are involved) is obtained as follows: assume we have a state $s \in S$ and an input $i \in \mathsf{In}$. Apply $\langle \mathsf{obs}, \mathsf{next} \rangle$ to s produces an output $\mathsf{obs}((s,i)) \in \mathsf{Out}$ and leads the machine state to its successor state $\mathsf{next}((s,i)) \in S$. In the same way, we can re-apply $\langle \mathsf{obs}, \mathsf{next} \rangle$ on $\mathsf{next}(s)$ for a new input $i' \in \mathsf{In}$, and then produces an output $\mathsf{obs}((\mathsf{next}(((s,i)), i')) \in \mathsf{Out}$ and the successor state $\mathsf{next}((\mathsf{next}(((s,i)), i')))$. Continuing in this way[4], for each finite sequence of inputs $(i_0, i_1, \ldots, i_n)$, we can get a finite sequence of outputs $(o_0, o_1, \ldots, o_n)$. Thus, everything we can observe about a state $s \in S$ after receiving a finite sequence of inputs $\sigma \in \mathsf{In}^+$ is obtained via the function $\mathsf{beh}(s) : \mathsf{In}^+ \longrightarrow \mathsf{Out}^+$ which assigns to any finite non-empty sequence of inputs $(i_0, i_1, \ldots, i_n) \in \mathsf{In}^+$, the finite non-empty sequence of outputs $(o_0, o_1, \ldots, o_n) \in \mathsf{Out}^+$.

Definition 4.3 *(**Observable behaviour of Mealy machines**) Let $FX = (\mathsf{Out} \times X)^{\mathsf{In}}$ be a functor. Let $\mathcal{C} = (S, \langle \mathsf{obs}_{\mathcal{C}}, \mathsf{next}_{\mathcal{C}} \rangle)$ be a F-coalgebra, let s be a state in S and let $(i_0, i_1, \ldots, i_n) \in \mathsf{In}^+$. The **observable behaviour** of s after receiving $(i_0, i_1, \ldots, i_n)$ is defined by:*

$$\mathsf{beh}(s) = (\mathsf{obs}_{\mathcal{C}}(\mathsf{next}^*_{\mathcal{C}}(s)(i_0)), \mathsf{obs}_{\mathcal{C}}(\mathsf{next}^*_{\mathcal{C}}(s)((i_0, i_1))), \ldots, \mathsf{obs}_{\mathcal{C}}(\mathsf{next}^*_{\mathcal{C}}(s)((i_0, i_1, \ldots, i_n))))$$

where the function next^* is inductively defined via:

$$\begin{cases} \mathsf{next}^*_{\mathcal{C}}(s)(i) & = \mathsf{next}_{\mathcal{C}}(s)(i) \\ \mathsf{next}^*_{\mathcal{C}}(s)((i_0, i_1, \ldots, i_n)) & = \mathsf{next}^*_{\mathcal{C}}(\mathsf{next}_{\mathcal{C}}(s)(i_0))((i_1, i_2, \ldots, i_n)) \end{cases}$$

The set of states Γ of the final coalgebra is then the set

$$\{\phi \mid \phi \in (\mathsf{Out}^+)^{\mathsf{In}^+}\} = \{\phi \mid \phi : \mathsf{In}^+ \longrightarrow \mathsf{Out}^+\}$$

and its transition function π is:

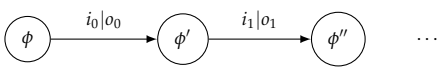
$$\begin{array}{rccc} \pi : & \Gamma \times \mathsf{In} & \to & (\mathsf{Out} \times \Gamma) \\ & (\phi, i) & \mapsto & \langle \phi(i), \phi' \rangle \end{array}$$

where $\phi' : \mathsf{In}^+ \longrightarrow \mathsf{Out}^+$ is defined for each $\sigma \in \mathsf{In}^+$ by:

$$\phi'(\sigma)(n) = \phi(\sigma)(n+1),\ \forall n \in \mathbb{N}$$

Hence, the function π splits the elements of Γ into couples in $(\mathsf{Out} \times \Gamma)$ as follows: let $\phi \in \Gamma$ and $i \in \mathsf{In}$:

- $\phi(i)$ is the output associated to the current state;
- $\phi' : \mathsf{In}^+ \longrightarrow \mathsf{Out}^+$ is the function which associates to every input sequence $(i_1, \ldots, i_n) \in \mathsf{In}^+$ the same sequence of outputs $(o_1, o_2, \ldots, o_n)$ that is obtained by the function ϕ when the input sequence $(i_0, i_1, i_2, \ldots, i_n)$ is applied to it.



Rutten proposed in [41, 74] another way to define the final coalgebra of Mealy automata using transfer functions. He showed that the set of all causal stream functions $\Gamma = \{\mathcal{F} : \mathsf{In}^\omega \longrightarrow \mathsf{Out}^\omega \mid \mathcal{F} \text{ is causal}\}$ carries itself the structure of a Mealy coalgebra via the notions of initial output and stream function derivative.

4. In^+ denotes the set of finite non-empty sequences over In.

Definition 4.4 *(**Derivative function**) Let $\mathcal{F} : \mathsf{In}^\omega \longrightarrow \mathsf{Out}^\omega$ be a transfer function. We define:*

1. *the **initial output** on input $i \in \mathsf{In}$ by $\mathcal{F}[i] = \mathcal{F}(i.\sigma)(0)$ and*
2. *the **derivative function** on input $i \in \mathsf{In}$ by $\mathcal{F}_i : \mathsf{In}^\omega \longrightarrow \mathsf{Out}^\omega$ with $\mathcal{F}_i(\sigma) = \mathcal{F}(i.\sigma)'$*

for any $\sigma \in \mathsf{In}^\omega$ chosen arbitrarily.

Let us now define the coalgebra (Γ, π) as follows:

- $\Gamma = \{\mathcal{F} : \mathsf{In}^\omega \longrightarrow \mathsf{Out}^\omega \mid \mathcal{F} \text{ is causal}\}$;
- $\pi : \Gamma \times \mathsf{In} \longrightarrow (\mathsf{Out} \times \Gamma)$ is the transition function defined for every $\mathcal{F} \in \Gamma$ and for every $i \in \mathsf{In}$ by:

$$\pi(\mathcal{F})(i) = \langle \mathcal{F}[i], \mathcal{F}_i \rangle$$

Theorm 4.5 *The Mealy machine (Γ, π) defined above is a final Mealy machine coalgebra: for every Mealy machine (S, α) there is a unique homomorphism $!_\alpha : S \longrightarrow \Gamma$ from (S, α) to (Γ, π).*

Proof *For every Mealy machine coalgebra (S, α), let us define $!_\alpha : S \longrightarrow \Gamma$ which for every state $s \in S$, associates the transfer function $!_\alpha(s) : \mathsf{In}^\omega \longrightarrow \mathsf{Out}^\omega$ which is defined for every $s \in S$, every $\sigma \in \mathsf{In}^\omega$ and every $k \in \omega$ as follows:*

$$!_\alpha(s)(\sigma(k)) = o_k \text{ such that } s_0 \xrightarrow{\sigma(0)|o_0} s_1 \xrightarrow{\sigma(1)|o_1} \dots \xrightarrow{\sigma(k)|o_k} s_{k+1}$$

where there exists an infinite sequence of states $s_0, \dots, s_k, s_{k+1} \in S$ and $s_0 = s$.
It is not difficult to check that $!_\alpha$ is causal, and that $!_\alpha$ defined in this way is the unique homomorphism making the diagram below commute.

$$\begin{array}{ccc} S & \overset{!_\alpha}{\dashrightarrow} & \Gamma \\ {\scriptstyle\alpha}\downarrow & & \downarrow{\scriptstyle\pi} \\ (\mathsf{Out} \times S)^{\mathsf{In}} & \underset{(\mathsf{id}_{\mathsf{Out}} \times !_\alpha)^{\mathsf{id}_{\mathsf{In}}}}{\dashrightarrow} & (\mathsf{Out} \times \Gamma)^{\mathsf{In}} \end{array}$$

End

4.3 Labeled transition systems

Consider again the finitely branching labeled transition systems that we previously described as coalgebras $(S, \alpha : S \longrightarrow \mathcal{P}_{\text{fin}}(A \times S))$ of the functor $FX = \mathcal{P}_{\text{fin}}(A \times X)$ with A as an alphabet of actions. By Theorem 4.3, the powerset functor $\mathcal{P}$ does not admit a final coalgebra. This is indeed due to *Cantor*'s theorem [75, 76] that states that, for any set X, its powerset $\mathcal{P}(X)$ has a strictly greater cardinality than X itself. That means there is no bijection between X and $\mathcal{P}(X)$ (there is in fact no injection from $\mathcal{P}(X)$ to X). However, by Theorem 4.4, the finite powerset $\mathcal{P}_{\text{fin}}$ belonging to the class of polynomial functors, admits a final coalgebra.

In the following, we then describe the final coalgebra (Γ, π) of finitely branching labeled transition systems. The state set Γ that we will note $ST^A_\sim$ is the equivalence classes of so-called *synchronization trees* over A, *modulo* bisimilarity. We can in fact look at the synchronization trees [35] as transition systems without circuits whose states (or nodes) are represented implicitly and transitions (or arcs) carry labels of A. Intuitively, a synchronization tree is a representation that captures at once all possible behaviours of a system. It can be obtained by unfolding

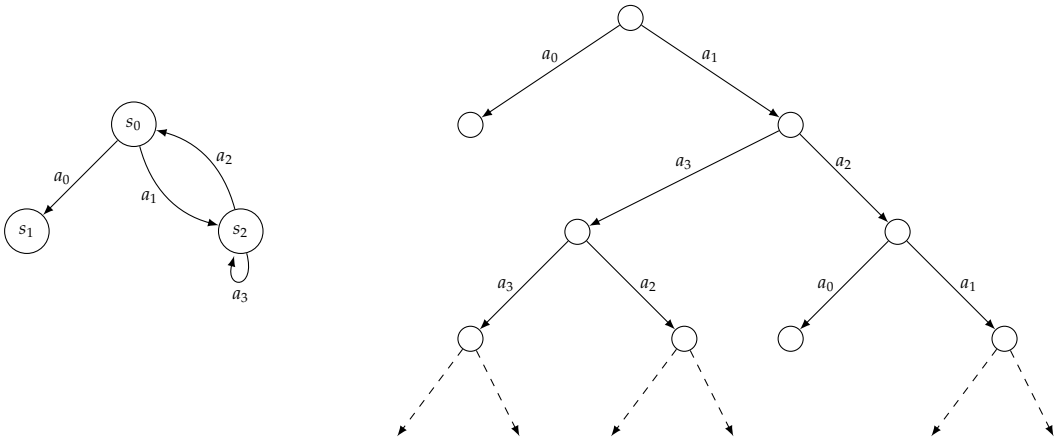


Figure III.2 – Example of a synchronization tree

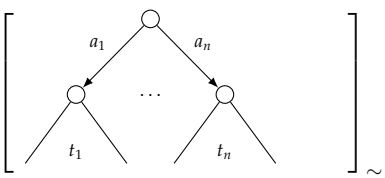
the system starting from its initial state. The right part of Figure III.2 shows a labeled transition system, while the left part shows its unfolding starting from the initial state s_0 as a synchronization tree.

Let us note some points:

- Synchronization trees have finite branches and eventually infinite paths. Many nodes (potentially infinitely many) of the synchronization tree represent the same state of the original labeled transition system: they correspond in fact to different visits to the state;
- Nodes of synchronization trees are represented implicitly (they are not labeled). Information on nodes is not present in the behaviour;
- Two states which are bisimilar in the labeled transition systems are mapped to the same synchronization tree in $TS^A_\sim$ because $TS^A_\sim$'s elements are equivalence classes *modulo* bisimilarity $\sim$.

The transition function $\pi : TS^A_\sim \longrightarrow \mathcal{P}_{\text{fin}}(A \times TS^A_\sim)$ is the application splitting the synchronization tree into its immediate subtrees as follows:

$$TS^A_\sim \qquad \longrightarrow \qquad \mathcal{P}_{\text{fin}}(A \times TS^A_\sim)$$

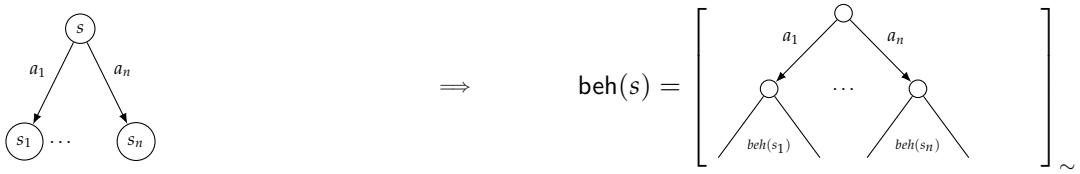


$$\mapsto \qquad \{(a_1, [t_1]_\sim), \ldots, (a_n, [t_n]_\sim)\}$$

Now, to prove that $(TS^A_\sim, \pi)$ is the final coalgebra, we need to verify the commutativity of the following diagram:

$$\begin{tikzcd}
S \arrow[r, dashed, "\mathsf{beh}"] \arrow[d, "\alpha"'] & TS^A_\sim \arrow[d, "\pi"] \\
\mathcal{P}_{\text{fin}}(A \times S) \arrow[r, dashed, "\mathcal{P}_{\text{fin}}(\mathsf{id}_A \times \mathsf{beh})"'] & \mathcal{P}_{\text{fin}}(A \times TS^A_\sim)
\end{tikzcd}$$

with beh is the function that associates to every state $s \in S$, the corresponding synchronization tree $\mathsf{beh}(s) \in TS^A_{\sim}$ as follows:



It is straightforward to verify the commutativity of the above diagram.

4.4 More examples

We present here more examples of systems modeling in terms of coalgebras. For each system, we give the corresponding functor as well as its final model without going into technical details. [5]

1. **Finite stream over a set of labels** A**:** is a coalgebra $(S, \alpha : S \longrightarrow (A \times S) \cup \{\bot\})$ of the functor $FX = (A \times X) + 1$ where $\bot$ is a state expressing the possibility of termination. The final coalgebra is $(A^\infty, \pi : A^\infty \longrightarrow (A \times A^\infty) \cup \{\bot\})$ where π is the function defined for $\sigma \in A^\infty$ by:

$$\pi(\sigma) = \begin{cases} \bot & \text{if} \quad \sigma = \epsilon \\ (a, \sigma') & \text{if} \quad \sigma = a.\sigma' \end{cases}$$

2. **Binary tree over a set of labels** A**:** is a coalgebra $(S, \alpha : S \longrightarrow ((A \times S) \times (A \times S)) \cup \{\bot\})$ of the functor $FX = ((A \times X) \times (A \times X)) + 1$. The final coalgebra is (Γ, π) where:

   – $$\Gamma = \{\phi : \{0,1\}^* \to (A \times A) \cup \{\bot\} \mid \forall v \in \{0,1\}^*, \phi(v) \in \{\bot\} \Longrightarrow (\forall w \in \{0,1\}^*, \phi(v.w) = \phi(v))\}$$

   is the set of all binary trees whose arcs are labeled with actions of A and whose branches are eventually infinites

   – $\pi : \Gamma \longrightarrow ((A \times \Gamma) \times (A \times \Gamma)) \cup \{\bot\}$ is the function defined for any tree $\phi \in \Gamma$ by:

$$\pi(\phi) = \begin{cases} \bot & \text{if} \quad \phi(\epsilon) = \bot \\ (\langle a_1, \phi_1 \rangle, \langle a_2, \phi_2 \rangle) & \text{if} \quad \phi(\epsilon) = \langle a_1, a_2 \rangle \end{cases}$$

   with for $i = 1, 2, \phi_i$ is defined for $v \in \{0,1\}^*$ by $\phi_i(v) = \phi(a_i.v)$.

3. **Deterministic automata:** is a coalgebra $(S, \alpha : S \longrightarrow \{0,1\} \times S^A)$ of the functor $FX = \{0,1\} \times X^A$. The final coalgebra is $(\mathcal{L}, \langle o_\mathcal{L}, d_\mathcal{L} \rangle)$ where $\mathcal{L} = \{L | L \subseteq A^*\}$ is the set all languages over the alphabet A, and $\langle o_\mathcal{L}, d_\mathcal{L} \rangle$ is the cartesian product of $o_\mathcal{L}$ and $d_\mathcal{L}$ that are defined for $a \in A$ and $L \in \mathcal{L}$ by:

$$o_\mathcal{L}(L) = \begin{cases} 1 & \text{if } \epsilon \in L \\ 0 & \text{otherwise} \end{cases} \qquad \text{and} \qquad d_\mathcal{L}(L)(a) = \{v \in A^* \mid a.v \in L\}$$

5. $A^\infty = A^* \cup A^{\mathbb{N}}$ is the set of finite and infinite sequences of elements over A, ϵ denotes the empty sequence and $\_.\_ : A^\infty \times A^\infty \longrightarrow A^\infty$) is the binary operation of concatenation.

4. **Moore automata:** is the coalgebra $(S, \alpha : S \longrightarrow \mathsf{Out} \times S^{\mathsf{In}})$ of the functor

$$FX = \mathsf{Out} \times X^{\mathsf{In}}$$

The final coalgebra is $(\mathsf{Out}^{\mathsf{In}^*}, \langle \mathsf{head}, \mathsf{tail} \rangle : \mathsf{Out}^{\mathsf{In}^*} \longrightarrow \mathsf{Out} \times \mathsf{Out}^{\mathsf{In}^*})$ where $\langle \mathsf{head}, \mathsf{tail} \rangle$ is the function defined for $\phi : \mathsf{In}^* \longrightarrow \mathsf{Out} \in \mathsf{Out}^{\mathsf{In}^*}$ by:

$$\langle \mathsf{head}, \mathsf{tail} \rangle(\phi) = \langle \phi(\epsilon), \phi' \rangle$$

with ϕ' is the function defined for $i \in \mathsf{In}$ and $\sigma \in \mathsf{In}^*$ by:

$$\phi'(i)(v) = \phi(i.\sigma)$$

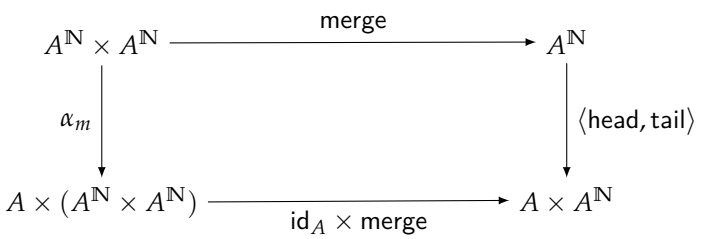
5 Co-induction

Due to the fact that coalgebras are duals of algebras, definitions and results known from universal algebra were dualized to the coalgebra theory such as coinduction principle which we present in this section. The coinduction principle is the categorical dual of induction which, in a categorical setting, refers to the use of the initiality principle for algebras. In fact, induction is used to build new data entities from data entities already constructed. On the contrary, coinduction is used to observe potentially infinite data entities whose structure may contain patterns that repeat infinitely. It does not tell us how to build objects, it tells us only what we can observe on them. As mentioned in the previous section, once we know the final coalgebra (Γ, π) of a given functor F, we can use its finality to define functions into its carrier set Γ and to prove properties. Hence, existence of final coalgebra allows us to define functions (it is called *coinduction definition principle*) while its uniqueness allows us to prove properties of the functions already constructed (it is called *coinduction proof principle*).

Example 5.1 ***(Definition of functions by coinduction)*** *In this example, we define by coinduction, the function* $\mathsf{merge} : A^{\mathbb{N}} \times A^{\mathbb{N}} \longrightarrow A^{\mathbb{N}}$ *which builds a new sequence by interleaving the elements of two sequences. For this, it is enough to verify that the function* merge *is a morphism of coalgebras from an appropriate coalgebra with the same domain of* merge *i.e.* $A^{\mathbb{N}} \times A^{\mathbb{N}}$ *to the final coalgebra* $(A^{\mathbb{N}}, \langle \mathsf{head}, \mathsf{tail} \rangle : A^{\mathbb{N}} \longrightarrow A \times A^{\mathbb{N}})$ *of the functor* $FX = A \times X$*. Let us take a coalgebra whose carrier set is* $A^{\mathbb{N}} \times A^{\mathbb{N}}$ *and whose transition function is* $\alpha_{\mathsf{m}} : A^{\mathbb{N}} \times A^{\mathbb{N}} \longrightarrow A \times (A^{\mathbb{N}} \times A^{\mathbb{N}})$ *defined by:*

$$\forall (\sigma_1, \sigma_2) \in A^{\mathbb{N}} \times A^{\mathbb{N}}, \ \ \alpha_m((\sigma_1, \sigma_2)) = (\mathsf{head}(\sigma_1), (\sigma_2, \mathsf{tail}(\sigma_1)))$$

Now, by the conduction definition principle, we can directly conclude the existence of the morphism $\mathsf{merge} : A^{\mathbb{N}} \times A^{\mathbb{N}} \longrightarrow A^{\mathbb{N}}$ *which makes the following diagram commute:*



The fact that this diagram is commutative gives indeed:

$$\mathsf{head}(\mathsf{merge}((\sigma_1, \sigma_2)) = \mathsf{head}(\sigma_1) \textit{ and } \mathsf{tail}(\mathsf{merge}((\sigma_1, \sigma_2))) = (\mathsf{merge}((\sigma_2, \mathsf{tail}(\sigma_1))) \qquad \text{(III.8)}$$

as expected.

Example 5.2 ***(Proof by coinduction)*** *Let us consider a property of the function* merge *that is:*

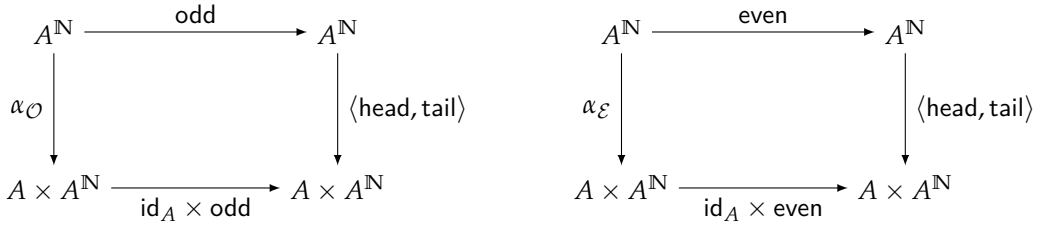
$$\textit{for any sequence } \sigma \in A^{\mathbb{N}}, \mathsf{merge}((\mathsf{odd}(\sigma), \mathsf{even}(\sigma))) = \sigma$$

and then prove it by conduction. For this, we first coinductively define two functions: $\mathsf{odd} : A^{\mathbb{N}} \longrightarrow A^{\mathbb{N}}$ *and* $\mathsf{even} : A^{\mathbb{N}} \longrightarrow A^{\mathbb{N}}$*. The first one, maps each sequence* $\sigma \in A^{\mathbb{N}}$ *to a new sequence* $\mathsf{odd}(\sigma)$ *containing only elements of* σ *at odd positions and the second one maps each sequence to a new sequence* $\mathsf{even}(\sigma)$ *containing only elements of* σ *at even positions. Similarly to* merge*, we define two coalgebras*

$$\mathcal{O} = (A^{\mathbb{N}}, \alpha_{\mathcal{O}} : A^{\mathbb{N}} \longrightarrow A \times A^{\mathbb{N}}) \textit{ and } \mathcal{E} = (A^{\mathbb{N}}, \alpha_{\mathcal{E}} : A^{\mathbb{N}} \longrightarrow A \times A^{\mathbb{N}})$$

of the functor $FX = A \times X$ *where* $\alpha_{\mathcal{O}}$ *and* $\alpha_{\mathcal{E}}$ *are defined respectively for any sequence* $\sigma \in A^{\mathbb{N}}$ *by:*[6]

$$\alpha_{\mathcal{O}}(\sigma) = (\mathsf{head}(\mathsf{tail}(\sigma)), \mathsf{tail}^3(\sigma)) \textit{ and } \alpha_{\mathcal{E}}(\sigma) = (\mathsf{head}(\sigma), \mathsf{tail}^2(\sigma)))$$



Then, the commutativity of the above diagrams gives the two following equations: for any sequence $\sigma \in A^{\mathbb{N}}$*:*

$$\mathsf{head}(\mathsf{even}(\sigma)) = \mathsf{head}(\sigma) \textit{ and } \mathsf{tail}(\mathsf{even}(\sigma)) = \mathsf{even}(\mathsf{tail}(\mathsf{tail}(\sigma))) \tag{III.9}$$

$$\mathsf{head}(\mathsf{odd}(\sigma)) = \mathsf{head}(\mathsf{tail}(\sigma)) \textit{ and } \mathsf{tail}(\mathsf{odd}(\sigma)) = \mathsf{odd}(\mathsf{tail}(\mathsf{tail}(\mathsf{tail}(\sigma)))) \tag{III.10}$$

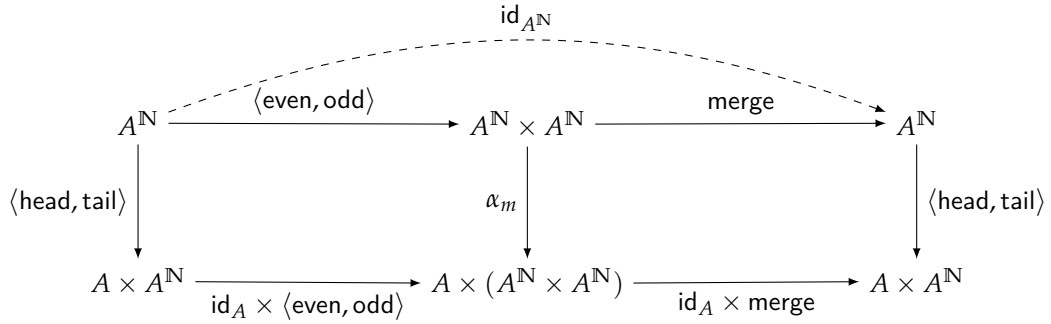
We can also easily prove that:

$$\mathsf{odd}(\sigma)) = \mathsf{even}(\mathsf{tail}(\sigma)) \textit{ and } \mathsf{even}(\mathsf{tail}^2(\sigma)) = \mathsf{odd}(\mathsf{tail}(\sigma)) \tag{III.11}$$

Now, we need to show that the composite function

$$\mathsf{merge} \circ \langle \mathsf{even}, \mathsf{odd} \rangle : A^{\mathbb{N}} \longrightarrow A^{\mathbb{N}}$$

and the identity function on $A^{\mathbb{N}}$ *are equal. It suffices then to prove that they are both morphisms for the same coalgebra structure on* $A^{\mathbb{N}}$*. It is not difficult to see that the identity function* $\mathsf{id}_{A^{\mathbb{N}}} : A^{\mathbb{N}} \to A^{\mathbb{N}}$ *is indeed a morphism of coalgebras from* $(A^{\mathbb{N}}, \langle \mathsf{head}, \mathsf{tail} \rangle)$ *to* $(A^{\mathbb{N}}, \langle \mathsf{head}, \mathsf{tail} \rangle)$*. We still have to show that* $\mathsf{merge} \circ \langle \mathsf{even}, \mathsf{odd} \rangle$ *is also a morphism from* $A^{\mathbb{N}}$ *to* $A^{\mathbb{N}}$*. For this, it is enough to show the commutativity of the following diagram:*



6. The function $\mathsf{tail}^n : A^{\mathbb{N}} \longrightarrow A^{\mathbb{N}}$ has been defined in the previous section for $\sigma = (a_i)_{i \in \mathbb{N}} \in A^{\mathbb{N}}$ and $n \in \mathbb{N}$ by $\mathsf{tail}^n(\sigma) = (a_{i+n})_{i \in \mathbb{N}}$

This amounts to show that:

$$\langle \mathsf{head}, \mathsf{tail} \rangle \circ (\mathsf{merge} \circ \langle \mathsf{even}, \mathsf{odd} \rangle) = (\mathsf{id}_A \times (\mathsf{merge} \circ \langle \mathsf{even}, \mathsf{odd} \rangle)) \circ \langle \mathsf{head}, \mathsf{tail} \rangle$$

This is proved for $\sigma \in A^{\mathbb{N}}$ by:

$$\begin{array}{lll}
\mathsf{head}(\mathsf{merge}(\mathsf{even}(\sigma), \mathsf{odd}(\sigma))) = \mathsf{head}(\mathsf{even}(\sigma)) & & \textit{see Equation III.8} \\
= \mathsf{head}(\sigma) & & \textit{see Equation III.9} \\
\\
\mathsf{tail}(\mathsf{merge}(\mathsf{even}(\sigma), \mathsf{odd}(\sigma))) = \mathsf{merge}(\mathsf{odd}(\sigma), \mathsf{tail}(\mathsf{even}(\sigma))) & & \textit{see Equation III.8} \\
= \mathsf{merge}(\mathsf{odd}(\sigma), \mathsf{even}(\mathsf{tail}(\mathsf{tail}(\sigma)))) & & \textit{see Equation III.9} \\
= \mathsf{merge}(\mathsf{even}(\mathsf{tail}(\sigma)), \mathsf{odd}(\mathsf{tail}(\sigma))) & & \textit{see Equation III.11} \\
= \mathsf{merge}(\langle \mathsf{even}, \mathsf{odd} \rangle (\mathsf{tail}(\sigma))) & & \\
= \mathsf{merge} \circ \langle \mathsf{even}, \mathsf{odd} \rangle)(\mathsf{tail}(\sigma)) & &
\end{array}$$

5.1 Proof by bisimulation

There is another alternative for proving properties. The underlying idea is that two states are behaviourally equivalent if and only if they are bisimilar (see Theorem 4.2). This result indeed gives rise to a proof method. Then, two states are equals if and only if they are contained in a bisimulation relation.

Corollary 5.1 *Two states s and s' have the same behaviour if and only if there is a bisimulation R such as $(s, s') \in R$. For every bisimulation R on the final[7] coalgebra (Γ, π), one has $R \subseteq \Delta_\Gamma$ where $\Delta_\Gamma = \{(s, s) \mid s \in \Gamma\}$. Equivalently, for all s and s' in Γ, one has:*

$$s \sim s' \iff s = s'$$

Example 5.3 ***(Proof by bisimulation)*** *Let us consider again the property:*

$$\forall \sigma \in A^{\mathbb{N}}, \mathsf{merge}((\mathsf{odd}(\sigma), \mathsf{even}(\sigma))) = \sigma$$

proved in Example 5.2, and then prove it using the notion of bisimulation.
Our aim is to prove the equality:

$$\mathsf{merge}(\mathsf{odd}(\sigma), \mathsf{even}(\sigma)) = \sigma$$

Then, we need a bisimulation $R \subseteq A^{\mathbb{N}} \times A^{\mathbb{N}}$ containing both sides of the equation. We take:

$$R = \{(\mathsf{merge}(\mathsf{odd}(\sigma), \mathsf{even}(\sigma)), \sigma) \mid \sigma \in A^{\mathbb{N}}\}$$

Recall from Section 3 that a relation $R \subseteq A^{\mathbb{N}} \times A^{\mathbb{N}}$ is a bisimulation if the two projections $\pi_1 : R \longrightarrow A^{\mathbb{N}}$ and $\pi_2 : R \longrightarrow A^{\mathbb{N}}$ are morphisms of coalgebras from the coalgebra (R, γ) to $(A^{\mathbb{N}}, \langle \mathsf{head}, \mathsf{tail} \rangle)$ where γ is defined as follows:

$$\begin{array}{rccl}
\gamma : & R & \longrightarrow & A \times R \\
& (\sigma_1, \sigma_2) & \mapsto & (\mathsf{head}(\sigma_1), (\mathsf{tail}(\sigma_1), \mathsf{tail}(\sigma_2)))
\end{array}$$

By finality of $A^{\mathbb{N}}$, it is obvious to show that π_1 and π_2 are morphisms of coalgebras from

$$(\{(\mathsf{merge}((\mathsf{odd}(\sigma), \mathsf{even}(\sigma))), \sigma) | \sigma \in A^{\mathbb{N}}\}, \gamma) \textit{ to } (A^{\mathbb{N}}, \langle \mathsf{head}, \mathsf{tail} \rangle)$$

Then, one has $\mathsf{merge}((\mathsf{odd}(\sigma), \mathsf{even}(\sigma))) = \sigma$.

7. Recall s and s' define behaviours belonging to the final coalgebra.

Example 5.4 *Let $a^{\mathbb{N}}$ and $b^{\mathbb{N}}$ be two concrete infinite streams and let us show that* $\mathsf{merge}(a^{\mathbb{N}}, b^{\mathbb{N}}) = (ab)^{\mathbb{N}}$. *For this, we first define a relation $R \subseteq A^{\mathbb{N}} \times A^{\mathbb{N}}$ containing the following pairs:*

$$(\mathsf{merge}(a^{\mathbb{N}}, b^{\mathbb{N}}), (ab)^{\mathbb{N}}) \text{ and } (\mathsf{merge}(b^{\mathbb{N}}, a^{\mathbb{N}}), (ba)^{\mathbb{N}})$$

R is a bisimulation if for any pair of streams $(\sigma_1, \sigma_2) \in R$, one has:

$$R((\sigma_1, \sigma_2)) \Longrightarrow \begin{cases} \mathsf{head}(\sigma_1) & = & \mathsf{head}(\sigma_2) \\ \mathsf{tail}(\sigma_1) & = & \mathsf{tail}(\sigma_2) \end{cases}$$

Consider the first pair $(\mathsf{merge}(a^{\mathbb{N}}, b^{\mathbb{N}}), (ab)^{\mathbb{N}})$ *of R, the only transition step of its left component is:*

$$\mathsf{merge}(a^{\mathbb{N}}, b^{\mathbb{N}}) \xrightarrow{a} \mathsf{merge}(b^{\mathbb{N}}, a^{\mathbb{N}})$$

whereas its right component can take the step:

$$(ab)^{\mathbb{N}} \xrightarrow{a} (ba)^{\mathbb{N}}$$

Hence, the resulting pair $(\mathsf{merge}(b^{\mathbb{N}}, a^{\mathbb{N}}), (ba)^{\mathbb{N}})$ *is again in the relation R.*
In the same manner, we can show that the resulting pair of the second pair $(\mathsf{merge}(b^{\mathbb{N}}, a^{\mathbb{N}}), (ba)^{\mathbb{N}})$ *is* $(\mathsf{merge}(a^{\mathbb{N}}, b^{\mathbb{N}}), (ab)^{\mathbb{N}})$ *which is in the relation R.*
Consequently, R is a bisimulation. Corollary 5.1 tells us that $R \subseteq \Delta_{A^{\mathbb{N}}}$, proving then the equality:

$$\mathsf{merge}(a^{\mathbb{N}}, b^{\mathbb{N}}) = (ab)^{\mathbb{N}}$$

Part II

Systems modeling framework

This part provides the first contribution of this thesis. It presents a formal abstract framework for developing, in a compositional way, complex software systems viewed as component-based systems. It intends to contribute to the following topics:

– The definition of a generic modeling of components. This will then enable us to unify a wide family of state-based formalisms classically used to specify components, such as Mealy machines [33, 34], labeled transition systems (LTS) [35] and input-output labeled transition systems (IOLTS) [39, 40]. We will see that the generality of our formalization will be obtained by taking into account various kinds of computation structures such as non-determinism, partiality, etc. [16].

– The definition of a trace model over components using causal transfer functions [41, 74, 77] as is usual in control theory or physics when dealing with dynamic system modeling.

– The definition of a minimal and unified set of component integration operators able to take into account the interaction semantics present in most modern systems. Such integration operators will be used to combine component behaviours that interact together in order to build a larger system.

This part consists of two chapters. The first chapter introduces a generic unified coalgebraic model of components enabling us to naturally describe a large family of state-based formalisms as well as its trace model given as transfer functions. The second chapter introduces how components can be composed to obtain systems. It presents two basic integration operators: *cartesian product* and *feedback*, and shows that both seem sufficient to build most other standard operators such as synchronous product and sequential, double sequential, concurrent and synchronous parallel operators, by composition.

Chapter IV

Generic components

1 Components as coalgebras 74
1.1 Motivation 74
1.2 Components 74
1.3 Genericity of component definition 77
2 Component traces 80
2.1 Transfer function 80
2.2 Component Traces 81
3 Results 82
3.1 Final model 83
3.2 Minimal component 85
4 Conclusion 88

This chapter presents a formal generic framework for developing basic components viewed as state-based systems. As explained in the introduction, our formalization is based on *Barbosa*'s components [9, 10, 11, 12, 13]. We then start, in Section 1, by introducing *Barbosa*'s definition of a component as well as some explicative concrete examples. We further show that this component definition is powerful enough to be a unified generic state-based formalism by providing some basic examples. In Section 2 we define, by relying on the work proposed by *Rutten* in [41], a trace model over components using causal transfer functions. The formalization of components and their traces as coalgebras and transfer functions respectively, allows us to extend standard results connected to the definition of a final component in Section 3. We show, over some assumptions, the existence of a final model which will be useful to define the basic integration operators in Chapter V. Finally, Section 4 concludes with some final remarks and an assessment of the results.

1 Components as coalgebras

1.1 Motivation

In a series of papers [9, 10, 11], *Barbosa* and his colleagues used the coalgebra theory to define a generic notion of a state-based software component. In these works, a *component* was then introduced as a generalized Mealy automaton in which the dependence between outputs and both current state and inputs is relaxed from a strict deterministic, to encompass more complex behaviours such as partiality, non determinism, etc. The reasons for using *Barbosa*'s definition are twofold:

- It fits to the standard view of functional components that is, at a high level of abstraction, components may be considered as black boxes that take inputs and provide appropriate outputs. The behaviour of a component is then specified by describing how inputs drive changes in component state and how outputs are produced.
- It is considered as an excellent modeling tool for representing several kinds of computational effects, such as: determinism, non-determinism and partiality, as well as abstractly providing a unifying formal framework in which a great diversity of state-based formalisms used to describe dynamical systems behaviour, such as: Mealy machines [33, 34], Labeled Transition Systems (LTS) [35, 36], Input-Output Labeled Transition Systems (IOLTS) [39, 40] can be naturally captured (see Section 1.2). This is due to the introduction of monads in the component definition.

1.2 Components

Definition 1.1 *(**Components**) Let* In *and* Out *be two sets denoting, respectively, the input and output domains. Let* T *be a monad. A* ***component*** $\mathcal{C}$ *is a coalgebra* $(S, init, \alpha)$ *for the signature* $H = T(\mathsf{Out} \times \_)^{\mathsf{In}} : \mathbf{Set} \longrightarrow \mathbf{Set}$ *where* $init \in S$ *is a distinguished element denoting the initial state of the component* $\mathcal{C}$.
When the initial state init is removed, $\mathcal{C}$ *is called* ***a pre-component****.*

Barbosa further requires that the monad T should be strong [78]. That means T is equipped with two natural transformations: $\tau^r : TX \times Y \longrightarrow T(X \times Y)$ and $\tau^l : X \times TY \longrightarrow T(X \times Y)$ for any sets X and Y, called the *right* and *left strength* respectively, satisfying certain coherent conditions [78]. This requirement is mainly supposed to be able to link in [13] computations carried by T. For instance, when composing two components $\mathcal{C}_1$ and $\mathcal{C}_2$ over $T(\mathsf{Out}_1 \times S_1)$ and $T(\mathsf{Out}_2 \times S_2)$ respectively, τ_r followed by τ_l allows the mapping of

$$T(\mathsf{Out}_1 \times S_1) \times T(\mathsf{Out}_2 \times S_2) \text{ to } TT((\mathsf{Out}_1 \times S_1) \times (\mathsf{Out}_2 \times S_2))$$

This latter can then be flattened to $T((\mathsf{Out}_1 \times S_1) \times (\mathsf{Out}_2 \times S_2))$ via μ.
We do not require that the monad T should be *strong* as it was assumed in the original definition introduced in [13]. We rather assume the existence of two natural transformations η' and η'^{-1} that we will present in Section 2. These two natural transformations will be useful not only to link computations carried by T, like strong monads do, when composing components in Chapter V but also to define a trace model over components in Section 2.

Example 1.1 *(**Coffee machine**) We consider a simple example of a* coffee machine $\mathcal{M}$ *modeled by the transition diagram shown in Figure IV.1. The behaviour of* $\mathcal{M}$ *is the following: from its initial state* STDBY, *when it receives a coin from the user, it goes into the* READY *state. Then, when the user presses the "coffee" button, three cases are distinguished:* (1) $\mathcal{M}$ *serves a coffee to the user and goes back to the*

STDBY *state;* (2) $\mathcal{M}$ *serves a coffee to the user and goes to the* FAILED *state (for example, when there is one cup in the machine);* (3) $\mathcal{M}$ *fails to deliver coffee to the user and so refunds him and goes to the* FAILED *state. The only escape from the* FAILED *state is to have a repair.*

This machine can be modeled as a component $\mathcal{M} = (S, init, \alpha)$ *over the signature* $\mathcal{P}_{\text{fin}}(\mathsf{Out} \times \_)^{\mathsf{In}}$. *The state space is* $S = \{\text{STDBY}, \text{READY}, \text{FAILED}\}$ *and* $init = \text{STDBY}$. *The sets of inputs and outputs are* $\mathsf{In} = \{\text{coin}, \text{coffee}, \text{repair}\}$ *and* $\mathsf{Out} = \{\mathsf{abs}, \text{served}, \text{refund}\}$. *Finally, the transition function:*

$$\alpha : S \longrightarrow \mathcal{P}_{\text{fin}}\big(\{\mathsf{abs}, \text{served}, \text{refund}\} \times S\big)^{\{\text{coin},\text{coffee},\text{repair}\}}$$

is defined as follows:

$$\begin{cases} \alpha(\text{STDBY})(\text{coin}) = \big\{(\mathsf{abs}, \text{READY})\big\} \\ \alpha(\text{READY})(\text{coffee}) = \big\{(\text{served}, \text{STDBY}), (\text{served}, \text{FAILED}), (\text{refund}, \text{FAILED})\big\} \\ \alpha(\text{FAILED})(\text{repair}) = \big\{(\mathsf{abs}, \text{STDBY})\big\} \end{cases}$$

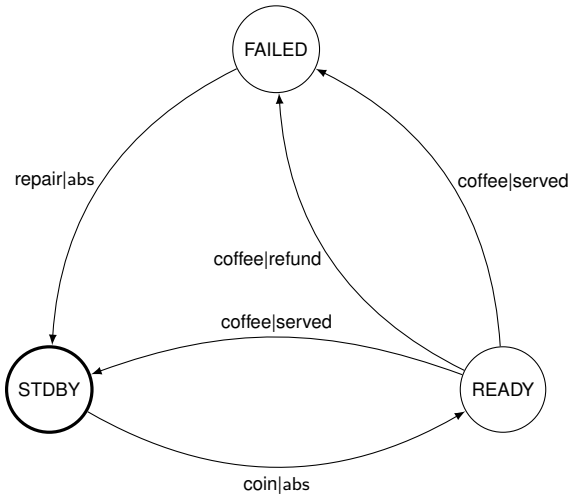


Figure IV.1 – Coffee machine

Example 1.2 ***(ATM)*** *We consider a simple specification of a* bank Automated Teller Machine *(ATM)* $\mathcal{M}$ *modeled by the transition diagram shown in Figure IV.2. The behaviour of* $\mathcal{M}$ *is the following: from its initial state* s_0*, it is waiting either for a request for amount (the "*amount*" action) or a request for bank account balance (the "*check*" action):*

- *if the user enters amount,* $\mathcal{M}$ *goes to* s_1*. An internal test is then made to check whether the received amount can be withdrawn (represented by the action* τ*). If there is enough money in the customer's account, the requested amount is withdrawn and the ATM returns to its initial state (the "*cash*" action). If there is not enough money in the customer's account, the withdraw operation fails and an error message is displayed on the machine screen (the "*screen*" action).*
- *If the user asks for the amount of his/her account,* $\mathcal{M}$ *does an internal action and then notifies the user by the amount (the "*sold*" action). It then returns back to its initial state* s_0*.*

This machine can be modeled[1] *as a component* $\mathcal{M} = (S, init, \alpha)$ *over the signature* $(\mathsf{Out} \times \_)^{\mathsf{In}}$. *The state space is* $S = \{s_0, s_1, s_2, s_3, s_4\}$ *and* $init = s_0$*. The sets of inputs and outputs are*

$$\mathsf{In} = \{\text{amount}, \text{check}, \text{sold}, \text{screen}, \text{cash}, \tau\} \text{ and } \mathsf{Out} = \{\mathsf{abs}\}$$

1. We deliberately omitted the abs action to make the representation easier. Hence, transitions are labeled with (a | abs) are simply represented as transitions labeled with a.

respectively. Finally, the transition function:

$$\alpha : \{s_0, s_1, s_2, s_3, s_4\} \times \{\text{amount}, \text{check}, \text{sold}, \text{screen}, \text{cash}, \tau\} \longrightarrow \big(\{\text{abs}\} \times \{s_0, s_1, s_2, s_3, s_4\}\big)$$

is defined as follows:

$$\begin{cases} \alpha(s_0)(\text{amount}) = (\text{abs}, s_1) \\ \alpha(s_0)(\text{check}) = (\text{abs}, s_3) \\ \alpha(s_1)(\tau) = (\text{abs}, s_2) \end{cases} \qquad \begin{cases} \alpha(s_3)(\tau) = (\text{abs}, s_4)\} \\ \alpha(s_2)(\text{screen}) = (\text{abs}, s_0) \\ \alpha(s_2)(\text{cash}) = (\text{abs}, s_0) \\ \alpha(s_4)(\text{sold}) = (\text{abs}, s_0) \end{cases}$$

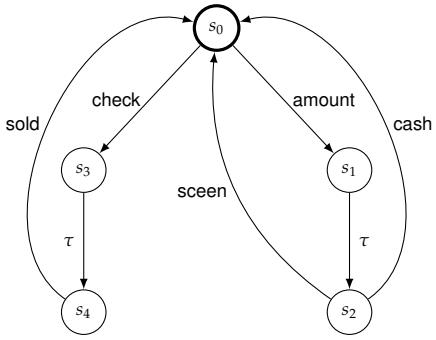


Figure IV.2 – *ATM* component

Example 1.3 *(**Pedestrian crossing**) A* pedestrian crossing *is a portion of a roadway where pedestrians are permitted to cross the roadway. A typical view of a pedestrian crossing can be seen in Figure IV.3. It consists of two essential parts: the road and the crosswalk. A pedestrian crossing system is then made to avoid any interactions between pedestrians that want to cross the crosswalk and vehicles that want to cross the road. A traffic light is then usually used to control pedestrian and vehicle flows at the crosswalk. Such a typical traffic light consists of three colors: green, orange and red, and behaves as follows: in the absence of any pedestrian request, the green color is illuminated. When a pedestrian wishes to cross the road, he then pushes the request button that sends a signal to the traffic light system to change its illumination to an orange light to prepare to stop the vehicles. Then, it switches to a red light prohibiting any vehicle flow at the road and yielding to the pedestrians. Once the crosswalk is free from pedestrians, the traffic light receives a signal saying the road is free and thus the green color is again illuminated. In this way, the traffic light system allows one to yield either to pedestrians or vehicles by repetitively making the green-orange-red-green cycle.*

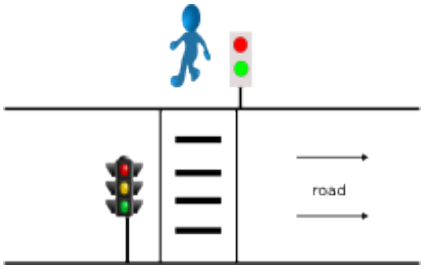


Figure IV.3 – Pedestrian crossing

The traffic light system can be modeled as a component $\mathcal{M} = (S, s_0, \alpha_{\mathcal{M}})$ *over the signature* $(\mathsf{Out} \times \_)^{\mathsf{In}}$. *The state space is* $S = \{s_0, s_1, s_2, s_3, s_4\}$. *The sets of inputs and outputs are:*

$\mathsf{In} = \{\text{stopLight}, \text{lightOk}, \mathsf{abs}\}$ *and* $\mathsf{Out} = \{\text{lightRed}, \text{lightOrange}, \text{lightGreen}, \text{pedestrianOk}, \mathsf{abs}\}$

Finally, the transition function:

$$\alpha_{\mathcal{M}} : S \longrightarrow (\{\text{lightRed}, \text{lightOrange}, \text{lightGreen}, \text{pedestrianOk}, \mathsf{abs}\} \times S)^{\{\text{stopLight},\text{lightOk},\mathsf{abs}\}}$$

is defined as follows:

$$\begin{cases} \alpha_{\mathcal{M}}(s_0)(\mathsf{abs}) = (\text{lightGreen}, s_0) \\ \alpha_{\mathcal{M}}(s_0)(\text{stopLight}) = (\text{lightGreen}, s_1) \\ \alpha_{\mathcal{M}}(s_1)(\mathsf{abs}) = (\text{lightOrange}, s_2) \end{cases} \quad \begin{cases} \alpha_{\mathcal{M}}(s_2)(\mathsf{abs}) = (\text{lightRed}, s_3) \\ \alpha_{\mathcal{M}}(s_3)(\mathsf{abs}) = (\text{pedestrianOk}, s_4) \\ \alpha_{\mathcal{M}}(s_4)(\text{lightOk}) = (\mathsf{abs}, s_0) \end{cases}$$

The graphical diagram of this component is shown in Figure IV.4.

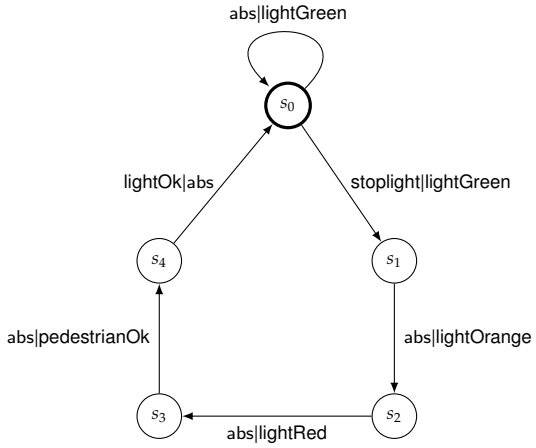


Figure IV.4 – Pedestrian crossing modeling

Definition 1.2 *(**Category of components**) Let* $\mathcal{C}$ *and* $\mathcal{C}'$ *be two components over* $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$. *A **component morphism*** $h : \mathcal{C} \longrightarrow \mathcal{C}'$ *is a coalgebra homomorphism* $h : (S, \alpha) \longrightarrow (S', \alpha')$ *such that* $h(init) = init'$.
We note $\mathbf{Comp}(H)$ *(resp.* $\mathbf{PComp}(H)$*) the **category of components** (resp. pre-components) over* H.

This will be in the category of pre-components $\mathbf{PComp}(H)$, where we will show the existence of a final model under some conditions (see Section 3).

1.3 Genericity of component definition

Definition 1.1 is generic enough to unify in a single framework a large family of formalisms classically used to specify state-based systems [79]. We describe the most important of them in three tables: Table IV.1, Table IV.2 and Table IV.3. Each table illustrates the possible models of dynamic systems that can be framed as instances of our component definition depending on the monad T. The first column consists of instances of the input set In, the second column consists of instances of the output set Out, the third column introduces the possible coalgebraic models

obtained by making a particular choice for In and Out, and the fourth column gives an example of which dynamic system can be obtained.

Table IV.1 suggests a possible[2] taxonomy of coalgebras that can be obtained when T is the identity functor id.

How to model it		Resulting component	Typical example
In	Out		
$\{\}$	$\{\mathsf{abs}\}$	$X \longrightarrow X$	run forever
$\{\}$	set of actions Act	$X \longrightarrow X \times Act$	infinite streams
set of inputs I	$\{\mathsf{abs}\}$	$X \longrightarrow X^I$	systems with input
set of inputs I	set of outputs O	$X \longrightarrow (X \times O)^I$	Mealy machines

Table IV.1 – The deterministic computational features

Table IV.2 suggests a possible taxonomy of coalgebras that can be obtained when T is the partial functor $(\mathsf{id} + \mathbf{1})$.

How to model it		Resulting component	Typical example
In	Out		
$\{\}$	$\{\mathsf{abs}\}$	$X \longrightarrow X + \mathbf{1}$	systems with interruption run
$\{\}$	set of actions Act	$X \longrightarrow (X \times Act) + \mathbf{1}$	finite streams
set of actions Act	$\{\mathsf{abs}\}$	$X \longrightarrow X^{Act} + \mathbf{1}$	systems with input
set of inputs I	set of outputs O	$X \longrightarrow (X \times O)^I + \mathbf{1}$	partial Mealy machines

Table IV.2 – The partial computational features

Table IV.3 suggests a possible taxonomy of coalgebras that can be obtained when T is the powerset functor $\mathcal{P}$.

How to model it		Resulting component	Typical example
In	Out		
$\{\}$	$\{\mathsf{abs}\}$	$X \longrightarrow \mathcal{P}(X)$	
$\{\}$	action set Act	$X \longrightarrow \mathcal{P}(Act \times X)$	*LTS* as in *Milner's CSS*
action set Act	$\{\mathsf{abs}\}$	$X \longrightarrow \mathcal{P}(X)^{Act}$	*LTS* as in *process algebra*
input set I	output set O	$X \longrightarrow \mathcal{P}(O \times X)^I$	extended Mealy machines
input set I	output set O	$X \longrightarrow \mathcal{P}((\{!\} \times O) \cup \{\mathsf{abs}_!\} \times X)^{(\{?\} \times I) \cup \{\mathsf{abs}_?\}}$	*IOLTS*

Table IV.3 – The non-deterministic computational features

2. abs, $\mathsf{abs}_?$ and $\mathsf{abs}_!$ are particular fresh action, input action and output action denoting the lack of reaction, input and output respectively.

To completely define *IOLTS*, we need to impose the supplementary property on the transition function $\alpha : S \longrightarrow \mathcal{P}(\mathsf{Out} \times S)^{\mathsf{In}}$:

$$\forall i \in \mathsf{In},\ \forall s \in S,\ (o,s) \in \alpha(s)(i) \Longrightarrow \text{either } i = \mathsf{abs}_? \text{ or } o = \mathsf{abs}_!$$

to express that input and output are mutually exclusive. This construct of α means that one can define three kinds of transition depending on the label that is allowed to appear in it.

- *Input-kind*: the transition can only be labeled by input action, that is $i \mid \mathsf{abs}_!$;
- *Output-kind*: the transition can only be labeled by output action, that is $\mathsf{abs}_? \mid o$;
- *Internal-kind*: the transition is labeled by $\mathsf{abs}_? \mid \mathsf{abs}_!$.

Therefore, a transition can be labeled by input or output [3] actions, but never both.

Each of these instances leads to obvious algorithms that transform each of previous models into components according to Definition 1.1. Let us illustrate this for *IOLTS* models.

Definition 1.3 *(IOLTS **as component**) Let $\Sigma = \Sigma^? \cup \Sigma^! \cup \{\tau\}$ be an alphabet of actions. Let $H = \mathcal{P}(\mathsf{Out} \times \_)^{\mathsf{In}}$ be the signature associated to IOLTS where*

$$\mathsf{In} = (\{?\} \times \Sigma^?) \cup \{\mathsf{abs}_?\} \text{ and } \mathsf{Out} = (\{!\} \times \Sigma^!) \cup \{\mathsf{abs}_!\}$$

*The **transformation of an** IOLTS **over** $\mathcal{IOLTS}(\Sigma)$ **into a component over** H is the application:*

$$\phi : \mathcal{IOLTS}(\Sigma) \longrightarrow \mathbf{Comp}(H)$$

that maps an IOLTS $\mathcal{M} = (Q, q_0, \Sigma, \mathsf{Tr})$ to a component [4] $\mathcal{C} = (S, s_0, \alpha)$ as follows:

- *$S = Q$ and $s_0 = q_0$;*
- *$\alpha : S \times \mathsf{In} \longrightarrow \mathcal{P}(\mathsf{Out} \times S)$ is the function defined by the following rules:*

$$\frac{s \xrightarrow{a}_{\mathsf{Tr}} s' \text{ and } a \in \Sigma^?}{\langle \mathsf{abs}_!, s' \rangle \in \alpha(s)(?a)} \qquad \frac{s \xrightarrow{a}_{\mathsf{Tr}} s' \text{ and } a \in \Sigma^!}{\langle !a, s' \rangle \in \alpha(s)(\mathsf{abs}_?)} \qquad \frac{s \xrightarrow{\tau}_{\mathsf{Tr}} s'}{\langle \mathsf{abs}_!, s' \rangle \in \alpha(s)(\mathsf{abs}_?)}$$

We illustrate this transformation with a simple example shown in Figure IV.5.

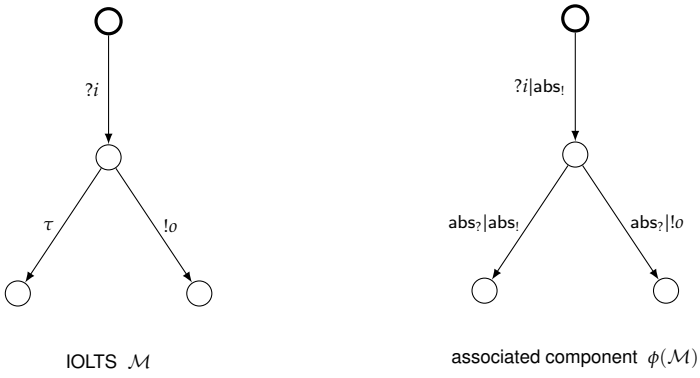


Figure IV.5 – Transformation of an *IOLTS* into a component over $\mathcal{P}(\mathsf{Out} \times \_)^{\mathsf{In}}$

3. Our transformation of an *IOLTS* into our framework is inspired from the works done by *M. Phalippou* in his thesis [38], that transform an *IOLTS* into finite state machine.

4. $\mathcal{IOLTS}(\Sigma)$ denotes the set of input-output labeled transitions systems over the alphabet Σ.

2 Component traces

As shown in Table IV.1, Mealy automata with input set In and output set Out are coalgebras of the functor $F : \mathbf{Set} \longrightarrow \mathbf{Set}$ defined by $F(X) = (\mathsf{Out} \times X)^{\mathsf{In}}$. For a coalgebraic modeling of Mealy automata, *Rutten* in [41] defined [5] the final coalgebra of F (i.e. the set of all possible observable behaviours) as *causal stream functions* (traditionally called *transfer functions*). Hence, it is shown in [41, 74] that the final coalgebra of F is isomorphic to the set of all causal functions from the set of infinite input sequences to the set of infinite output sequences equipped with the operations of initial output and stream function derivative. In this section, similarly to *Rutten*'s work, we show that the observable behaviour of our components can also be characterized by causal functions mapping infinite input sequences to infinite output sequences.

2.1 Transfer function

In the following, we note ω the least infinite ordinal, identified with the corresponding hereditarily transitive set.

Definition 2.1 *(**Dataflow**) A **dataflow** over a set of values A is a mapping $\sigma : \omega \to A$. The set of all dataflows over A is noted A^ω.*

Definition 2.2 *(**Derivative dataflow**) Let σ be a dataflow over a set A. The dataflow σ' **derivative** of σ is defined by: $\forall n \in \omega,\ \sigma'(n) = \sigma(n+1)$.*
For every $a \in A$, let us note $a.\sigma$ the dataflow σ defined by:

$$\sigma(0) = a \quad \textit{and} \quad \forall n \in \omega \setminus \{0\},\ \sigma(n) = \sigma(n-1)$$

Hence, $\sigma = \sigma(0).\sigma'$.

Transfer functions will be used to describe the observable behaviour of components. They can be seen as dataflow transformers satisfying the causality condition as this is classically done in control theory and physics for modeling dynamic systems [80], that is the output data at index n only depends on input data at indexes $0, \ldots, n$.

Definition 2.3 *(**Transfer function**) Let In and Out be two sets denoting, respectively, the input and output domains. A function $\mathcal{F} : \mathsf{In}^\omega \longrightarrow \mathsf{Out}^\omega$ is a **transfer function** if, and only if it is causal, that is:*

$$\forall n \in \omega, \forall \sigma_1, \sigma_2 \in \mathsf{In}^\omega, (\forall m, 0 \leq m \leq n, \sigma_1(m) = \sigma_2(m)) \Longrightarrow \mathcal{F}(\sigma_1)(n) = \mathcal{F}(\sigma_2)(n)$$

Causal transfer functions and the notion of transfer function derivative were first introduced by *Raney* in [77]. *Raney*'s [77] also showed that both composition of two causal transfer functions and the derivative of a causal transfer function are again causal.

Example 2.1 *The function $\mathcal{F} : \{0,1\}^\omega \longrightarrow \{0,1\}^\omega$ defined for every $\sigma \in \{0,1\}^\omega$ and every $k \in \omega$ by*

$$\mathcal{F}(\sigma)(k) = \big(\sum_{i=0}^{k} \sigma(i)\big) \bmod 2$$

is the transfer function that takes a sequence of bits $\sigma \in \{0,1\}^\omega$ and checks at each step k whether it has received an odd number of "ones". It then returns 0 if the number of "ones" is even, and 1 otherwise. In Example 3.1, we will define the component that implements it.

5. See Chapter III, Subsection 4.2 for more explanations for *Rutten*'s work.

2.2 Component Traces

To associate behaviours to components by transfer functions, we have to require the existence of two natural transformations $\eta' : T \Longrightarrow \mathcal{P}$ and $\eta'^{-1} : \mathcal{P} \Longrightarrow T$ such that $\eta'^{-1} \circ \eta' = \mathsf{id}_T$ where $\mathcal{P}$ is the powerset functor. Indeed, from a component (S, α), we need to "compute" for an infinite input sequence $\sigma \in \mathsf{In}^\omega$ all the outputs o after going through any sequence of states $(s_0, \ldots, s_k)$ such that s_j is obtained from s_{j-1} by $\sigma(j-1)$. However, we do not know how to characterize s_j with respect to $\alpha(s_{j-1})(\sigma(j-1))$ because nothing ensures that elements in $\alpha(s_{j-1})(\sigma(j-1))$ are (output, state) couples. Indeed, the monad T may yield a set with a structure different from $\mathsf{Out} \times S$. The mapping $\eta'_{\mathsf{Out}\times S}$ maps back to this structure. $\eta'^{-1}_{\mathsf{Out}\times S}$ is useful for going back to T when defining final models.

Most monads used to represent computation situations satisfy the above condition. For instance, for the monad $T : \mathcal{P}$, both η'_S and η'^{-1}_S are the identity on sets. For the functor $T : \mathsf{id} + \mathbf{1}$, η'_S associates the singleton $\{s\}$ to any $s \in S$ and the empty set to $\bot$, and η'^{-1}_S associates the state s to the singleton $\{s\}$ and $\bot$ to any other subset of S which is not a singleton. Let us observe that given a monad T, the couple (η', η'^{-1}) when it exists, is not necessarily unique. Indeed, for the monad $T = \mathsf{id}$, η'_S can still be defined as $s \mapsto \{s\}$. However, η'^{-1}_S is not unique. Indeed, any mapping η'^{-1}_S that associates the singleton $\{s\}$ to s, and every subset of S which is not a singleton to a given $s' \in S$, satisfies $\eta'^{-1}_S \circ \eta'_S = \mathsf{id}_S$.
Hence, in the following, given a signature $T(\mathsf{Out} \times \_)^{\mathsf{In}}$, we will assume given a couple (η', η'^{-1}) such that $\eta'^{-1} \circ \eta' = \mathsf{id}$.

In the following, we note $\eta'_{\mathsf{Out}\times S}(\alpha(s)(i))_{|_1}$ (*resp.* $\eta'_{\mathsf{Out}\times S}(\alpha(s)(i))_{|_2}$) the set composed of all first arguments (*resp.* second arguments) of couples in $\alpha(s)(i)$.

Let us now associate behaviours to components by their transfer functions. Let us consider a state $s \in S$ of such a component $\mathcal{C} = (S, \alpha)$ over $T(\mathsf{Out} \times \_)^{\mathsf{In}}$. Applying α to s after receiving an input $i_1 \in \mathsf{In}$ yields a set $\eta'_{\mathsf{Out}\times S}(\alpha(s)(i_1))$ of couples (output|successor state). Similarly, after receiving a new input $i_2 \in \mathsf{In}$, we can repeat this step for each state $s' \in \eta'_{\mathsf{Out}\times S}(\alpha(s)(i_1))_{|_2}$ and form another set of couples (output|successor state). Thus, we get for each infinite sequence of inputs $\langle i_1, i_2, \ldots \rangle \in \mathsf{In}^\omega$, a set of infinite sequences of outputs $\langle o_1, o_2, \ldots \rangle \in \mathsf{Out}^\omega$. All we can possibly observe about a state $s \in S$ is obtained in this way. More formally, this leads to:

Definition 2.4 *(**Component behaviour**) Let $\mathcal{C} = (S, init, \alpha)$ be a component over $T(\mathsf{Out} \times \_)^{\mathsf{In}}$. The **behaviour** of a state s of $\mathcal{C}$, noted $\mathsf{beh}_{\mathcal{C}}(\mathbf{s})$ is the set of transfer functions $\mathcal{F} : \mathsf{In}^\omega \longrightarrow \mathsf{Out}^\omega$ that associate to every $\sigma_1 \in \mathsf{In}^\omega$ a dataflow $\sigma_2 \in \mathsf{Out}^\omega$ such that there exists an infinite sequence of couples $(o_1, s_1), \ldots, (o_k, s_k), \cdots \in \mathsf{Out} \times S$ satisfying:*

$$\forall j \geq 1, (o_j, s_j) \in \eta'_{\mathsf{Out}\times S}(\alpha(s_{j-1})(\sigma_1(j-1)))$$

with $s_0 = s$, and for every $k < \omega$, $\sigma_2(k) = o_{k+1}$.
*Hence, $\mathcal{C}$**'s behaviour** is the set $\mathsf{beh}_{\mathcal{C}}(init)$.*

Example 2.2 *The behaviour $beh_{\mathcal{M}}(s_0)$ of the coffee machine $\mathcal{M}$ presented in Example 1.1 is defined by all the functions*

$$\mathcal{F}_\sigma : \{\mathsf{coin}, \mathsf{coffee}, \mathsf{repair}\}^\omega \longrightarrow \{\mathsf{abs}, \mathsf{served}, \mathsf{refund}\}^\omega$$

where $\sigma = n_1.n'_1.n_2.n'_2 \ldots n_i.n'_i \cdots \in \mathbb{N}^\omega$ defined by

$\mathcal{F}_\sigma : (\mathrm{coin.coffee})^{n_1}.(\mathrm{coin.coffee.repair})^{n'_1} \ldots (\mathrm{coin.coffee})^{n_i}.(\mathrm{coin.coffee.repair})^{n'_i} \ldots$
$\mapsto$

$(\mathsf{abs.served})^{n_1}.(\mathsf{abs.refund.abs})^{n'_1} \dots (\mathsf{abs.served})^{n_i}.(\mathsf{abs.refund.abs})^{n'_i} \dots$

where
$(\mathsf{coin.coffee})^0 = (\mathsf{coin.coffee.repair})^0 = (\mathsf{abs.served})^0 = (\mathsf{abs.refund.abs})^0 = \epsilon$ *(the empty word).*

Hence, the transfer function that would remain in the loop between the states STDBY *and* READY *could be defined by any function* $\mathcal{F}_\sigma$ *with* $\sigma = n_1.0.n_2.0 \dots n_i.0 \dots$

In the context of our work, we will need to use finite traces. Finite traces are finite sequences of couples (input | output) defined as follows :

Definition 2.5 ***(Component finite traces)*** *Let* $\mathcal{F} \in \mathsf{beh}_\mathcal{C}(init)$ *be a trace of a component* $\mathcal{C}$*. Let* $n \in \mathbb{N}$*. The* ***finite trace****, noted* $\mathcal{F}_{|_n}$*, of length n associated to* $\mathcal{F}$ *is the whole set of the finite sequence* $\langle i_0|o_0, \dots, i_n|o_n\rangle$ *such that there exists* $x \in \mathsf{In}^\omega$ *where for every* $j, 0 \leq j \leq n$*:*

- $x(j) = i_j$
- *and* $\mathcal{F}(x(j)) = o_j$

Hence, $Trace(\mathcal{C}) = \bigcup_{\mathcal{F} \in \mathsf{beh}_\mathcal{C}(init)} \bigcup_{n \in \mathbb{N}} \mathcal{F}_{|_n}$ *defines the whole set of finite traces over* $\mathcal{C}$*.*

Taking advantage of having generically defined components, and having shown that most state-based formalisms are instances of Definition 1.1, by Definition 2.4, we can associate to them semantics from causal functions. We have formally presented in Section 1.2 the mapping $\phi : \mathcal{IOLTS}(\Sigma) \longrightarrow \mathbf{Comp}(H)$ that defines how an *IOLTS* can be transformed in our framework (see Definition 1.3). Similarly, we define another transformation ϕ_t allowing us to make the link between the set of traces of an *IOLTS* model $\mathcal{M}$ and its associated component $\phi(\mathcal{M})$ in our framework.

Definition 2.6 *Let* $\mathcal{M} \in \mathcal{IOLTS}(\Sigma_\tau)$ *be an IOLTS and* $tr \in Trace(\mathcal{M})$*. Let* $\phi(\mathcal{M}) \in \mathbf{Comp}(H)$ *be its associated component.* ***The transformation of*** $Trace(\mathcal{M})$ ***into*** $Trace(\phi(\mathcal{M}))$ *is the function:*

$$\phi_t(\mathcal{M}) : Trace(\mathcal{M}) \longrightarrow Trace(\phi(\mathcal{M}))$$

which is inductively defined as follows:

- *for every*[6] *trace* $tr = \epsilon \in trace(\mathcal{M}), \phi_t(tr) = \epsilon$*;*
- *for every trace* $tr = ?i \in trace(\mathcal{M})$*,* $\phi_t(tr) = ?i|\mathsf{abs}_!$*;*
- *for every trace* $tr = !o \in trace(\mathcal{M})$*,* $\phi_t(tr) = \mathsf{abs}_?|!o$*;*
- *for every trace* $tr = \tau \in trace(\mathcal{M})$*,* $\phi_t(tr) = \mathsf{abs}_?|\mathsf{abs}_!$*;*
- *for every trace* $tr = a_1 \dots a_n \in trace(\mathcal{M})$ *with* $a \in \Sigma_\tau$*,* $\phi_t(tr) = \phi_t(a_1) \dots \phi_t(a_n)$*.*

Corollary 2.1 *For any IOLTS* $\mathcal{M}$*,* $\phi_t(\mathcal{M})$ *is bijective, i.e.* $Trace(\phi(\mathcal{M})) = \phi_t(Trace(\mathcal{M}))$*.*

3 Results

Following *Rutten*'s works [74, 41], this trace model can be computed. This first requires the existence of a final model in the category $\mathbf{PComp}(H)$. As usual, this terminal model can be obtained under some conditions on the cardinality of the set yielded by the mapping $\mathsf{beh}_\mathcal{C}$ for every component $\mathcal{C} = (S, \alpha) \in \mathbf{PComp}(H)$.

6. ϵ stands for the empty trace.

3.1 Final model

The condition we need to obtain the existence of a final model is the following:

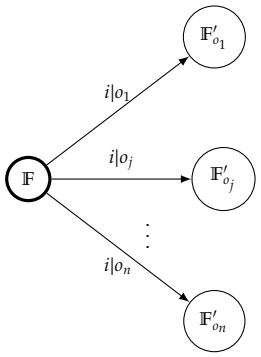
Assumption: *we suppose that for every pre-component $\mathcal{C} = (S, \alpha)$ over a signature $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$, and for every $s \in S$, the cardinality of $\mathsf{beh}_{\mathcal{C}}(s)$ is less than a cardinal κ.*

This assumption allows us then to define a coalgebra (Γ, π) over H and to show that it is final in $\mathbf{PComp}(H)$.

- $\Gamma = \mathcal{P}_{\leq \kappa}(\{\mathcal{F} : \mathsf{In}^\omega \longrightarrow \mathsf{Out}^\omega \mid \mathcal{F} \text{ is causal}\})$
- for every $\mathbb{F} \in \Gamma$ and for every $i \in \mathsf{In}$, $\pi(\mathbb{F})(i) = \eta'^{-1}_{\mathsf{Out} \times \Gamma}(\Pi)$ where:

$$\Pi = \Big\{ (o, \mathbb{F}'_o) \quad | \quad o \in \bigcup_{\mathcal{F} \in \mathbb{F}} (\mathcal{F}(i.x)(0)) \text{ and}, \\ \mathbb{F}'_o = \{\mathcal{F}(i.x)' \mid \mathcal{F}(i.x)(0) = o \text{ and } \mathcal{F} \in \mathbb{F}\}, \\ \text{for } x \in \mathsf{In}^\omega \text{ chosen arbitrarily} \Big\}$$

Let us note here that using $\mathbb{F}'_o = \{\mathcal{F}(i.x)' \mid \mathcal{F}(i.x)(0) = o \text{ and } \mathcal{F} \in \mathbb{F}\}$ instead of $\mathbb{F}' = \{\mathcal{F}(i.x)' \mid \mathcal{F} \in \mathbb{F}\}$ in the definition of (Γ, π) allows us to keep the computational effects carried by the monad T. This is done by linking the output o to the derivative function set $\mathbb{F}'$ *i.e.* the derivative function set is not only linked to the input i but also to the output associated to i. This construction of the set $\mathbb{F}'$ is useful to prove that (Γ, π) is final in $\mathbf{PComp}(H)$.



Theorm 3.1 *Let $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$ be a signature such that for every pre-component $\mathcal{C} = (S, \alpha)$ over H, and for every $s \in S$, $|\mathsf{beh}_{\mathcal{C}}(s)| \leq \kappa$. Then, the coalgebra (Γ, π) is final in $\mathbf{PComp}(H)$.*

Proof *Let (Γ, π) be as stated, and let $\mathcal{C} = (S, \alpha) \in \mathbf{PComp}(H)$ be an arbitrary component. We have to show that there exists a unique homomorphism of components $S \to \Gamma$. For this, let us take the behaviour mapping $\mathsf{beh}_{\mathcal{C}} : S \longrightarrow \Gamma$ (see Definition 2.4) which for every $s \in S$ associates a finite set of transfer functions $\mathbb{F} = \{\mathcal{F} : \mathsf{In}^\omega \longrightarrow \mathsf{Out}^\omega \mid \mathcal{F} \text{ is causal}\} \in \Gamma$. We have to prove that it is the unique homomorphism making the following diagram commute.*

$$\begin{array}{ccc} S \times \mathsf{In} & \xrightarrow{\mathsf{beh}_{\mathcal{C}} \times \mathsf{id}_{\mathsf{In}}} & \Gamma \times \mathsf{In} \\ \downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle \pi} \\ T(\mathsf{Out} \times S) & \xrightarrow[T(\mathsf{id}_{\mathsf{Out}} \times \mathsf{beh}_{\mathcal{C}})]{} & T(\mathsf{Out} \times \Gamma) \end{array}$$

We first prove the commutation.
First of all, it is easy to see that the following properties are satisfied:

$$\forall f, f : S \longrightarrow S', \forall X \in T(S) : T(f)(X) = \eta'^{-1}_{S'} \circ \mathcal{P}(f) \circ \eta'_S(X) \tag{IV.1}$$

$$\begin{gathered} \forall s \in S, i \in \mathsf{In} \ \textit{and} \ \forall (o, s') \in \eta'_{\mathsf{Out}\times S}(\alpha(s)(i)) \\ \mathsf{beh}_{\mathcal{C}}(s') = \mathsf{beh}_{\mathcal{C}}(s)'_o = \{\mathcal{F}(i.x)' \mid \mathcal{F}(i.x)(0) = o \textit{ and } \mathcal{F} \in \mathsf{beh}_{\mathcal{C}}(s)\} \end{gathered} \tag{IV.2}$$

$$\begin{gathered} \forall i \in \mathsf{In}, s \in S \ \textit{and} \ \mathsf{beh}_{\mathcal{C}}(s) \in \Gamma, \\ \eta'_{\mathsf{Out}\times S}(\alpha(s)(i))_{|_1} = \{\mathcal{F}(i.x)(0) \mid \mathcal{F} \in \mathsf{beh}_{\mathcal{C}}(s)\} \end{gathered} \tag{IV.3}$$

Hence, let $s \in S$, $i \in \mathsf{In}$ and $x \in \mathsf{In}^\omega$ be arbitrary. We have to prove that:

$$(T(\mathsf{id}_{\mathsf{Out}} \times \mathsf{beh}_{\mathcal{C}}) \circ \alpha)(s)(i) = (\pi \circ (\mathsf{beh}_{\mathcal{C}} \times \mathsf{id}_{\mathsf{In}}))(s)(i)$$

$$\begin{aligned}
(T(id_{\mathsf{Out}} \ &\times \ \mathsf{beh}_{\mathcal{C}}) \circ \alpha)(s)(i) \\
&= T(\mathsf{id}_{\mathsf{Out}} \times \mathsf{beh}_{\mathcal{C}})(\alpha(s)(i)) \\
&= \eta'^{-1}_{\mathsf{Out}\times\Gamma}(\mathcal{P}(\mathsf{id}_{\mathsf{Out}} \times \mathsf{beh}_{\mathcal{C}})(\eta'_{\mathsf{Out}\times S}(\alpha(s)(i)))) \qquad \textit{Property IV.1} \\
&= \eta'^{-1}_{\mathsf{Out}\times\Gamma}(\{(o, \mathsf{beh}_{\mathcal{C}}(s')) \mid (o, s') \in \eta'_{\mathsf{Out}\times S}(\alpha(s)(i))\}) \qquad \textit{Property IV.2} \\
&= \eta'^{-1}_{\mathsf{Out}\times\Gamma}\Big\{(o, \mathsf{beh}_{\mathcal{C}}(s)'_o) \mid o \in \eta'_{\mathsf{Out}\times S}(\alpha(s)(i))_{|_1} \textit{ and }, \\
&\qquad \mathsf{beh}_{\mathcal{C}}(s)'_o = \{\mathcal{F}(i.x)' \mid \mathcal{F}(i.x)(0) = o \textit{ and } \mathcal{F} \in \mathsf{beh}(s)\}\Big\} \qquad \textit{Property IV.3} \\
&= \eta'^{-1}_{\mathsf{Out}\times\Gamma}\Big\{(o, \mathsf{beh}_{\mathcal{C}}(s)'_o) \mid o \in \mathcal{F} \in \mathsf{beh}_{\mathcal{C}}(s)(\mathcal{F}(i.x)(0)) \textit{ and }, \\
&\qquad \mathsf{beh}_{\mathcal{C}}(s)'_o = \{\mathcal{F}(i.x)' \mid \mathcal{F}(i.x)(0) = o \textit{ and } \mathcal{F} \in \mathsf{beh}_{\mathcal{C}}(s)\}\Big\} \qquad \textit{Definition of } \pi \\
&= \pi((\mathsf{beh}_{\mathcal{C}}(s), i)) \\
&= \pi(\mathsf{beh}_{\mathcal{C}} \times \mathsf{id}_{\mathsf{In}})(s)(i) \\
&= (\pi \circ (\mathsf{beh}_{\mathcal{C}} \times \mathsf{id}_{\mathsf{In}}))(\mathbf{s})(\mathbf{i})
\end{aligned}$$

Next we have to prove uniqueness. In order to prove this last point, we need to prove the following lemma:

Lemma 3.1 *For every component homomorphism $f : S \longrightarrow \Gamma$, for every $x \in \mathsf{In}^\omega$ and for every $s \in S$ we have:*

$$(f(s)(x))' = \{f(s')(x') \mid s' \in \eta'_{\mathsf{Out}\times S}(\alpha(s)(x(0)))_{|_2}\}$$

where x' is the derivative of x.

Proof

$$\begin{aligned}
(f(s)(x))' &= \Big\{(o_1,o_2,\dots,o_k,\dots)' \mid \exists s_0,s_1,\dots,s_k,\dots \in S \\
&\quad \textit{such that } s = s_0,\ \langle o_1,s_1\rangle \in \eta'_{\mathsf{Out}\times S}(\alpha(s_0)(x(0))) \\
&\quad \textit{and } \forall 2 \le j \le k-1, (o_j,s_j) \in \eta'_{\mathsf{Out}\times S}((s_{j-1})(x(j-1))), \\
&\quad \textit{and } o_k \in \eta'_{\mathsf{Out}\times S}(\alpha((s_k)(x(k)))_{|_1})\Big\} \\
&= \Big\{(o_2,\dots,o_k,\dots) \mid \exists s_1,\dots,s_k,\dots \in S \\
&\quad \textit{such that } \ s_1 \in \eta'_{\mathsf{Out}\times S}(\alpha(s_0)(x(0)))_{|_2} \\
&\quad \textit{and } \forall 2 \le j \le k-1, s_j \in \eta'_{\mathsf{Out}\times S}(\alpha((s_{j-1})(x(j-1))))_{|_2}, \\
&\quad \textit{and } o_k \in \eta'_{\mathsf{Out}\times S}(\alpha(s_k)(x(k)))_{|_1}\Big\} \\
&= \{f(s_1)(x') \mid s_1 \in \eta'_{\mathsf{Out}\times S}(\alpha(s_0)(x(0))_{|_2}\} \\
&= \{f(s')(x') \mid s' \in \eta'_{\mathsf{Out}\times S}(\alpha(s)(x(0))_{|_2}\}
\end{aligned}$$

End

Now, let us assume that $g : S \to \Gamma$ is also a homomorphism of components. Let us show that the relation $R \subseteq \mathcal{P}_\kappa(\mathsf{Out}^\omega) \times \mathcal{P}_\kappa(\mathsf{Out}^\omega)$ defined as:

$$R = \{\langle g(s)(x), \mathsf{beh}(s)(x)\rangle \mid s \in S, x \in \mathsf{In}^\omega,\ g(s)(x) = \mathsf{beh}(s)(x)\}$$

is a bisimulation.
It can be shown by coinduction on $x \in \mathsf{In}^\omega$, that for all $s \in S$ we have:

$$g(s)(x) = \mathsf{beh}(s)(x)$$

The initial set outputs of $g(s)(x)$ and $\mathsf{beh}(s)(x)$ agree, since at the initial input $x(0)$ of x, we have:

$$g(s)(x)(0) \;=\; \eta'_{\mathsf{Out}\times S}(\alpha(s)(x(0))_{|_1} \;=\; \mathsf{beh}(s)(x)(0)$$
$$(g(s)(x))' = (g(s)(x(0).x'))' \;=\; \{g(s')(x') \mid s' \in \eta'_{\mathsf{Out}\times S}(\alpha(s)(x(0)))_{|_2}\} \quad \textit{Lemma } 2$$

$$(\mathsf{beh}(s)(x))' = (\mathsf{beh}(s)(x(0).x'))' \;=\; \{\mathsf{beh}(s')(x') \mid s' \in \eta'_{\mathsf{Out}\times S}(\alpha(s)(x(0)))_{|_2}\} \quad \textit{Lemma } 2$$

Hence the function derivatives sets are also R-related, and we conclude that R is a bisimulation.
End

3.2 Minimal component

A final model of the functor $F = T(\mathsf{Out} \times \_)^{\mathsf{In}}$ provides an abstract model of all possible behaviours of its F-coalgebras. Hence, in practice, it cannot be handled as a whole, but we can construct the minimal part of it (minimality refers to the cardinality of the state set) for every state $s \in S$ of a F-coalgebra $\mathcal{C} = (S, \alpha)$. This is done by computing the smallest subcoalgebra in (Γ, π) containing $\mathsf{beh}_{\mathcal{C}}(s)$. More generally, given a subset $\mathbb{F} \in \Gamma$ of causal functions, we can compute the smallest subcoalgebra in (Γ, π), noted $\langle\mathbb{F}\rangle$, containing $\mathbb{F}$. This coalgebra is called the *coalgebra generated by* $\mathbb{F}$ in (Γ, π) in [5].

This construction will be useful to us to define our composition operators (see Chapter V).

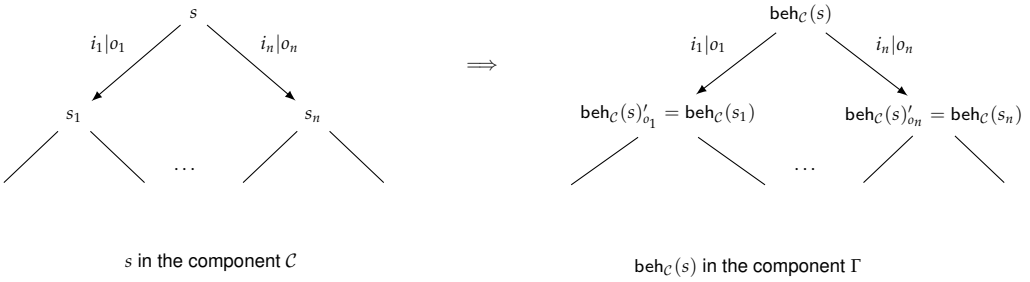
Definition 3.1 ***(Component generated by*** $\mathbb{F}$***)*** *Let* (Γ, π) *be the final model over* $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$. *Let* $\mathbb{F} \in \Gamma$. *The* ***component*** $\langle \mathbb{F} \rangle$ ***generated by*** $\mathbb{F}$ *in* (Γ, π) *is the component* $(\langle \mathbb{F} \rangle, \mathbb{F}, \alpha_{\langle \mathbb{F} \rangle})$ *defined as follows:*

- $\mathbb{F}$ *is the initial state,*
- $\langle \mathbb{F} \rangle$ *is the set of transfer function sets inductively defined as follows:*
  - $\langle \mathbb{F} \rangle^0 = \{\mathbb{F}\}$
  - $\langle \mathbb{F} \rangle^j = \Big\{ \mathbb{G}' \mid \exists \mathbb{G} \in \langle \mathbb{F} \rangle^{j-1}, \exists i \in \mathsf{In}, \exists o \in \mathsf{Out}, o \in \bigcup_{\mathcal{F} \in \mathbb{G}} \mathcal{F}(i.x)(0)$
    $\quad$ *and* $\mathbb{G}' = \{\mathcal{F}(i.x)' \mid \mathcal{F}(i.x)(0) = o$ *and* $\mathcal{F} \in \mathbb{G}\}$,
    $\quad$ *for* $x \in \mathsf{In}^\omega$ *chosen arbitrarily* $\Big\}$

  Hence, $\langle \mathbb{F} \rangle = \bigcup_{j<\omega} \langle \mathbb{F} \rangle^j$
- $\alpha_{\langle \mathbb{F} \rangle} : \langle \mathbb{F} \rangle \times \mathsf{In} \to T(\mathsf{Out} \times \langle \mathbb{F} \rangle)$ *is the mapping which for every* $\mathbb{G} \in \langle \mathbb{F} \rangle$, *and for every input* $i \in \mathsf{In}$ *associates* $\eta'^{-1}_{\mathsf{Out} \times \langle \mathbb{F} \rangle}(\Pi')$ *where* Π' *is the set:*

$$\Pi' = \Big\{ (o, \mathbb{G}'_o) \quad | \quad o \in \bigcup_{\mathcal{F} \in \mathbb{G}} (\mathcal{F}(i.x)(0)) \text{ and}, \quad \mathbb{G}'_o = \{\mathcal{F}(i.x)' \mid \mathcal{F}(i.x)(0) = o \text{ and } \mathcal{F} \in \mathbb{G}\}, \quad \text{for } x \in \mathsf{In}^\omega \text{ chosen arbitrarily} \Big\}$$

It is easy to notice that both components $\mathcal{C} = (S, init, \alpha)$ and $\langle \mathsf{beh}_{\mathcal{C}}(init) \rangle$ share the same trace semantics i.e. $Trace(\mathcal{C}) = Trace(\langle \mathsf{beh}_{\mathcal{C}}(init) \rangle)) = \mathsf{beh}_{\mathcal{C}}(init)$. (see Definition 2.4).



From the finality of (Γ, π), the component $\langle \mathbb{F} \rangle$ generated by $\mathbb{F}$ in (Γ, π) can be built by a repeated computation of derivative sets starting from $\mathbb{F}$. The state of $\langle \mathbb{F} \rangle$ therefore contains all derivative sets of $\mathbb{F}$ and may eventually not be finite. In the following we will be only interested in component $\langle \mathbb{F} \rangle$ with finite state spaces.

Example 3.1 ***(Minimal component)*** *For a better understanding of the definition of a minimal component, we consider an example of binary Mealy machine* $\mathcal{M}$ *modeled by the transition diagram shown on Figure IV.6. This machine* $\mathcal{M}$ *is considered as a component* $\mathcal{M} = (\{s_0, s_1, s_2\}, s_0, \alpha)$ *over the signature* $(\{0,1\} \times \_)^{\{0,1\}}$ *where the transition function:*

$$\alpha : \{s_0, s_1, s_2\} \longrightarrow (\{0,1\} \times \{s_0, s_1, s_2\})^{\{0,1\}}$$

is defined as follows:

$$\begin{cases} \alpha(s_0)(0) = (0, s_2) \\ \alpha(s_0)(1) = (1, s_1) \end{cases} \qquad \begin{cases} \alpha(s_1)(0) = (1, s_1) \\ \alpha(s_1)(1) = (0, s_2) \end{cases} \qquad \begin{cases} \alpha(s_2)(0) = (0, s_2) \\ \alpha(s_2)(1) = (1, s_1) \end{cases}$$

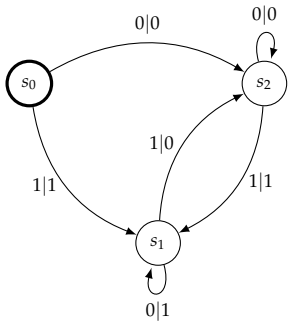


Figure IV.6 – Binary Mealy automaton

It is not difficult to see that applying Definition 2.4 to the initial state s_0 leads to the minimal set of transfer functions $\mathsf{beh}_{\mathcal{M}}(s_0) = \{\mathcal{F}_1\}$ where $\mathcal{F}_1 : \{0,1\}^\omega \longrightarrow \{0,1\}^\omega$ is the transfer function of Example 2.1 i.e. the one defined for every $\sigma \in \{0,1\}^\omega$ and for every $k \in \omega$ by:

$$\mathcal{F}_1(\sigma(k)) = \Big(\sum_{i=0}^{k} \sigma(i)\Big) \bmod 2$$

Now to compute the minimal component $\langle \mathsf{beh}_{\mathcal{M}}(s_0)\rangle$, we need to compute all derivative sets of transfer functions starting from $\mathsf{beh}_{\mathcal{M}}(s_0)$. With a simple computing, we can conclude that the state of $\langle \mathsf{beh}_{\mathcal{M}}(s_0)\rangle$ consists of two states: $\{\mathcal{F}_1\}$ and $\{\mathcal{F}_2\}$ where $\mathcal{F}_2 : \{0,1\}^\omega \longrightarrow \{0,1\}^\omega$ is the transfer function defined for every $\sigma \in \{0,1\}^\omega$ and for every $k \in \omega$ by:

$$\mathcal{F}_2(\sigma(k)) = 1 - \Big(\sum_{i=0}^{k} \sigma(i)\Big) \bmod 2$$

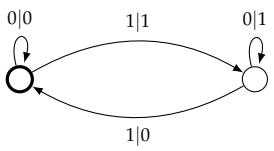
Computing further derivative sets will not yield any new transfer functions sets. Thus, $\langle \mathsf{beh}_{\mathcal{M}}(s_0)\rangle$ is the component $(\{\mathcal{F}_1, \mathcal{F}_2\}, \{\mathcal{F}_1\}, \alpha_{\mathsf{beh}_{\mathcal{M}}(s_0)})$ where:

$$\alpha_{\mathsf{beh}_{\mathcal{M}}(s_0)} : \{\{\mathcal{F}_1\}, \{\mathcal{F}_2\}\} \longrightarrow (\{0,1\} \times \{\{\mathcal{F}_1\}, \{\mathcal{F}_2\}\})^{\{0,1\}}$$

is the transition function defined as follows:

$$\begin{cases} \alpha_{\mathsf{beh}_{\mathcal{M}}(s_0)}(\{\mathcal{F}_1\})(0) = (0, \{\mathcal{F}_1\}) \\ \alpha_{\mathsf{beh}_{\mathcal{M}}(s_0)}(\{\mathcal{F}_1\})(1) = (1, \{\mathcal{F}_2\}) \\ \alpha_{\mathsf{beh}_{\mathcal{M}}(s_0)}(\{\mathcal{F}_2\})(0) = (1, \{\mathcal{F}_2\}) \\ \alpha_{\mathsf{beh}_{\mathcal{M}}(s_0)}(\{\mathcal{F}_2\})(1) = (0, \{\mathcal{F}_1\}) \end{cases}$$

and can be then depicted as:



4 Conclusion

The contribution of this chapter is threefold: first, it shows the effectiveness of *Barbosa*'s coalgebraic definition of components in unifying in a single framework a wide variety of state-based formalisms such as Mealy automata [33, 34], Labeled Transition Systems [35, 36] and Input-Output Labeled Transition Systems [39, 40] by using a suitable choice of computation structures introduced by the monad T [16, 15]. Second, this way of modeling the behaviour of components allows us, following *Rutten*'s works [41, 74], to define a trace model over components by causal transfer functions [77]. Such functions are dataflow transformations of the form: $y = \mathcal{F}(x, s, t)$ where x, y and s are respectively the input, output and state of the component under consideration, and t stands for discrete time. This representation of system behaviour forms the first step towards a unified framework that will capture not only different usual computations, but also time heterogeneity (i.e. both discrete and continuous times). Indeed, in this thesis, we restrict ourselves to discrete time. However, there are other current works extending our framework to be able to take into account continuous time using non-standard analysis [81]. Third, defining a trace model from causal functions (which is the main contribution of this chapter) allows us to show the existence of a final coalgebra in the category of coalgebras over a signature $T(\mathsf{Out} \times \_)^{\mathsf{In}}$ under some sufficient conditions on the monad T. Final coalgebras are indeed important because their existence is the key of *co-induction*, a powerful reasoning principle in coalgebraic theory. Such a final minimal component model will be the cornerstone of defining how components are combined to define larger components in the following chapter.

Chapter V

Integration of components

1 Basic integration 90
1.1 Cartesian product 90
1.2 Feedback 90
2 Complex operators 99
2.1 Sequential composition 100
2.2 Double sequential composition 102
2.3 Synchronous product 104
2.4 Concurrent composition 105
2.5 Synchronous parallel composition 106
3 Systems and compositionality 108
3.1 Systems 108
3.2 Examples 109
3.3 Compositionality 118
4 Related works 123
5 Conclusion 125

So far in this part we have seen that the easiest way to model complex systems is to describe them as compositions of simpler systems, being considered as coalgebraic components. In this chapter, we explain how to define a larger component by composition of multiple components using integration operators. Many different integration operators have been defined and studied in the literature such as sequential composition, double sequential composition, synchronous product, concurrent composition or synchronous parallel composition. Here, we will show that most of them can be obtained by a composition of two basic integration operators, namely: *cartesian product* and *feedback*. In Section 1, we will then define these two basic operators for building larger components from simpler ones. In Section 2, we will define more complex operators by composition of our basic integration operators. Finally, in Section 3, we will define how systems can be built over these complex operators defined in Section 2 and will give some concrete system examples.

1 Basic integration

1.1 Cartesian product

The *cartesian product* is a composition where both components are executed simultaneously when triggered by a pair of input values (see Figure V.1).

Definition 1.1 *(**Cartesian product** $\otimes$) Let $H_1 = T(\mathsf{Out}_1 \times \_)^{\mathsf{In}_1}$ and $H_2 = T(\mathsf{Out}_2 \times \_)^{\mathsf{In}_2}$ be two signatures. Let $H = T((\mathsf{Out}_1 \times \mathsf{Out}_2) \times \_)^{(\mathsf{In}_1 \times \mathsf{In}_2)}$ be the signature resulting from the product of H_1 and H_2. Let us define the **cartesian integration functor**:*

$$\begin{array}{ccccc} \otimes : & & \mathbf{Comp}(H_1) \times \mathbf{Comp}(H_2) & \longrightarrow & \mathbf{Comp}(H) \\ & (\xrightarrow{\mathsf{In}_1} \boxed{(S_1, \alpha_1)} \xrightarrow{\mathsf{Out}_1} , \xrightarrow{\mathsf{In}_2} \boxed{(S_2, \alpha_2)} \xrightarrow{\mathsf{Out}_2}) & \mapsto & \xrightarrow{\mathsf{In}_1 \times \mathsf{In}_2} \boxed{(S, \alpha)} \xrightarrow{\mathsf{Out}_1 \times \mathsf{Out}_2} \end{array}$$

as follows: $\forall \mathcal{C}_1 = (S_1, init_1, \alpha_1) \in \mathbf{Comp}(H_1), \forall \mathcal{C}_2 = (S_2, init_2, \alpha_2) \in \mathbf{Comp}(H_2)$,

$$\otimes((\mathcal{C}_1, \mathcal{C}_2)) = (S, init, \alpha)$$

where:

- $S = S_1 \times S_2$ *is the set of states,*
- $init = (init_1, init_2)$ *is the initial state,*
- $\alpha : S \times (\mathsf{In}_1 \times \mathsf{In}_2) \longrightarrow T((\mathsf{Out}_1 \times \mathsf{Out}_2) \times S)$ *is the mapping defined as follows:* $\forall i = (i_1, i_2) \in \mathsf{In}_1 \times \mathsf{In}_2$ *and* $(s_1, s_2) \in S$*:*

$$\alpha((s_1, s_2))(i) = \eta'^{-1}_{(\mathsf{Out}_1 \times \mathsf{Out}_2) \times S} \{ ((o_1, o_2), (s_1', s_2')) | (o_1, s_1') \in \eta'_{\mathsf{Out}_1 \times S_1}(\alpha_1(s_1)(i_1)) \text{ and } (o_2, s_2') \in \eta'_{\mathsf{Out}_2 \times S_2}(\alpha_2(s_2)(i_2)) \}$$

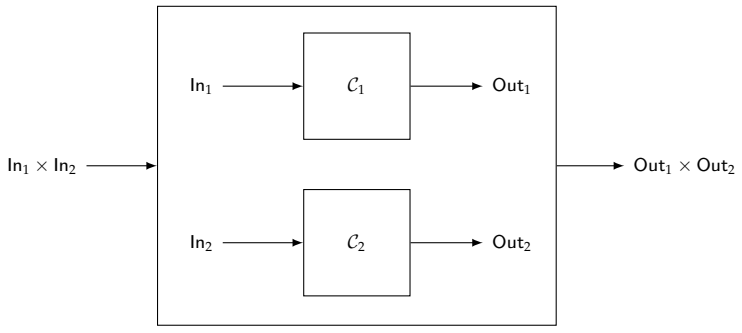


Figure V.1 – Cartesian product

1.2 Feedback

A component with *feedback* has directed cycles, where an output from a component is fed back to affect an input of the same component (see Figure V.2). That means the output of a component in any feedback composition depends on an input value that in turn depends on its own output value. The feedback operator is then a composition where some outputs of a component

are linked to its inputs i.e. some outputs can be fed back as inputs. In order to obtain a model which fits our component definition, we need to take into account the computational effects of the monad T. This monad impacts both the evolution of component states and the observation of its outputs. Therefore, the feedback link between outputs and inputs carries part of the structure imposed by T to the inputs. For instance, with the monad built on $\mathcal{P}$ for modeling non-determinism, the feedback may bring non-determinism to the inputs of the component.

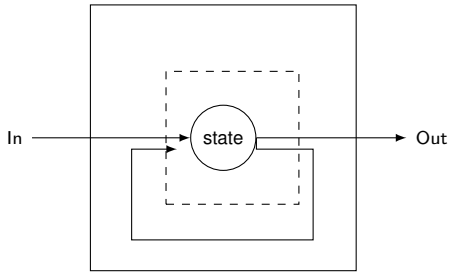


Figure V.2 – Illustration of a system with feedback

We introduce feedback interfaces for defining correspondences between outputs and inputs of components. A feedback interface also allows us to keep only the inputs and the outputs that are not involved in feedback thanks to component-wise projections π_i and π_o:

Definition 1.2 *(**Feedback interface**) Let $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$ be a signature. A **feedback interface** over H is a triplet $\mathcal{I} = (f, \pi_i, \pi_o)$ where:*

– $f : \mathsf{In} \times \mathsf{Out} \longrightarrow \mathsf{In}$ *is a function such that:*

$$\forall (i,o) \in \mathsf{In} \times \mathsf{Out}, f(f(i,o),o) = f(i,o)$$

– $\pi_i : \mathsf{In} \longrightarrow \mathsf{In}'$ *and* $\pi_o : \mathsf{Out} \longrightarrow \mathsf{Out}'$ *are surjective mappings*[1] *such that:*

$$\forall (i,o) \in \mathsf{In} \times \mathsf{Out}, \pi_i(i) = \pi_i(f((i,o)))$$

The mapping f allows us to specify how components are linked and which parts of their interfaces are involved in the composition process. Both mappings π_i and π_o can be thought of as extensions of the hiding connective found in process calculi [82]. Thereby, the feedback interface enables encapsulation by making the internal interactions made in the scope of the component invisible. Such an encapsulation helps to separate both the internal behaviour and the interaction of a component from the external interaction with the global system, and thus deals with the interaction between components independently of the behaviour of individual components.

Two kinds of feedback operators are usually distinguished: *relaxed* feedback and *synchronous* feedback. The first kind means that in a reaction, the output is not simultaneous with the input. This relaxed feedback composition depends on the previous output and the current input. The second kind means that the reaction of a system takes no observable time [83]. Systems produce their outputs synchronously with their inputs. More precisely, at some reaction r, the output of system $\mathcal{S}$ in r must be available to its inputs in the same reaction r.

Definition 1.3 *(**Relaxed feedback** $\hookleftarrow$) Let $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$ be a signature and $\mathcal{I} = (f, \pi_i, \pi_o)$ be a feedback interface over H. Let us note $H' = T(\mathsf{Out}' \times \_)^{\mathsf{In}'}$. Let $\mathcal{C} = (S, s_0, \alpha)$ be a component over H. Let us define for every $x \in \mathsf{In}^\omega$, the set Θ_x whose elements are couples $(\bar{x}, y_{\bar{x}}) \in \mathsf{In}^\omega \times \mathsf{Out}^\omega$ inductively defined from an infinite sequence of states $(s_0, s_1, \ldots, s_k, \ldots)$ of S as follows:*

1. *i.e* component-wise projections

- $\bar{x}(0) = x(0)$ *and* $y_{\bar{x}}(0) \in \eta'_{\mathsf{Out}\times S}(\alpha(s_0)(x(0)))_{|_1}$
- $\forall n, 0 < n < \omega$,
  - $\bar{x}(n) = f(x(n), y_{\bar{x}}(n-1))$
  - $y_{\bar{x}}(n) \in \eta'_{\mathsf{Out}\times S}(\alpha(s_n)(\bar{x}(n)))_{|_1}$
  - *and* $s_n \in \eta'_{\mathsf{Out}\times S}(\alpha(s_{n-1})(\bar{x}(n-1)))_{|_2}$

Then, ***the operation of relaxed feedback over*** $\mathcal{I}$, $\hookleftarrow_{\mathcal{I}}$: $\mathbf{Comp}(H) \longrightarrow \mathbf{Comp}(H')$ *associates to every component* $\mathcal{C} = (S, s_0, \alpha)$ *over* H, *the component* $(\langle \mathbb{F} \rangle, \mathbb{F}, \alpha_{\langle \mathbb{F} \rangle})$ *over* H' *where* $\mathbb{F}$ *is the set of transfer functions* $\mathcal{F} : \mathsf{In}'^{\omega} \longrightarrow \mathsf{Out}'^{\omega}$, *each one defined by* $\mathcal{F}(x') = y'$ *where there exists* $x \in \mathsf{In}^{\omega}$ *such that there exists* $(\bar{x}, y_{\bar{x}}) \in \Theta_x$ *satisfying*

$$\forall i < \omega, x'(i) = \pi_i(\bar{x}(i)) \text{ and } y'(i) = \pi_o(y_{\bar{x}}(i))$$

Definition 1.3 calls for some comments. We want to build a component that hides the relaxed feedback of a component $\mathcal{C}$. As one can see in Figure V.3, the relaxed feedback component $\hookleftarrow_{\mathcal{I}}(\mathcal{C})$ is given as a set of transfer functions, each one mapping an infinite sequence of inputs $x' \in \mathsf{In}'^{\omega}$ to an infinite sequence of outputs $y' \in \mathsf{Out}'^{\omega}$. The outputs are then hidden from any state s

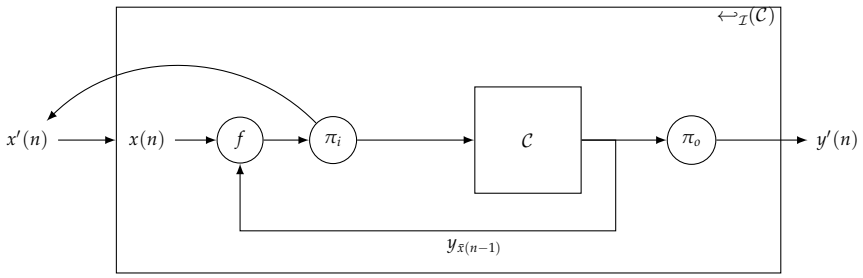


Figure V.3 – Relaxed feedback composite: $\hookleftarrow_{\mathcal{I}}(\mathcal{C})$

that are fed back as inputs to the successor of s. The result is a component with input and output sets In' and Out' respectively. This is done by means of the feedback interface $\mathcal{I} = (f, \pi_i, \pi_o)$. Let us suppose that the current state of $\mathcal{C}$ at the nth reaction is $s_n \in S$ and the current external input is $x(n) \in \mathsf{In}$, then let us compute both new input $x'(n) \in \mathsf{In}'$ and output $y'(n) \in \mathsf{Out}'$ when $\mathcal{C}$ is triggered by $x(n)$. First, by f, we compute the input $\bar{x}(n) = f(x(n), y_{\bar{x}(n-1)})$. Then, $\bar{x}(n)$ becomes the new input of $\mathcal{C}$. Indeed, component $\mathcal{C}$ reacts by updating its state to s_{n+1} and producing an output $y_{\bar{x}(n)}$. In this way, the output of $\mathcal{C}$ at the nth reaction is given by relying on the previous output $y_{\bar{x}(n-1)}$ and the current input $x(n)$. Second, by means of π_i and π_o, we hide both input and output involved in the feedback, and then produce the input $x'(n) = \pi_i(\bar{x}(n))$ and the output $y'(n) = \pi_o(\bar{y}(n))$ of the relaxed feedback component $\hookleftarrow_{\mathcal{I}}(\mathcal{C})$.

Proposition 1.1 $\hookleftarrow_{\mathcal{I}}$: $\mathbf{Comp}(H) \longrightarrow \mathbf{Comp}(H')$ *is a functor.*

Proof *It only remains for us to make a correspondence between homomorphisms in* $\mathbf{Comp}(H)$ *and homomorphisms in* $\mathbf{Comp}(H')$. *Let* $f : \mathcal{C}_1 \longrightarrow \mathcal{C}_2$ *be an homomorphism in* $\mathbf{Comp}(H)$. *Then, let us define* $\hookleftarrow_{\mathcal{I}}(f) : \hookleftarrow_{\mathcal{I}}(\mathcal{C}_1) \longrightarrow \hookleftarrow_{\mathcal{I}}(\mathcal{C}_2)$ *where* $\hookleftarrow_{\mathcal{I}}(\mathcal{C}_i) = (\langle \mathbb{F}_i \rangle, \mathbb{F}_i, \alpha_{\langle \mathbb{F}_i \rangle})$ *for* $i = 1, 2$ *as follows:*

- $\hookleftarrow_{\mathcal{I}}(f)(\mathbb{F}_1) = \mathbb{F}_2$
- *for every* $j, 0 < j < \omega$, *for every* $\mathbb{G}' \in \langle \mathbb{F}_1 \rangle^j$, *we know by definition that there exists*

$$\mathbb{G} \in \langle \mathbb{F}_1 \rangle^{j-1}, i \in \mathsf{In} \text{ and } o \in \mathsf{Out} \text{ such that :}$$

– $o \in \bigcup_{\mathcal{F}\in\mathbb{G}} (\mathcal{F}(i.x)(0))$
– $\mathbb{G}' = \{\mathcal{F}(i.x)' \mid \mathcal{F}(i.x)(0) = o \text{ and } \mathcal{F} \in \mathbb{G}\}$
for $x \in \mathsf{In}^\omega$ chosen arbitrarily. It is sufficient to write down

$$\leftrightarrow_{\mathcal{I}}(f)(\mathbb{G}') = \left\{\mathcal{F}'(i.x)' \mid \mathcal{F}'(i.x)(0) = o \text{ and } \mathcal{F}' \in \leftrightarrow_{\mathcal{I}}(f)(\mathbb{G})\right\}$$

f being a morphism on coalgebras, we can easily show that $\leftrightarrow_{\mathcal{I}}(f)(\mathbb{G}')$ is nonempty.

Let us finish by showing that $\leftrightarrow_{\mathcal{I}}(f)$ preserves identities and compositions.
For identities, let $\mathcal{C} \in \mathbf{Comp}(H)$, $\leftrightarrow_{\mathcal{I}}(\mathcal{C}) = \langle\mathbb{F}\rangle$, and let us prove by induction on the structure of $\mathbb{F}$ that $\leftrightarrow_{\mathcal{I}}(\mathsf{id}_{\mathcal{C}}) = \mathsf{id}_{\leftrightarrow_{\mathcal{I}}(\mathcal{C})}$.

– ***Basic Step:*** *By definition of $\leftrightarrow_{\mathcal{I}}(\mathsf{id}_{\mathcal{C}})$, one has $\leftrightarrow_{\mathcal{I}}(\mathsf{id}_{\mathcal{C}})(\mathbb{F}) = \mathbb{F} = \mathsf{id}_{\leftrightarrow_{\mathcal{I}}(\mathcal{C})}(\mathbb{F})$*
– ***Induction Step:*** *let $\mathbb{G}' \in \langle\mathbb{F}\rangle^{j+1}$. We know by definition of $\mathbb{G}'$ that there exists $\mathbb{G} \in \langle\mathbb{F}\rangle^{j}$, $i \in \mathsf{In}$ and $o \in \mathsf{Out}$ such that $o \in \bigcup_{\mathcal{F}\in\mathbb{G}} (\mathcal{F}(i.x)(0))$ and $\mathbb{G}' = \{\mathcal{F}(i.x)' \mid \mathcal{F}(i.x)(0) = o \text{ and } \mathcal{F} \in \mathbb{G}\}$ for $x \in \mathsf{In}^\omega$ chosen arbitrarily. Then, by definition of $\leftrightarrow_{\mathcal{I}}(\mathsf{id}_{\mathcal{C}})$ one has:*

$$\begin{aligned}
\leftrightarrow_{\mathcal{I}}(\mathsf{id}_{\mathcal{C}})(\mathbb{G}') &= \left\{\mathcal{F}'(i.x)' \mid \mathcal{F}'(i.x)(0) = o \text{ and } \mathcal{F}' \in \leftrightarrow_{\mathcal{I}}(\mathsf{id}_{\mathcal{C}})(\mathbb{G})\right\} \\
&\quad \textit{by induction hypothesis} \\
&= \left\{\mathcal{F}'(i.x)' \mid \mathcal{F}'(i.x)(0) = o \text{ and } \mathcal{F}' \in \mathsf{id}_{\leftrightarrow_{\mathcal{I}}(\mathcal{C})}(\mathbb{G})\right\} \\
&\quad \textit{by definition of } \mathsf{id}_{\leftrightarrow_{\mathcal{I}}(\mathcal{C})} \\
&= \left\{\mathcal{F}'(i.x)' \mid \mathcal{F}'(i.x)(0) = o \text{ and } \mathcal{F}' \in \mathbb{G}\right\} \\
&\quad \textit{by hypothesis} \\
&= \mathbb{G}' \\
&= \mathsf{id}_{\leftrightarrow_{\mathcal{I}}(\mathcal{C})}(\mathbb{G}')
\end{aligned}$$

For preservation of composition. Let $f_1 : \mathcal{C}_1 \longrightarrow \mathcal{C}_2$ and $f_2 : \mathcal{C}_2 \longrightarrow \mathcal{C}_3$ be two homomorphisms in $\mathbf{Comp}(H)$. Let $\leftrightarrow_{\mathcal{I}}(f_1) : \langle\mathbb{F}_1\rangle \longrightarrow \langle\mathbb{F}_2\rangle$ and $\leftrightarrow_{\mathcal{I}}(f_2) : \langle\mathbb{F}_2\rangle \longrightarrow \langle\mathbb{F}_3\rangle$ their associated homomorphisms in $\mathbf{Comp}(H')$ where $\leftrightarrow_{\mathcal{I}}(\mathcal{C}_1) = \langle\mathbb{F}_1\rangle$, $\leftrightarrow_{\mathcal{I}}(\mathcal{C}_2) = \langle\mathbb{F}_2\rangle$ and $\leftrightarrow_{\mathcal{I}}(\mathcal{C}_3) = \langle\mathbb{F}_3\rangle$.
Let us then prove by induction on the structure of $\mathbb{F}$ that $\leftrightarrow_{\mathcal{I}}(f_2 \circ f_1) = \leftrightarrow_{\mathcal{I}}(f_2) \circ \leftrightarrow_{\mathcal{I}}(f_1)$.

– ***Basic Step:*** *By definition of $\leftrightarrow_{\mathcal{I}}(f_2 \circ f_1)$, one has*

$$\begin{aligned}
\leftrightarrow_{\mathcal{I}}(f_2 \circ f_1)(\mathbb{F}_1) &= \mathbb{F}_3 \qquad \textit{by definition of } \leftrightarrow_{\mathcal{I}}(f_2) \\
&= \leftrightarrow_{\mathcal{I}}(f_2)(\mathbb{F}_2) \qquad \textit{by definition of } \leftrightarrow_{\mathcal{I}}(f_1) \\
&= \leftrightarrow_{\mathcal{I}}(f_2)(\leftrightarrow_{\mathcal{I}}(f_1)(\mathbb{F}_1)) \\
&= \leftrightarrow_{\mathcal{I}}(f_2) \circ \leftrightarrow_{\mathcal{I}}(f_1)(\mathbb{F}_1)
\end{aligned}$$

– ***Induction Step:*** *let $\mathbb{G}'_1 \in \langle\mathbb{F}_1\rangle^{j+1}$. We know by definition of $\mathbb{G}'_1$ that there exists $\mathbb{G}_1 \in \langle\mathbb{F}_1\rangle^{j}$, $i \in \mathsf{In}$ and $o \in \mathsf{Out}$ such that $o \in \bigcup_{\mathcal{F}_1\in\mathbb{G}_1} (\mathcal{F}_1(i.x)(0))$ and $\mathbb{G}'_1 = \{\mathcal{F}_1(i.x)' \mid \mathcal{F}_1(i.x)(0) = o \text{ and } \mathcal{F}_1 \in \mathbb{G}_1\}$ for $x \in \mathsf{In}^\omega$ chosen arbitrarily. By definition of $\leftrightarrow_{\mathcal{I}}(f_1)$, we also know that $\leftrightarrow_{\mathcal{I}}(f_1)(\mathbb{G}'_1) = \left\{\mathcal{F}'_1(i.x)' \mid \mathcal{F}'_1(i.x)(0) = o \text{ and } \mathcal{F}'_1 \in \leftrightarrow_{\mathcal{I}}(f_1)(\mathbb{G}_1)\right\}$.*

Let us denote by the set $\left\{\mathcal{F}'_1(i_1.x_1)' \mid \mathcal{F}'_1(i_1.x_1)(0) = o_1 \text{ and } \mathcal{F}'_1 \in \leftrightarrow_{\mathcal{I}}(f_1)(\mathbb{G}_1)\right\}$ by $\mathbb{G}'_2$. This set belongs to $\langle\mathbb{F}_2\rangle$. Then, we know by definition of $\mathbb{G}'_2$ that $\exists\mathbb{G}_2 \in \langle\mathbb{F}_2\rangle$ such that $\mathbb{G}_2 = \leftrightarrow_{\mathcal{I}}(f_1)(\mathbb{G}_1)$,

$o \in \bigcup_{\mathcal{F}_2 \in \mathbb{G}_2} (\mathcal{F}_2(i.x)(0))$ *and* $\mathbb{G}'_2 = \{\mathcal{F}_2(i.x)' \mid \mathcal{F}_1(i.x)(0) = o$ *and* $\mathcal{F}_2 \in \mathbb{G}_2\}$. *By definition of* $\hookleftarrow_{\mathcal{I}}(f_2)$, *we know that* $\hookleftarrow_{\mathcal{I}}(f_2)(\mathbb{G}'_2) = \{\mathcal{F}'_2(i.x)' \mid \mathcal{F}'_2(i.x)(0) = o$ *and* $\mathcal{F}'_2 \in \hookleftarrow_{\mathcal{I}}(f_2)(\mathbb{G}_2)\}$.

Now, we have that

$$\begin{aligned}
\hookleftarrow_{\mathcal{I}}(f_2) \circ \hookleftarrow_{\mathcal{I}}(f_1)(\mathbb{G}'_1) &= \hookleftarrow_{\mathcal{I}}(f_2)(\hookleftarrow_{\mathcal{I}}(f_1)(\mathbb{G}'_1)) \\
&= \hookleftarrow_{\mathcal{I}}(f_2)(\mathbb{G}'_2) \\
&= \{\mathcal{F}'_2(i.x)' \mid \mathcal{F}'_2(i.x)(0) = o \text{ and } \mathcal{F}'_2 \in \hookleftarrow_{\mathcal{I}}(f_2)(\mathbb{G}_2)\} \\
&= \{\mathcal{F}'_2(i.x)' \mid \mathcal{F}'_2(i.x)(0) = o \text{ and } \mathcal{F}'_2 \in \hookleftarrow_{\mathcal{I}}(f_2)(\hookleftarrow_{\mathcal{I}}(f_1)(\mathbb{G}_1))\} \\
&= \{\mathcal{F}'_2(i.x)' \mid \mathcal{F}'_2(i.x)(0) = o \text{ and } \mathcal{F}'_2 \in \hookleftarrow_{\mathcal{I}}(f_2) \circ \hookleftarrow_{\mathcal{I}}(f_1)(\mathbb{G}_1)\} \\
&\quad \textit{by induction hypothesis} \\
&= \{\mathcal{F}'_2(i.x)' \mid \mathcal{F}'_2(i.x)(0) = o \text{ and } \mathcal{F}'_2 \in \hookleftarrow_{\mathcal{I}}(f_2 \circ f_1)(\mathbb{G}_1)\} \\
&= \hookleftarrow_{\mathcal{I}}(f_2 \circ f_1)(\mathbb{G}'_1)
\end{aligned}$$

End

Example 1.1 ***(Syracuse's sequence)***
Syracuse's sequence *is a finite or infinite sequence of integers* $n_0, n_1, n_2, \ldots, n_i, \ldots$ *generated*[2] *as follows: for any* $i \in \mathbb{N}^*$

$$n_{i+1} \mapsto \begin{cases} n_i/2 & if \qquad n_i \equiv 0 \pmod 2 \\ 3n_i + 1 & otherwise \end{cases}$$

This sequence starts with any positive integer n, and produces at each step either half of itself (i.e. $n/2$*) if it is even or three times itself plus one (i.e.* $3n + 1$*) if it is odd. We can easily see that any output* n_i *produced at the* i^{th} *step is linked to the input at the* $(i+1)^{th}$ *i.e.* n_i *will be considered as the input at the next step. Then, there is a relaxed feedback. Figure V.4 shows the component that can be considered as a structural model of the Syracuse sequence. When a new positive integer n is available, it is directly available as an input of the component C that produces it as output. If n is even, n is sent to the component* DivideBy2 *which produces as an output* $(n/2)$. *If n is odd, n is sent to the component* MultipleBy3Plus1 *which produces as an output* $(3n + 1)$. *Both outputs of* DivideBy2 *and* MultipleBy3Plus1 *are linked to the component* Δ *which is considered as a unit delay allowing time to pass.*
This sequence can be modeled in our framework as a component $\mathcal{S} = (S, init, \alpha)$ *over the functor*

$$(\mathbb{N}^* \times \_)^{(\{\perp\} \times \mathbb{N}^*) \cup \mathbb{N}^*}$$

where:

- *init is the initial state;*
- $S = \{s_i \mid i \in \mathbb{N}^*\} \cup \{init\}$ *is the set of states;*
- $\alpha : S \times ((\{\perp\} \times \mathbb{N}^*)\} \cup \mathbb{N}^*) \longrightarrow \mathbb{N}^* \times S$ *is defined as follows:*

$$\alpha(init)(k) = (k, s_k) \text{ for every } k \in \mathbb{N}^*$$

and

$$\alpha(s_k)((\perp, k)) = \begin{cases} (k/2, s_{k/2}) & if \qquad k \equiv 0 \pmod 2 \\ (3k+1, s_{3k+1}) & otherwise \end{cases}$$

2. $\mathbb{N}^* = \mathbb{N} \setminus \{0\}$

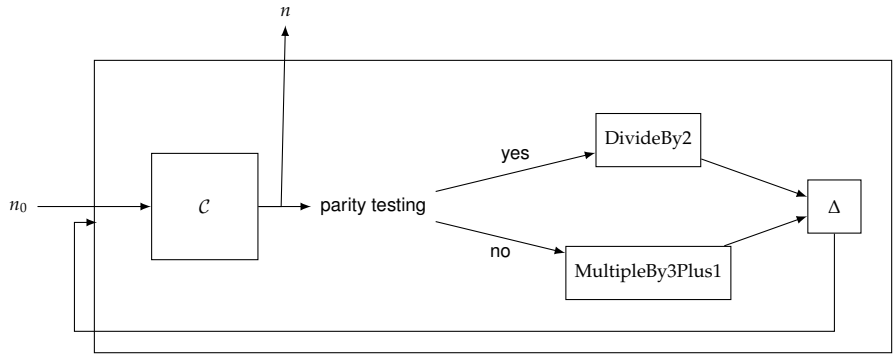


Figure V.4 – Syracuse's sequence component

Thus, the inputs are either:

- *of the form i where i stands for the initial input received from the environment (i.e. the integer for which we compute its associated Syracuse sequence). For instance,* $\alpha(init)(10) = (10, s_{10})$ *states that the component starts running by receiving from the environment an input value* 10 *and goes to state* s_{10} *while producing the output value* 10*.*
- *or pair of input values* $(\perp, i)$ *where* $\perp$ *is the external input value received from the environment*[3] *and i is the input value feeding back from the output. For instance,* $\alpha(s_{10})((\perp, 10)) = (5, s_5)$ *expresses that the component receives no value from the environment and the input value* 10 *feeding back from the output and goes to state* s_5 *while producing the output value* 5*.*

Starting with the initial state init, one gets for example the following execution:

$$init \xrightarrow{3|3} s_3 \xrightarrow{(\perp,3)|10} s_{10} \xrightarrow{(\perp,10)|5} s_5 \xrightarrow{(\perp,5)|16} s_{16} \xrightarrow{(\perp,16)|8} s_8 \xrightarrow{(\perp,8)|4} s_4 \xrightarrow{(\perp,4)|2} s_2 \xrightarrow{(\perp,2)|1} s_1 \quad \text{(V.1)}$$

Let us now compute the Syracuse*'s feedback component. We need to compute a transfer function that takes an input value* n_0 *and computes its associated* Syracuse *sequence. For this, we have to hide both the input and output values involved in the feedback. Taking Sequence V.1, one needs to hide the output values* $3, 10, 5, 16, 8, 4$ *and* 2 *in* $(\perp, 3), (\perp, 10), (\perp, 5), (\perp, 16), (\perp, 8), (\perp, 4), (\perp, 2)$ *that are fed back as inputs. This can be obtained by applying the relaxed feedback operator* $\hookleftarrow_{\mathcal{I}}$ *on* $\mathcal{S}$ *where* $\mathcal{I} = (f, \pi_i, \pi_o)$ *is the feedback interface defined as follows:*

- $f : \{\perp\} \times \mathbb{N}^* \longrightarrow \{\perp\} \times \mathbb{N}^*$ *is the identity on* $\{\perp\} \times \mathbb{N}^*$*;*
- π_i *is the function defined as follows:*

$$\begin{array}{rlcl} \pi_i : & \mathbb{N}^* \cup (\{\perp\}, \mathbb{N}^*) & \rightarrow & \mathbb{N}^* \cup \{\perp\} \\ & n & \mapsto & n \\ & (\perp, n) & \mapsto & \perp \end{array}$$

- $\pi_o : \mathbb{N}^* \longrightarrow \mathbb{N}^*$ *is the identity function on* $\mathbb{N}^*$*.*

Thus, the input-output feedback sequence computed from Sequence V.1 is the following:

$$\langle 3|3, \perp|10, \perp|10, \perp|5, \perp|16, \perp|8, \perp|4, \perp|2, \perp|1 \rangle$$

Hence, it becomes clear that $\hookleftarrow_{\mathcal{I}} (\mathcal{S})$ *is equal to the component* $\langle \mathbb{F} \rangle$ *with* $\mathbb{F}$ *as the set containing the unique transfer function* $\mathcal{F} : (\mathbb{N}^* \cup \{\perp\})^\omega \longrightarrow \mathbb{N}^{*\omega}$ *defined for every input sequence* $\sigma \in$

3. We require that the environment provides an artificial input $\perp$ to make the component react.

$(\mathbb{N}^* \cup \{\perp\})^\omega$ *and for every* $k \in \omega$ *as follows:*

$$\mathcal{F}(\sigma)(k) = \begin{cases} \sigma(k) & \text{if} \quad \sigma(k) \neq \perp \\ \begin{cases} (\mathcal{F}(\sigma)(k-1))/2 & \text{if} \quad \mathcal{F}(\sigma)(k-1) \text{ is even} \\ (3 \times \mathcal{F}(\sigma)(k-1)) + 1 & \text{if} \quad \mathcal{F}(\sigma)(k-1) \text{ is odd} \end{cases} & \text{otherwise} \end{cases}$$

The *synchronous feedback* is more difficult to define because it requires the existence of an instantaneous fixpoint (i.e. defined at the same time and not deferred by one unit). This gives rise to the notion of *well-formed feedback composition*.

Definition 1.4 *(**Well-formed feedback composition**) Let* $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$ *be a signature. Let* $\mathcal{C}$ *be a component over* H *and* $\mathcal{I} = (f, \pi_i, \pi_o)$ *be a feedback interface over* H*. We say that the* ***synchronous feedback composition*** $\mathcal{C}$ ***over*** $\mathcal{I}$ ***is well-formed*** *if, and only if for every state* $s \in S$ *and every* $x \in \mathsf{In}^\omega$*:*

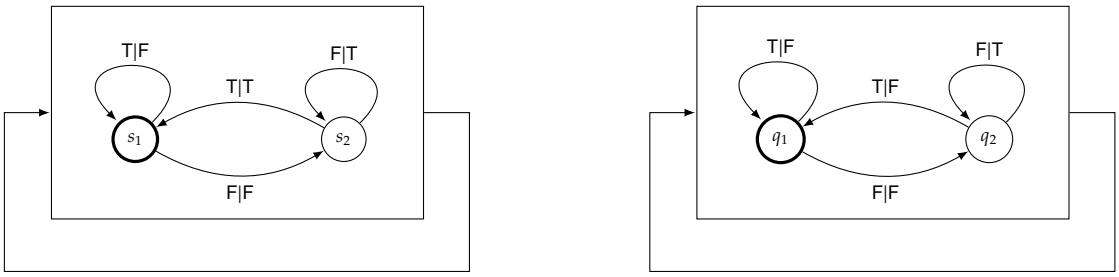
$$\text{there exists } y \in \mathsf{Out}^\omega \text{ such that for every } n < \omega,\ y(n) \in \eta'_{\mathsf{Out} \times S}(\alpha(s)(f(x(n), y(n))))_{|_1}$$

We illustrate the last definition with the following example:

Example 1.2 *(**Well-formed composition**) Consider two components* $\mathcal{C}_1 = (\{s_1, s_2\}, s_1, \alpha_1)$ *and* $\mathcal{C}_2 = (\{q_1, q_2\}, q_1, \alpha_2)$ *over the signature*

$$(\{T, F\} \times \_)^{\{T,F\}}$$

with the transition function α_1 *(respectively* α_2*) graphically drawn on the left side (respectively on the right side) of Figure V.5. There is then a feedback composition for* $\mathcal{C}_1$ *and a feedback composition for* $\mathcal{C}_2$



(a) Well-formed feedback composition of $\mathcal{C}_1$

(b) Ill-formed feedback composition of $\mathcal{C}_2$

Figure V.5 – Examples of feedback composition

because both outputs T *and* F *are fed back as inputs of* $\mathcal{C}_1$ *and* $\mathcal{C}_2$*. These compositions are considered to have a zero-delay feedback loop. Let us consider the component* $f : \mathsf{In} \times \mathsf{Out} \longrightarrow \mathsf{In}$ *of the interface feedback* $\mathcal{I} = (f, \pi_i, \pi_o)$ *as the "and" logic operator. More formally,* $f : \{T, F\} \times \{T, F\} \longrightarrow \{T, F\}$ *is defined as follows:*

$$f((i, o)) = \begin{cases} F & \text{if} \quad i = F \quad \text{or} \quad o = F \\ T & \text{otherwise} \end{cases}$$

The composition of the component $\mathcal{C}_1$ *is well-formed. This is due to the fact that for every state* $s \in \{s_1, s_2\}$ *and for every input sequence* $x \in \{T, F\}^\omega$ *there exists an output* $y \in \{T, F\}^\omega$ *such that for every* $n, n < \omega,\ y(n) \in \alpha_1(s)(f(x(n), y(n)))$*. More precisely*[4]*, one has:*

4. We deliberately used $\alpha_1(s)(f(x(n), y(n)))$ instead of $\eta'(\alpha_1(s)(f(x(n), y(n))))_{|_1}$ for sake of simplicity because η' considered here stands for the identify.

- $F \in \alpha(s_1)(f(F,F))$
- $F \in \alpha(s_1)(f(T,F))$
- $T \in \alpha(s_2)(f(F,T))$
- $T \in \alpha(s_2)(f(T,T))$

On the other hand, the feedback composition of $\mathcal{C}_2$ is not well-formed. Indeed, similarly, as just shown above for s_1, one gets a fixed point in state q_1 for both input F and T. But, there is no fixed point in state q_2 i.e. given an input sequence $x \in \{T,F\}^\omega$, there is no output sequence $y(n) \in \{T,F\}^\omega$ such that $y(n) \in \alpha_2(q_2)(f(x(n),y(n))))$. In fact, if we attempt to choose $x(n) = F$, then $\mathcal{C}_2$ may stay in state q_2 and its output may be the same as the fed back input T ($T \in \alpha_2(q_2)(f(F,T))$). If we attempt to choose $x(n) = T$, then $\mathcal{C}_2$ may go either to state q_1 and its output is not the same as the fed back input T ($T \notin \alpha_2(q_2)(f(T,T))$, or to stay in state q_2 and its output is not the same as the fed back input F ($F \notin \alpha_2(q_2)(f(T,F))$).

Hence, systems with feedbacks not well-formed (called *ill-formed*) will be rejected. They are considered to be unstable and insecure systems.

Definition 1.5 ***(Synchronous feedback** $\circlearrowleft$) Let $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$ be a signature and $\mathcal{I} = (f, \pi_i, \pi_o)$ be a feedback interface over H. Let us note $H' = T(\mathsf{Out}' \times \_)^{\mathsf{In}'}$. Let us define for every $x \in \mathsf{In}^\omega$, the set Θ_x of output sequences $y \in \mathsf{Out}^\omega$ defined from an infinite sequence of states $(s_0, s_1, \ldots, s_k, \ldots)$ of S as follows:*

$\forall n, 0 \leq n < \omega,\ (y(n), s_{n+1}) \in \eta'_{\mathsf{Out}\times S}(\alpha(s_n)(f(x(n),y(n))))$ (by hypothesis, $\mathcal{C}$'s feedback composition being well-formed over $\mathcal{I}$, such y exists)

Then, the operation of synchronous feedback *over $\mathcal{I}$ is the partial mapping*

$$\circlearrowleft_{\mathcal{I}}: \mathbf{Comp}(H) \longrightarrow \mathbf{Comp}(H')$$

that associates to every component $\mathcal{C} = (S, s_0, \alpha)$ over H whose feedback composition is well-formed, the component $(\langle \mathbb{F} \rangle, \mathbb{F}, \alpha_{\langle \mathbb{F} \rangle})$ over H' where $\mathbb{F}$ is the set of transfer functions $\mathcal{F} : \mathsf{In}'^\omega \longrightarrow \mathsf{Out}'^\omega$, each one defined by $\mathcal{F}(x') = y'$ where there exists $x \in \mathsf{In}^\omega$ s.t. there exists $y \in \Theta_x$ satisfying

$$\forall i, i < \omega, x'(i) = \pi_i(x(i)) \text{ and } y'(i) = \pi_o(y_x(i))$$

Proposition 1.2 $\circlearrowleft_{\mathcal{I}}: \mathbf{Comp}(H) \longrightarrow \mathbf{Comp}(H')$ *is a partial functor only defined for component $\mathcal{C}$ whose the synchronous feedback composition over $\mathcal{I}$ is well-formed.*

Proof *The proof is noticeably similar to the proof given for $\hookleftarrow_{\mathcal{I}}$.*
End

Example 1.3 *Consider again the component $\mathcal{C}_1$ shown in Figure V.5a and let us built the composite component that hides the feedback, as suggested by the last definition. We then choose the component $f : \mathsf{In} \times \mathsf{Out} \longrightarrow \mathsf{In}$ as the "and" operator, π_i and π_o as the identities on* In *and* Out *respectively.*
The function $\mathcal{F} : \{T,F\}^\omega \longrightarrow \{F,T\}^\omega$ defined for every $x \in \{T,F\}^\omega$ and for every $k, 0 \leq k < \omega$, by:

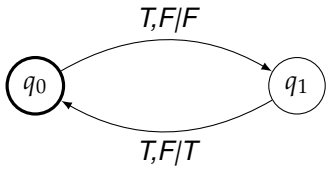
$$\mathcal{F}(x)(k) = \begin{cases} F & if\ k\ is\ even \\ T & otherwise \end{cases}$$

is the unique transfer function that can be defined using our synchronous feedback definition. Indeed, both associated outputs to each input $x \in \{T,F\}$ from s_1 are F, and both associated outputs to each

$x \in \{T,F\}$ from s_2 are T. Then the feedback composite $\circlearrowleft_{\mathcal{I}}(\mathcal{C})$ over the signature $\mathcal{I}$ is the component $(\langle\{\mathcal{F}\}\rangle, \{\mathcal{F}\}, \alpha_{\langle\{\mathcal{F}\}\rangle})$ where the set of states $\langle\{\mathcal{F}\}\rangle$ is obtained by a repeated computation of derivative starting from $\{\mathcal{F}\}$. The states of $\langle\{\mathcal{F}\}\rangle$ then contain the set of all derivative functions of $\mathcal{F}$ that are $\mathcal{F}$ and $\mathcal{F}'$ where $\mathcal{F}' : \{T,F\}^\omega \longrightarrow \{F,T\}^\omega$ is the function defined for every $x \in \{T,F\}^\omega$ and for every $k, 0 \leq k < \omega$, by:

$$\mathcal{F}'(x)(k) = \begin{cases} T & if \quad k \text{ is even} \\ F & otherwise \end{cases}$$

Note that computing further derivative sets will not yield any new transfer functions sets.
This then leads to the following component $\langle\{\mathcal{F}\}\rangle$:



We can also define the feedback in terms of its argument as concrete coalgebras, as done for the product in Definition 1.1, and not on behaviours as done in Definition 1.3 and Definition 1.5. For the synchronous feedback, this leads to:

Definition 1.6 *(**Synchronous feedback** $\circlearrowleft^c$) Let $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$ be a signature and $\mathcal{I} = (f, \pi_i, \pi_o)$ be a feedback interface over H. Let us note $H' = T(\mathsf{Out}' \times \_)^{\mathsf{In}'}$. The **operation of synchronous**[5] **feedback over** $\mathcal{I}$ is the partial functor $\circlearrowleft^c_{\mathcal{I}}$: $\mathbf{Comp}(H) \longrightarrow \mathbf{Comp}(H')$ that associates to every component $\mathcal{C} = (S, init, \alpha)$ over H whose the feedback composition over $\mathcal{I}$ is well-formed, the component $\mathcal{C}' = (S', init', \alpha')$ over H' such that:*

- *$S' = S$*
- *$init' = init$;*
- *$\alpha' : S' \longrightarrow T(\mathsf{Out}' \times S')^{\mathsf{In}'}$ is the transition mapping defined by:*
  $\forall s'_1 \in S', \forall i' \in \mathsf{In}', \alpha'(s'_1)(i') = \eta'^{-1}_{\mathsf{Out}' \times S'}(\Pi)$ where Π is the set:

$$\Pi = \left\{(o', s'_2) \mid \exists i \in \mathsf{In}, \exists o \in \mathsf{Out}, (o, s'_2) \in \eta'_{\mathsf{Out} \times S}(\alpha(s'_1)(f(i,o))), \pi_i(i) = i' \text{ and } \pi_o(o) = o'\right\}$$

Relaxed feedback can be defined similarly. Definition 1.5 and Definition 1.6 are equivalent. Indeed, it is obvious to check that

$$\mathsf{beh}_{\circlearrowleft^c_{\mathcal{I}}(\mathcal{C})}(init') = \mathsf{beh}_{\circlearrowleft_{\mathcal{I}}(\mathcal{C})}(\mathbb{F}) = \mathbb{F}$$

Although, $\circlearrowleft^c_{\mathcal{I}}$ is defined more uniformly with product $\otimes$ because both are defined as concrete coalgebras, the interest of $\circlearrowleft_{\mathcal{I}}$ (resp. $\hookleftarrow_{\mathcal{I}}$) is that the resulting component is the minimal one. This will allow easier compositionality proofs such as those given in Section 3.3 and Section 1.1 of Chapter VIII.

5. The exponent c in $\circlearrowleft^c_{\mathcal{I}}$ is to express that feedback is defined in terms of its argument as concrete coalgebras.

2 Complex operators

As previously explained, both cartesian product and feedback operators depend mainly on the component structure. However, when modeling systems, there is not only a need to specify component structure, but also some requirements for the input and output sets. On one hand, there is a need to make some component actions private and therefore inaccessible or hidden to the environment. To make this possible, our framework has to offer an operation to do that. This operator is classically called *hiding*[6]. Hidding aims to delimit the scope of both input and output sets. Let us observe that this operator can be naturally defined in terms of the feedback operator by taking the elements f, π_i and π_o of the feedback interface $\mathcal{I}$ as follows:

- $f : \mathsf{In} \times \mathsf{Out} \longrightarrow \mathsf{In}$ is the mapping defined by $(i, o) \mapsto i$;
- $\pi_i : \mathsf{In} \longrightarrow \{\mathsf{abs}\} \cup \mathsf{In} \setminus \mathsf{In}'$ is the mapping defined by:

$$\pi_i(i) = \begin{cases} i & \text{if} \quad i \in \mathsf{In} \setminus \mathsf{In}' \\ \mathsf{abs} & \text{otherwise} \end{cases}$$

- $\pi_o : \mathsf{Out} \longrightarrow \{\mathsf{abs}\} \cup \mathsf{Out} \setminus \mathsf{Out}'$ is the mapping defined by:

$$\pi_o(o) = \begin{cases} o & \text{if} \quad o \in \mathsf{Out} \setminus \mathsf{Out}' \\ \mathsf{abs} & \text{otherwise} \end{cases}$$

In the following, we denote by $\mathcal{C}[h_{in}, h_{out}]$ a component $\mathcal{C}$ over $T(\mathsf{Out} \times \_)^{\mathsf{In}}$ in which In and Out are restricted using h_{in} and h_{out} where h_{in} and h_{out} stand for π_i and π_o respectively.

On the other hand, we need another operation that allows us to rename component actions. This is useful when components have a common behavioural pattern and can be seen as specific instances of a generic component. We then define an operation called *renaming*[7] aiming to rename inputs and outputs of components. The renaming operation is defined as a pair of bijective functions

$$r_{in} : \mathsf{In} \longrightarrow \mathsf{In}' \quad \text{and} \quad r_{out} : \mathsf{Out} \longrightarrow \mathsf{Out}'$$

that maps each $i \in \mathsf{In}$ to an element $i' \in \mathsf{In}'$ (respectively, each $o \in \mathsf{Out}$ to an element $o \in \mathsf{Out}'$). We denote by $\mathcal{C}]_{r_{in}, r_{out}}$ a component $\mathcal{C}$ over $T(\mathsf{Out} \times \_)^{\mathsf{In}}$ in which In and Out are renamed using r_{in} and r_{out}.

Definition 2.1 *(**Complex operator**) The **set of complex operators**, is inductively defined as follows:*

- *$\_$ is a complex operator of arity 1;*
- *op is a complex operator of arity n and (r_{in}, r_{out}) is a renaming couple, then $(r_{in}, r_{out})(op)$ is a complex operator of arity n;*
- *if op_1 and op_2 are complex operators of arity n_1 and n_2 respectively, then $op_1 \otimes op_2$ is a complex operator of arity $n_1 + n_2$;*
- *if op is complex operator of arity n and $\mathcal{I}$ is a feedback interface, then $\circlearrowleft_{\mathcal{I}}(op)$ is a complex operator of arity n;*
- *if op is complex operator of arity n and $\mathcal{I}$ is a feedback interface, then $\leftrightarrow_{\mathcal{I}}(op)$ is a complex operator of arity n.*

6. also known as *restriction* operation.
7. also known as *relabeling* operation.

To show that both cartesian product and feedback operators are expressive enough to define, by composition, standard composition operators, we explain in this section how sequential, double sequential, concurrent and synchronous parallel compositions, and synchronous product can be obtained by their composition. Hence, the definition of the feedback interface $\mathcal{I} = (f, \pi_i, \pi_o)$ is rather wide and abstract in the sense that each suitable choice of f, π_i and π_o gives a particular semantic to the feedback composition and then a way of defining the global and final reaction of the composition of a set of components.

Before defining these operators in our framework, let us note two important points:

– there seems to be no consensus on the terminologies used to describe composition operators. For instance, sequential composition is introduced as "cascade composition" in [84, 85] and "1-way-cascade" in [86], the synchronous parallel composition is called "synchronous composition" in [87] and does not have the same semantic as "interleaving parallel composition" introduced in [86]. The cartesian product where the set of inputs are considered as disjoint is called "synchronous side-by-side composition" in [85]. The concurrent composition is called "asynchronous side-by-side composition" in [85], and there are many other examples. Therefore, we prefer to choose the terminology the best adapted to our framework, and give the informal definition of each operator before giving its formal definition to avoid any confusing terminology. Further technical details about the different kinds of composition presented in the following can be found in textbooks such as [84, 85, 88].

– The symbol abs is previously used to express the absence of component reaction when modeling *IOLTS* as components. Here, it is also useful for building complex components. Hence, the sets In_1 and In_2 (respectively, Out_1 and Out_2) may include some special action, denoted by abs, in order to allow components to stutter [8]. For instance, if the input action is (abs, i_2) with $i_2 \neq \mathsf{abs}$, the reaction of the composite consists only of the reaction of the second component (i.e. the composite behaves like the second component). Double stuttering corresponds to the input $(\mathsf{abs}, \mathsf{abs})$ with the following additional requirement that, for any $s \in S_1 \times S_2$, $\eta'_{(\mathsf{Out1}\times\mathsf{Out}_2)\times S}(\alpha(s)((\mathsf{abs}, \mathsf{abs}))) = \{((\mathsf{abs}, \mathsf{abs}), s)\}$. We will see later in this section that this can be useful for building larger components from smaller ones using composition. This is of interest if we want to take into account a reaction of the composite with only one of the components that reacts.

2.1 Sequential composition

The *sequential composition* (called also *cascade composition*, or *series composition*), denoted by $\mathcal{C} = \triangleright((\mathcal{C}_1, \mathcal{C}_2))$, of two components $\mathcal{C}_1$ and $\mathcal{C}_2$ corresponds to a composition where both components $\mathcal{C}_1$ and $\mathcal{C}_2$ are interconnected side-by-side and the output of one is the input of the other. Figure V.6 illustrates this kind of composition.
A reaction of $\mathcal{C}$ consists then of a reaction of both $\mathcal{C}_1$ and $\mathcal{C}_2$, where $\mathcal{C}_1$ reacts first, produces its outputs, and then $\mathcal{C}_2$ reacts. That is to say, when $\mathcal{C}_1$ is triggered by an input i from the environment, $\mathcal{C}_1$ executes i and the produced output is fed to $\mathcal{C}_2$. A requirement for this composition to be defined is that Out_1 has to be included into In_2 ($\mathsf{Out}_1 \subseteq \mathsf{In}_2$). This ensures that any output produced by $\mathcal{C}_1$ is an acceptable input to $\mathcal{C}_2$.

This kind of composition can be naturally defined in our framework using both the feedback operator and the cartesian product by:

$$\triangleright((\mathcal{C}_1, \mathcal{C}_2)) = \ominus_{\mathcal{I}}((\mathcal{C}_1 \otimes \mathcal{C}_2)) \tag{V.2}$$

8. **stutter** indicates that no progress of the component execution is made.

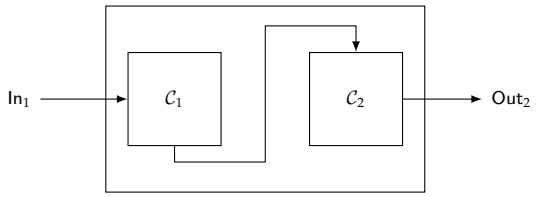


Figure V.6 – Sequential composition

where $\mathcal{I} = (f, \pi_i, \pi_o)$ is the feedback interface defined for every $(i, i') \in \mathsf{In}_1 \times \mathsf{In}_2$ and $(o, o') \in \mathsf{Out}_1 \times \mathsf{Out}_2$ as follows:

$$f((i,i'),(o,o')) = (i,o), \quad \pi_i((i,i')) = i \quad \text{and} \quad \pi_o((o,o')) = o'$$

and $\ominus$ stands for $\hookleftarrow$ or $\circlearrowleft$ depending on whether we want a relaxed or instantaneous sequential composition. For the first sequential composition, the output o produced from the component $\mathcal{C}_1$ after triggering by an input i takes some observable time to feed to the component $\mathcal{C}_2$. In this case, $\triangleright$ will be denoted by $\triangleright_r$. For the second one, the output o produced from the component $\mathcal{C}_1$ after triggering by an input i is directly fed to the input of the component $\mathcal{C}_2$. In this case, $\triangleright$ will be denoted by $\triangleright_s$.
Note that in both cases there is a causality dependency; that is, the outputs of $\mathcal{C}_1$ can affect the behaviour of $\mathcal{C}_2$ i.e. sequential composition entails an ordering of the component reactions.

As already mentioned, our synchronous feedback operator $\circlearrowleft$ is only applied to systems whose composition is well-formed. Therefore, applying this operator to the cartesian product $\otimes((\mathcal{C}_1, \mathcal{C}_2))$ of two components $\mathcal{C}_1$ and $\mathcal{C}_2$ requires that the composition of $\otimes((\mathcal{C}_1, \mathcal{C}_2))$ is well-formed.

Theorm 2.1 *Let $H_1 = T(\mathsf{Out}_1 \times \_)^{\mathsf{In}_1}$ and $H_2 = T(\mathsf{Out}_2 \times \_)^{\mathsf{Out}_1}$ be two signatures such that $\mathsf{Out}_1 \subseteq \mathsf{In}_2$. Let $\mathcal{I} = (f, \pi_i, \pi_o)$ be the feedback interface defined for every $(i, i') \in \mathsf{In}_1 \times \mathsf{In}_2$ and every $(o, o') \in \mathsf{Out}_1 \times \mathsf{Out}_2$ as follows:*

$$f((i,i'),(o,o')) = (i,o), \quad \pi_i((i,i')) = i \quad \textit{and} \quad \pi_o((o,o')) = o'$$

*Let $\mathcal{C}_1 \in \mathbf{Comp}(H_1)$ and $\mathcal{C}_2 \in \mathbf{Comp}(H_2)$. Then **the feedback composition of** $\otimes((\mathcal{C}_1, \mathcal{C}_2))$ **is well-formed**.*

Proof *Let $\mathcal{C}_1 = (S_1, \alpha_1) \in \mathbf{Comp}(H_1), \mathcal{C}_2 = (S_2, \alpha_2) \in \mathbf{Comp}(H_2)$ and $\mathcal{C} = \otimes((\mathcal{C}_1, \mathcal{C}_2)) = (S, \alpha)$ be the cartesian product of $\mathcal{C}_1$ and $\mathcal{C}_2$. Let us show that the synchronous feedback composition of $\mathcal{C}$ is well-formed. For this, let $(s_1, s_2) \in S$ be a state in S and (i_1, i_2) be an input in $\mathsf{In}_1 \times \mathsf{In}_2$ and then show that there exists an output $(o_1, o_2) \in \mathsf{Out}_1 \times \mathsf{Out}_2$ such that:*

$$(o_1, o_2) \in \eta'_{(\mathsf{Out}_1 \times \mathsf{Out}_2) \times S}(\alpha((s_1, s_2))(f((i_1, i_2), (o_1, o_2))))_{|_1} \quad (1)$$

*i.e. by the definition of f $(o_1, o_2) \in \eta'_{(\mathsf{Out}_1 \times \mathsf{Out}_2) \times S}(\alpha((s_1, s_2))((i_1, o_1)))_{|_1}$
$(i_1, i_2) \in \mathsf{In}_1 \times \mathsf{In}_2$, then there exists an output $o_1 \in \eta'_{\mathsf{Out}_1 \times S_1}(\alpha_1(s_1)(i_1))_{|_1}$. We also know that o_1 is an input of $\mathcal{C}_2$ since $\mathsf{Out}_1 \subseteq \mathsf{In}_2$. Hence, there exists an output $o_2 \in \eta'_{\mathsf{Out}_2 \times S_2}(\alpha_2(s_2)(o_1))_{|_1}$. We can now conclude that (i_1, o_1) is an input of $\mathcal{C}$ and there exists an output (o_1, o_2) of $\mathcal{C}$ such that:*

$$(o_1, o_2) \in \eta'_{(\mathsf{Out}_1 \times \mathsf{Out}_2) \times S}(\alpha((s_1, s_2))((i_1, o_1)))_{|_1}$$

Consequently, (1) is verified.
End

Let us explain now how the synchronous sequential composition of two components can be obtained with our modeling. Two components $\mathcal{C}_1$ and $\mathcal{C}_2$ are sequentially interconnected by linking the output of $\mathcal{C}_1$ to the input of $\mathcal{C}_2$. This interconnection is made without taking time (i.e. instantaneously). Suppose that at the nth reaction the input action of $\mathcal{C} = \triangleright_s((\mathcal{C}_1, \mathcal{C}_2))$ is $x(n) = (i_{1n}, i_{2n})$, the state is $s_n = (s_{1n}, s_{2n})$ (s_{1n} is the state of component $\mathcal{C}_1$ and s_{2n} is the state of component $\mathcal{C}_2$). There is an output action $y(n) = (o_{1n}, o_{2n}) \in \eta'_{\mathsf{Out}\times S}(s_n)(f(x(n), y(n))_{|_1}$ because the feedback composition of $\mathcal{C}$ is well-formed. Then, $f((i_{1n}, i_{2n}), (o_{1n}, o_{2n}))$ becomes the new input action of $\mathcal{C}$. This is equal to (i_{1n}, o_{1n}) according to the definition of $\mathcal{I}$. This means i_{1n} becomes the input of $\mathcal{C}_1$ and the output action o_{1n} of $\mathcal{C}_1$ becomes the input of $\mathcal{C}_2$ at the nth reaction. Then, component $\mathcal{C}_1$ (resp. $\mathcal{C}_2$) reacts by updating its state to $s_{1(n+1)}$ (resp. to $s_{2(n+1)}$) and producing an output action o_{1n} (resp. o_{2n}). Finally, to omit outputs that are involved in the feedback, we use π_i and π_o. Hence, at any reaction n, $\pi_i((i_{1n}, i_{2n})) = i_{1n}$ and $\pi_o((o_{1n}, o_{2n})) = o_{2n}$.

2.2 Double sequential composition

The *double sequential composition*, denoted by $\mathcal{C} = \bowtie((\mathcal{C}_1, \mathcal{C}_2))$, of two components $\mathcal{C}_1$ and $\mathcal{C}_2$ is a composition in which the system can be triggered either by an input of $\mathcal{C}_1$ and then feeds the output produced to $\mathcal{C}_2$ or by an input of $\mathcal{C}_2$ and then feeds the output produced to $\mathcal{C}_1$ (see Figure V.7).

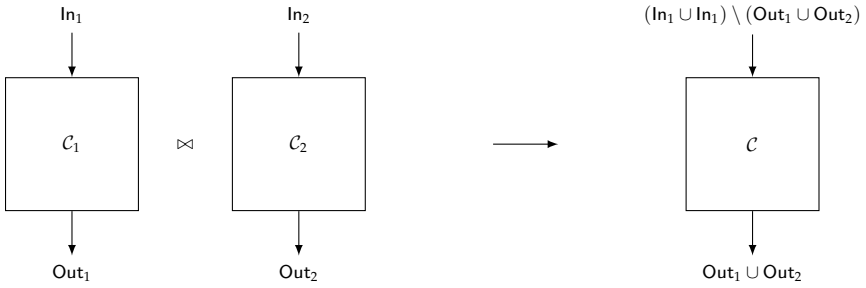


Figure V.7 – Double sequential composition

In our framework, this kind of composition can be obtained by modifying the cartesian product to be able to express not only the component reaction of the form $(i_1, i_2)/(o_1, o_2)$ but also those of the form $(i_2, i_1)/(o_2, o_1)$ while keeping the sequential feedback interface $\mathcal{I}$ as defined in Section 2.1. In other words, we extend the cartesian product operator $\otimes$ to an operator $\otimes_{\mathrm{e}}$ taking as input the set $(\mathsf{In}_1 \times \mathsf{In}_2) \cup (\mathsf{In}_2 \times \mathsf{In}_1)$ and as output the set $(\mathsf{Out}_1 \times \mathsf{Out}_2) \cup (\mathsf{Out}_2 \times \mathsf{Out}_1)$. This operator can be naturally expressed using both the sequential operator $\triangleright_s$ and the cartesian product $\otimes$ as follows:

$$\otimes_{\mathrm{e}}((\mathcal{C}_1, \mathcal{C}_2)) = \triangleright_s(\triangleright_s(\mathcal{C}_0, \otimes(\mathcal{C}_1, \mathcal{C}_2)), \mathcal{C}'_0) \tag{V.3}$$

where:

– $\mathcal{C}_0 = (\{init_0\}, init_0, \alpha_0)$ is the component over the signature

$$T((\mathsf{In}_1 \times \mathsf{In}_2) \times \_)^{(\mathsf{In}_1 \times \mathsf{In}_2) \cup (\mathsf{In}_2 \times \mathsf{In}_1)}$$

where α_0 is the transition mapping defined by: $\forall (i, i') \in (\mathsf{In}_1 \times \mathsf{In}_2) \cup (\mathsf{In}_2 \times \mathsf{In}_1)$

$$\alpha_0(init_0)(i, i') = \begin{cases} ((i, i'), init_0) & \text{if} \quad (i, i') \in \mathsf{In}_1 \times \mathsf{In}_2 \\ ((i', i), init_0) & \text{otherwise} \end{cases}$$

– $\mathcal{C}'_0 = (\{init'_0\}, init'_0, \alpha'_0)$ is the component over the signature

$$T((\mathsf{Out}_1 \times \mathsf{Out}_2) \cup (\mathsf{Out}_2 \times \mathsf{Out}_1) \times \_)^{(\mathsf{Out}_1 \times \mathsf{Out}_2)}$$

where α'_0 is the transition mapping defined by: $\forall (o, o') \in \mathsf{Out}_1 \times \mathsf{Out}_2$

$$\alpha'_0(init'_0)(o, o') = \begin{cases} ((o, o'), init'_0) & \text{if} \quad (o, o') \in (\mathsf{Out}_1 \cap \mathsf{In}_2) \times \mathsf{Out}_2 \\ ((o', o), init'_0) & \text{if} \quad (o, o') \in \mathsf{Out}_1 \times (\mathsf{Out}_2 \cap \mathsf{In}_1) \end{cases}$$

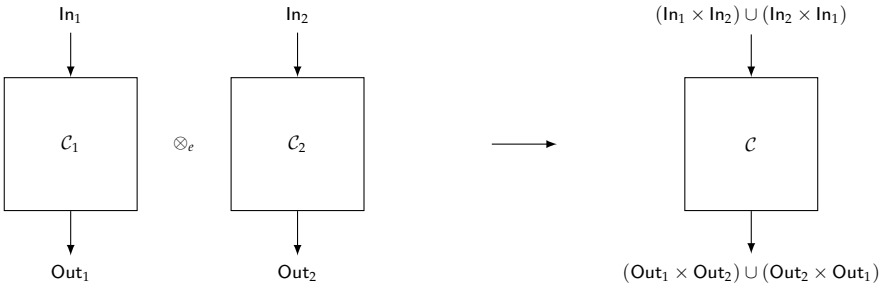


Figure V.8 – Extended cartesian product $\otimes_e$

Now, it is expected that $\bowtie(\mathcal{C}_1, \mathcal{C}_2)$ should be $\circlearrowleft_{\mathcal{I}}(\otimes_{\mathrm{e}}(\mathcal{C}_1, \mathcal{C}_2))$ with $\mathcal{I}$ as the sequential feedback interface, however this is not the case due to the fact that the feedback composition of $\otimes_{\mathrm{e}}(\mathcal{C}_1, \mathcal{C}_2)$ over the sequential interface $\mathcal{I}$ is not necessarily well-formed. Indeed, $\otimes_{\mathrm{e}}(\mathcal{C}_1, \mathcal{C}_2)$ may take an input (i_1, i_2) for which there is no output (o_1, o_2) that feeds back as input to $\otimes_{\mathrm{e}}(\mathcal{C}_1, \mathcal{C}_2)$. For instance, when $i_1 \in \mathsf{In}_1 \cap \mathsf{Out}_2$ and $i_2 \in \mathsf{In}_2 \setminus \mathsf{Out}_1$. To cope with this problem, it suffices to replace the set of inputs of $\otimes_{\mathrm{e}}(\mathcal{C}_1, \mathcal{C}_2)$ by

$$((\mathsf{In}_2 \setminus \mathsf{Out}_2) \times (\mathsf{In}_2 \cap \mathsf{Out}_1)) \cup ((\mathsf{In}_2 \setminus \mathsf{Out}_1) \times (\mathsf{In}_1 \cap \mathsf{Out}_2))$$

instead of $(\mathsf{In}_1 \times \mathsf{In}_2) \cup (\mathsf{In}_2 \times \mathsf{In}_1)$, and define $\mathcal{C}_0$ over the signature

$$T((\mathsf{In}_1 \times \mathsf{In}_2) \times \_)^{((\mathsf{In}_1 \setminus \mathsf{Out}_2) \times (\mathsf{In}_2 \cap \mathsf{Out}_1)) \cup ((\mathsf{In}_2 \setminus \mathsf{Out}_1) \times (\mathsf{In}_1 \cap \mathsf{Out}_2))}$$

where α_0 is the transition mapping defined by:
$\forall (i, i') \in ((\mathsf{In}_1 \setminus \mathsf{Out}_2) \times (\mathsf{In}_2 \cap \mathsf{Out}_1)) \cup ((\mathsf{In}_2 \setminus \mathsf{Out}_1) \times (\mathsf{In}_1 \cap \mathsf{Out}_2))$

$$\alpha_0(init_0)((i, i')) = \begin{cases} ((i, i'), init_0) & \text{if} \quad (i, i') \in (\mathsf{In}_1 \setminus \mathsf{Out}_2) \times (\mathsf{In}_2 \cap \mathsf{Out}_1) \\ ((i', i), init_0) & \text{otherwise} \end{cases}$$

Figure V.9 illustrates a simple example of the application of the extended cartesian product.

Hence, it is easy to see that the synchronous feedback composition of $\otimes_{\mathrm{e}}((\mathcal{C}_1, \mathcal{C}_2))$ is well-formed. Thus, the double sequential composition is defined by:

$$\bowtie((\mathcal{C}_1, \mathcal{C}_2) = \circlearrowleft_{\mathcal{I}}(\otimes_{\mathrm{e}}(\mathcal{C}_1, \mathcal{C}_2)) \tag{V.4}$$

with $\mathcal{I}$ is the sequential feedback interface.

Figure V.9 – Example: illustration of $\otimes_e$

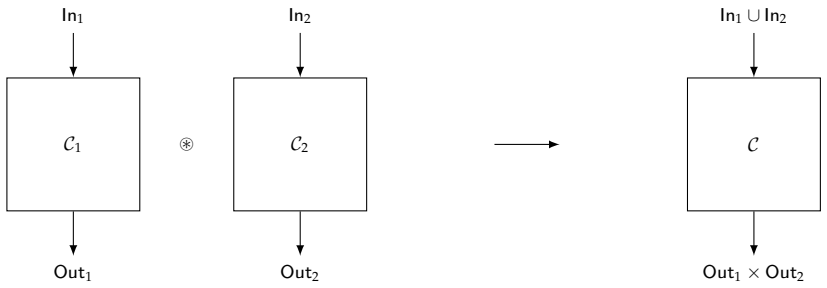


Figure V.10 – Synchronous product

2.3 Synchronous product

The *synchronous product*, denoted by $\mathcal{C} = \circledast((\mathcal{C}_1, \mathcal{C}_2))$, of two components $\mathcal{C}_1$ and $\mathcal{C}_2$ corresponds to a composition where both components $\mathcal{C}_1$ and $\mathcal{C}_2$ are executed independently or jointly, depending on the input. Hence, $\mathcal{C}_1$ and $\mathcal{C}_2$ are simultaneously executed when triggered by a joint input i that belongs to both inputs set of $\mathcal{C}_1$ and $\mathcal{C}_2$ (see Figure V.10).
This kind of product can also be naturally expressed in terms of the synchronous feedback operator and the cartesian product (see Figure V.11) as follows:

$$\circledast((\mathcal{C}_1, \mathcal{C}_2)) = \rhd_s(\mathcal{C}_0, (\mathcal{C}_1 \otimes \mathcal{C}_2)) \tag{V.5}$$

where $\mathcal{C}_0 = (\{init_0\}, init_0, \alpha_0)$ is the component over the signature

$$T((\mathsf{In}_1 \times \mathsf{In}_2) \times \_)^{\mathsf{In}_1 \cup \mathsf{In}_2}$$

where α_0 is the transition mapping defined by: $\forall i \in \mathsf{In}_1 \cup \mathsf{In}_2$

$$\alpha_0(init_0)(i) = \begin{cases} ((i,i), init_0) & \text{if} \quad i \in \mathsf{In}_1 \cap \mathsf{In}_2 \\ ((i, \mathsf{abs}), init_0) & \text{if} \quad i \in (\mathsf{In}_1 \setminus \mathsf{In}_1 \cap \mathsf{In}_2) \\ ((\mathsf{abs}, i), init_0) & \text{otherwise} \end{cases}$$

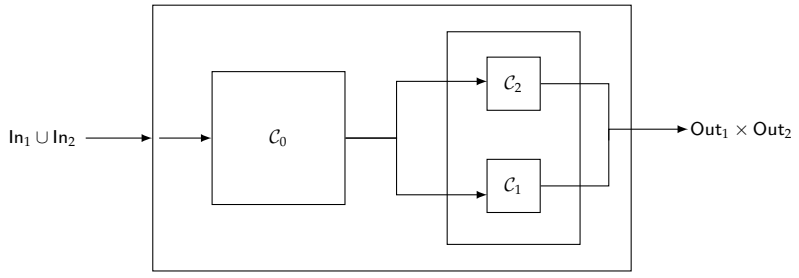


Figure V.11 – Synchronous product: $\circledast((\mathcal{C}_1, \mathcal{C}_2)) = \rhd_s(\mathcal{C}_0, (\mathcal{C}_1 \otimes \mathcal{C}_2))$

2.4 Concurrent composition

The *concurrent composition*, denoted by $\mathcal{C} = \oplus((\mathcal{C}_1, \mathcal{C}_2))$, of two components $\mathcal{C}_1$ and $\mathcal{C}_2$ corresponds to a composition where both components $\mathcal{C}_1$ and $\mathcal{C}_2$ are executed independently or jointly, depending on the input received from environment. It combines both choice and parallel compositions, in the sense $\mathcal{C}_1$ and $\mathcal{C}_2$ can be simultaneously executed when triggered by a pair of inputs (i_1, i_2) (i_1 belongs to inputs set of $\mathcal{C}_1$ and i_2 belongs to inputs set of $\mathcal{C}_2$), or separately when triggered by an input i: if $i \in \mathsf{In}_1$, then $\mathcal{C}_1$ is executed and the reaction of $\mathcal{C}$ is that of $\mathcal{C}_1$, otherwise $\mathcal{C}_2$ is executed and the reaction of $\mathcal{C}$ is that of $\mathcal{C}_2$. Figure V.12 illustrates this kind of composition.

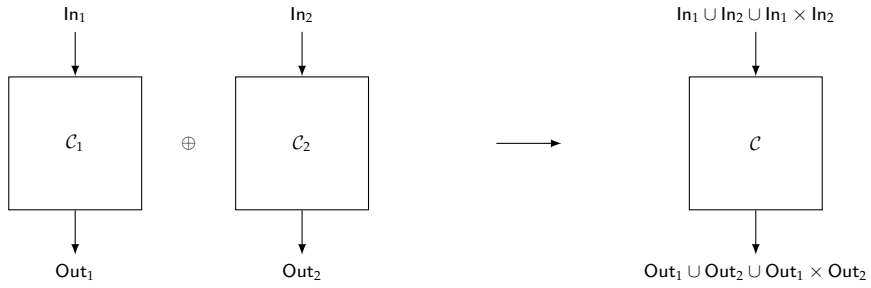


Figure V.12 – Concurrent composition

This kind of composition can also be naturally expressed in terms of the synchronous feedback operator and the cartesian product (see Figure V.13) as follows:

$$\oplus((\mathcal{C}_1, \mathcal{C}_2)) = \triangleright_s(\triangleright_s(\mathcal{C}_0, (\mathcal{C}_1 \otimes \mathcal{C}_2)), \mathcal{C}'_0) \tag{V.6}$$

where

– $\mathcal{C}_0 = (\{init_0\}, init_0, \alpha_0)$ is the component over the signature

$$T((\mathsf{In}_1 \times \mathsf{In}_2) \times \_)^{\mathsf{In}_1 \cup \mathsf{In}_2 \cup \mathsf{In}_1 \times \mathsf{In}_2}$$

where α_0 is the transition mapping defined by: $\forall i \in \mathsf{In}_1 \cup \mathsf{In}_2 \cup \mathsf{In}_1 \times \mathsf{In}_2$

$$\alpha_0(init_0)(i) = \begin{cases} (i, init_0) & \text{if} \quad i \in \mathsf{In}_1 \times \mathsf{In}_2 \\ ((i, \mathsf{abs}), init_0) & \text{if} \quad i \in \mathsf{In}_1 \\ ((\mathsf{abs}, i), init_0) & \text{otherwise} \end{cases}$$

– $\mathcal{C}'_0 = (\{init'_0\}, init'_0, \alpha'_0)$ is the component over the signature

$$T((\mathsf{Out}_1 \cup \mathsf{Out}_2 \cup \mathsf{Out}_1 \times \mathsf{Out}_2) \times \_)^{\mathsf{Out}_1 \times \mathsf{Out}_2}$$

where α'_0 is the transition mapping defined by: $\forall o = (o_1, o_2) \in \mathsf{Out}_1 \times \mathsf{Out}_2$

$$\alpha'_0(init'_0)(o) = \begin{cases} (o_1, init'_0) & \text{if} \quad o \in \mathsf{Out}_1 \times \{\mathsf{abs}\} \\ (o_2, init'_0) & \text{if} \quad o \in \{\mathsf{abs}\} \times \mathsf{Out}_2 \\ (o, init'_0) & \text{otherwise} \end{cases}$$

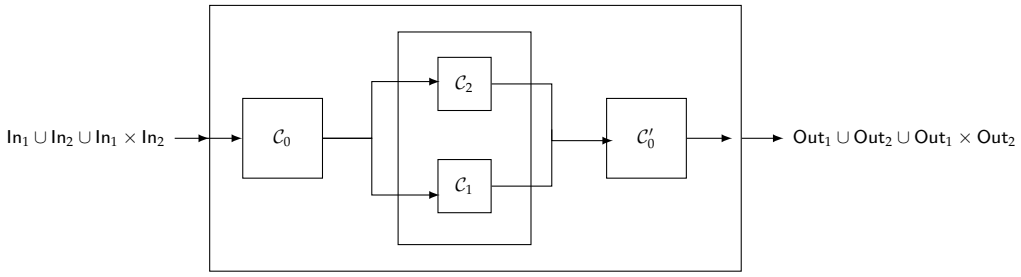


Figure V.13 – Concurrent composition: $\oplus((\mathcal{C}_1, \mathcal{C}_2)) = \rhd_s(\rhd_s(\mathcal{C}_0, (\mathcal{C}_1 \otimes \mathcal{C}_2)), \mathcal{C}'_0)$

2.5 Synchronous parallel composition

The *synchronous parallel composition*, denoted by $\mathcal{C} = \odot((\mathcal{C}_1, \mathcal{C}_2))$, of two components $\mathcal{C}_1$ and $\mathcal{C}_2$ is a composition in which both $\mathcal{C}_1$ and $\mathcal{C}_2$ are executed independently or jointly depending on the input, in such a way that each input action received by $\mathcal{C}$ from the environment consists exclusively of an input action of either $\mathcal{C}_1$ or $\mathcal{C}_2$ i.e. there is no common input action for $\mathcal{C}_1$ and $\mathcal{C}_2$. Indeed, when the global system receives an input which is supposed to be an input action of $\mathcal{C}_1$, $\mathcal{C}_1$ reacts by producing an output. If that output does not belong to the input set of $\mathcal{C}_2$, the reaction of the global system consists only of the reaction of $\mathcal{C}_1$. Otherwise, the output produced is directly fed to $\mathcal{C}_2$ and the reaction of the global system consists of the reaction of both $\mathcal{C}_1$ and $\mathcal{C}_2$ (one falls into the same composition as the sequential[9] composition). In the same manner, when the global system receives an input supposed to be an input action of $\mathcal{C}_2$, $\mathcal{C}_2$ reacts by producing an output. If that output does not belong to the input set of $\mathcal{C}_1$, the reaction of the global system consists only of the reaction of $\mathcal{C}_2$. Otherwise, the output produced is directly fed to $\mathcal{C}_1$ and the reaction of the global system consists of the reaction of both $\mathcal{C}_1$ and $\mathcal{C}_2$ (see Figure V.14).

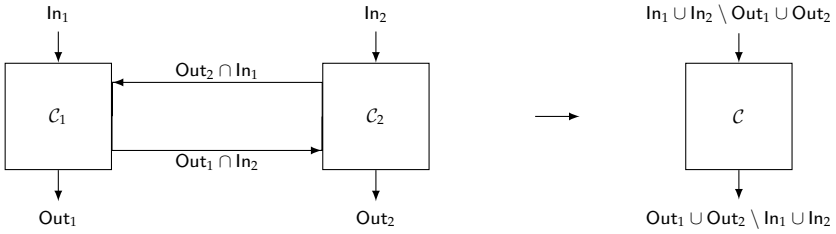


Figure V.14 – Synchronous parallel composition

This kind of composition can be seen as a general composition embodying both the synchronous and parallel (or interleaving parallel) aspects of composition. On one hand, it is synchronous in the sense that all common actions between $\mathcal{C}_1$ and $\mathcal{C}_2$ are synchronized. That means each output of $\mathcal{C}_1$ that is fed as input of $\mathcal{C}_2$ (i.e. $\mathsf{Out}_1 \cap \mathsf{In}_2$) and each output of $\mathcal{C}_2$ that is fed as input of $\mathcal{C}_1$ ($\mathsf{Out}_2 \cap \mathsf{In}_1$) are hidden (i.e. synchronized). They are not observable from the outside. On the other hand, it is parallel in the sense that both components $\mathcal{C}_1$ and $\mathcal{C}_2$ are considered autonomous, that is to say, a component may produce an output o regardless of whether o is specified as an input of the other component.

This kind of operator is not easy to be formally defined in our framework without any special treatment. In fact, it is not clear how the global reaction of the integrated system is given

9. Note it is easy to see that the double sequential composition is a particular case of the synchronous parallel composition.

when outputs produced by $\mathcal{C}_1$ (respectively $\mathcal{C}_2$) are unspecified in $\mathcal{C}_2$ (respectively $\mathcal{C}_1$). Nevertheless, let us observe that if the concurrent composition operator $\oplus$ is extended to an operator $\oplus_e$ taking as inputs not only the set $\mathsf{In}_1 \cup \mathsf{In}_2 \cup \mathsf{In}_1 \times \mathsf{In}_2$, but also the set $\mathsf{In}_2 \times \mathsf{In}_1$, and modifying the sequential feedback signature to deal with cases where component outputs are not fed back as inputs, we do not have that problem.

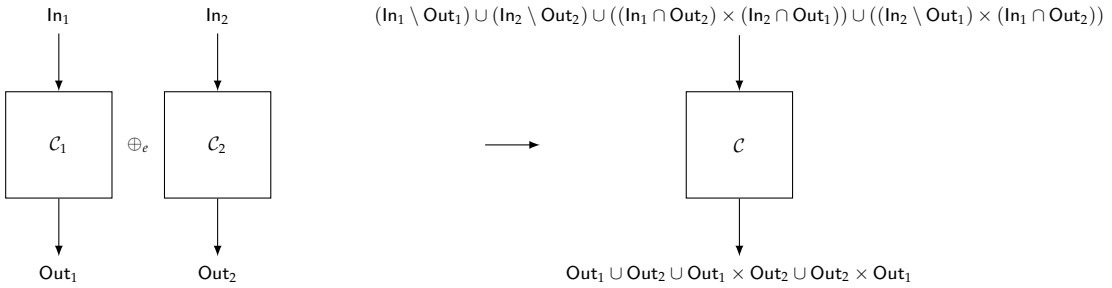


Figure V.15 – Extended concurrent composition $\oplus_e$

Let us first define the extended concurrent operator $\oplus_e$. Similarly as $\oplus$, $\oplus_e$ can also be naturally expressed in terms of the synchronous feedback operator and the cartesian product as follows:

$$\oplus_e((\mathcal{C}_1, \mathcal{C}_2)) = \triangleright_s(\triangleright_s(\mathcal{C}_0, (\mathcal{C}_1 \otimes \mathcal{C}_2)), \mathcal{C}'_0) \tag{V.7}$$

where

1. $\mathcal{C}_0 = (\{init_0\}, init_0, \alpha_0)$ is the component over the signature

$$T((\mathsf{In}_1 \times \mathsf{In}_2) \times \_)^{(\mathsf{In}_1 \setminus \mathsf{Out}_2) \cup (\mathsf{In}_2 \setminus \mathsf{Out}_1) \cup ((\mathsf{In}_1 \setminus \mathsf{Out}_2) \times (\mathsf{In}_2 \cap \mathsf{Out}_1)) \cup ((\mathsf{In}_2 \setminus \mathsf{Out}_1) \times (\mathsf{In}_1 \cap \mathsf{Out}_2))}$$

where α_0 is the transition mapping defined by:
$\forall i \in (\mathsf{In}_1 \setminus \mathsf{Out}_2) \cup (\mathsf{In}_2 \setminus \mathsf{Out}_1) \cup ((\mathsf{In}_1 \setminus \mathsf{Out}_2) \times (\mathsf{In}_2 \cap \mathsf{Out}_1)) \cup ((\mathsf{In}_2 \setminus \mathsf{Out}_1) \times (\mathsf{In}_1 \cap \mathsf{Out}_2))$

$$\alpha_0(init_0)(i) = \begin{cases} ((i_1, i_2), init_0) & \text{if} \quad i = (i_1, i_2) \in ((\mathsf{In}_1 \setminus \mathsf{Out}_2) \times (\mathsf{In}_2 \cap \mathsf{Out}_1)) \\ ((i_2, i_1), init_0) & \text{if} \quad i = (i_1, i_2) \in ((\mathsf{In}_2 \setminus \mathsf{Out}_1) \times (\mathsf{In}_1 \cap \mathsf{Out}_2)) \\ ((i, \mathsf{abs}), init_0) & \text{if} \quad i \in (\mathsf{In}_1 \setminus \mathsf{Out}_2) \\ ((\mathsf{abs}, i), init_0) & \text{otherwise} \end{cases}$$

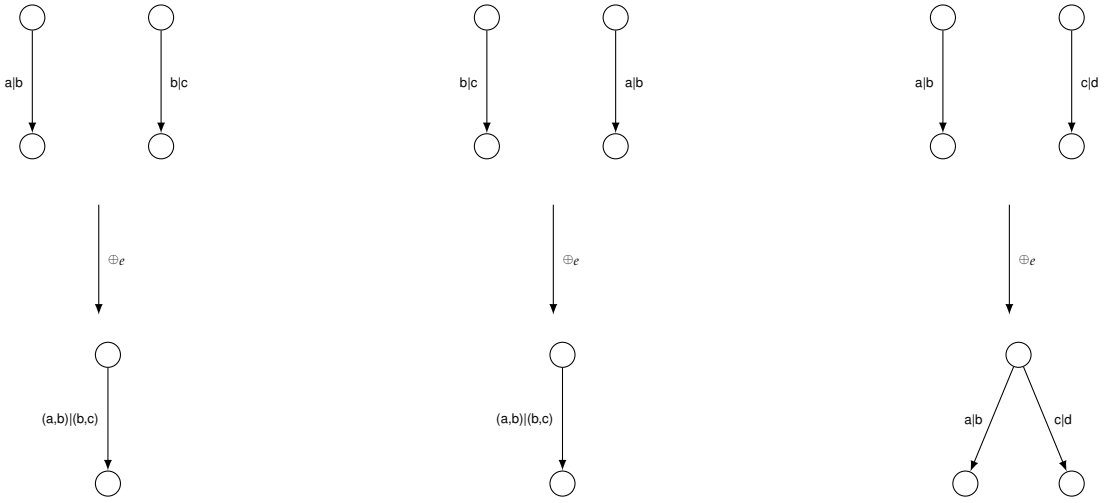
2. $\mathcal{C}'_0 = (\{init'_0\}, init'_0, \alpha'_0)$ is the component over the signature

$$T((\mathsf{Out}_1 \cup \mathsf{Out}_2 \cup (\mathsf{Out}_1 \times \mathsf{Out}_2) \cup (\mathsf{Out}_2 \times \mathsf{Out}_1)) \times \_)^{\mathsf{Out}_1 \times \mathsf{Out}_2}$$

where α'_0 is the transition mapping defined by: $\forall o = (o_1, o_2) \in \mathsf{Out}_1 \times \mathsf{Out}_2$

$$\alpha'_0(init'_0)(o) = \begin{cases} (o_1, init'_0) & \text{if} & o \in \mathsf{Out}_1 \times \{\mathsf{abs}\} \\ (o_2, init'_0) & \text{if} & o \in \{\mathsf{abs}\} \times \mathsf{Out}_2 \\ (o, init'_0) & \text{if} & o \in \mathsf{Out}_1 \times \mathsf{Out}_2 \\ ((o_2, o_1), init'_0) & \text{otherwise} & \end{cases}$$

Figure V.16 illustrates a simple example of the application of the extended concurrent operator.



(a) Cas 1: synchronization left-right

(b) Cas 2: synchronization right-left

(c) Cas 3: parallel composition

Figure V.16 – Example: illustration of $\oplus_e$

In this way, the synchronous parallel composition is defined in our framework as:

$$\odot(\mathcal{C}_1, \mathcal{C}_2) = \circlearrowleft_{\mathcal{I}} (\oplus_e(\mathcal{C}_1, \mathcal{C}_2)) \tag{V.8}$$

with $\mathcal{I} = (f, \pi_i, \pi_o)$ is the feedback interface defined:
$\forall i \in (\mathsf{In}_1 \setminus \mathsf{Out}_2) \cup (\mathsf{In}_2 \setminus \mathsf{Out}_1) \cup ((\mathsf{In}_1 \setminus \mathsf{Out}_2) \times (\mathsf{In}_2 \cap \mathsf{Out}_1)) \cup ((\mathsf{In}_2 \setminus \mathsf{Out}_1) \times (\mathsf{In}_1 \cap \mathsf{Out}_2))$ and $o \in \mathsf{Out}_1 \cup \mathsf{Out}_2 \cup (\mathsf{Out}_1 \times \mathsf{Out}_2) \cup (\mathsf{Out}_2 \times \mathsf{Out}_1)$ as follows:

$$f(i,o) = \begin{cases} i & \text{if} \quad i \in (\mathsf{In}_1 \setminus \mathsf{Out}_2) \cup (\mathsf{In}_2 \setminus \mathsf{Out}_1) \\ (i_1, o_1) \text{ with } i = (i_1, i_2) \text{ and } o = (o_1, o_2) & \text{otherwise} \end{cases}$$

3 Systems and compositionality

3.1 Systems

Complex operators for basic components yield larger components that we will call *systems*. However, it is not always possible to yield a component for a complex operator from any set of basic components passed in arguments. Indeed, for a complex operator of the form $\circlearrowleft_{\mathcal{I}}(op)$, according to the component $\mathcal{C}$ resulting from the evaluation of op, the interface $\mathcal{I}$ has to be defined over the signature of $\mathcal{C}$ and the feedback over $\mathcal{C}$ has to be well-formed over $\mathcal{I}$. This leads up to the following definition:

Definition 3.1 *(**Systems**) Let $\mathbb{C}$ be a set of components. The **set of systems over** $\mathbb{C}$ is inductively defined as follows:*

- *for any $\mathcal{C} \in \mathbb{C}$, a component over a signature H, $\_(\mathcal{C}) = \mathcal{C}$ is a system over H and $\_$ is **defined for** $\mathcal{C}$;*
- *if $(r_{in}, r_{out})(op)$ is a complex operator of arity n, then for every sequence $(\mathcal{C}_1, \ldots, \mathcal{C}_n)$ of components with $(S, init, \alpha)) = op(\mathcal{C}_1, \ldots, \mathcal{C}_n)$ is over $T(\mathsf{Out} \times \_)^{\mathsf{In}}$, then $(r_{in}, r_{out})op(\mathcal{C}_1, \ldots, \mathcal{C}_n)$*

is the component $(S', init', \alpha')$ *over* $T(r_{out}(\mathsf{Out}) \times \_)^{r_{in}(\mathsf{In})}$ *such that:* $S' = S, init' = init$ *and* $\forall s' \in S', \forall i' \in r_{in}(\mathsf{In})$

$\eta'^{-1}(\alpha'(s')(i'))_{|_1} = r_{out}(\eta'^{-1}(\alpha(s')(r_{in}^{-1}(i')))_{|_1}$ *and* $\eta'^{-1}(\alpha'(s')(i'))_{|_2} = \eta'^{-1}(\alpha(s')(r_{in}^{-1}(i'))_{|_2}$

– *if* $op_1 \otimes op_2$ *is a complex operator of arity* $n = n_1 + n_2$*, then for every sequence*

$$(\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_{n_1}, \mathcal{C}_{n_1+1}, \ldots, \mathcal{C}_n)$$

z of components in $\mathbb{C}$ *with each* $\mathcal{C}_i$ *over* $H_i = T(O_i \times \_)^{I_i}$*, if both* op_1 *and* op_2 *are defined for* $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_{n_1}$ *and* $\mathcal{C}_{n_1+1}, \ldots, \mathcal{C}_n$ *respectively, then* $op_1 \otimes op_2(\mathcal{C}_1, \ldots, \mathcal{C}_n) = op_1(\mathcal{C}_1, \ldots, \mathcal{C}_{n_1}) \otimes op_2(\mathcal{C}_{n_1+1}, \ldots, \mathcal{C}_n)$ *is a system over* $H = T(\prod_{i=1}^{n} O_i \times \_)^{\prod_{i=1}^{n} I_i}$ *and* $op_1 \otimes op_2$ *is* ***defined for*** $(\mathcal{C}_1, \ldots, \mathcal{C}_n)$*, else* $op_1 \otimes op_2$ *is* ***undefined for*** $(\mathcal{C}_1, \ldots, \mathcal{C}_n)$*;*

– *if* $\circlearrowleft_{\mathcal{I}}(op)$ *is a complex operator of arity n, then for every sequence* $(\mathcal{C}_1, \ldots, \mathcal{C}_n)$ *of components in* $\mathbb{C}$*, if op is defined for* $(\mathcal{C}_1, \ldots, \mathcal{C}_n)$ *with* $\mathcal{S} = op(\mathcal{C}_1, \ldots, \mathcal{C}_n)$ *is over H,* $\mathcal{I}$ *is a feedback interface over H and the feedback composition of* $\mathcal{S}$ *is well-formed, then* $\circlearrowleft_{\mathcal{I}}(op)(\mathcal{C}_1, \ldots, \mathcal{C}_n) = \circlearrowleft_{\mathcal{I}}(\mathcal{S})$ *is a system over* H' *and*[10] $\circlearrowleft_{\mathcal{I}}(op)$ *is* ***defined for*** $(\mathcal{C}_1, \ldots, \mathcal{C}_n)$*, else* $\circlearrowleft_{\mathcal{I}}(op)$ *is* ***undefined for*** $(\mathcal{C}_1, \ldots, \mathcal{C}_n)$*;*

– *if* $\hookleftarrow_{\mathcal{I}}(op)$ *is a complex operator of arity n, then for every sequence* $(\mathcal{C}_1, \ldots, \mathcal{C}_n)$ *of components in* $\mathbb{C}$*, if op is defined for* $(\mathcal{C}_1, \ldots, \mathcal{C}_n)$ *with* $\mathcal{S} = op(\mathcal{C}_1, \ldots, \mathcal{C}_n)$ *is over H and* $\mathcal{I}$ *is a feedback interface over H, then* $\hookleftarrow_{\mathcal{I}}(op)(\mathcal{C}_1, \ldots, \mathcal{C}_n) = \hookleftarrow_{\mathcal{I}}(\mathcal{S})$ *is a system over* H' *and*[11] $\hookleftarrow_{\mathcal{I}}(op)$ *is* ***defined for*** $(\mathcal{C}_1, \ldots, \mathcal{C}_n)$*, else* $\hookleftarrow_{\mathcal{I}}(op)$ *is* ***undefined for*** $(\mathcal{C}_1, \ldots, \mathcal{C}_n)$*.*

From Proposition 1.1 and Proposition 1.2, it is not difficult to see that any complex operator op of arity n defines a partial functor from $\mathbf{Comp}(H_1) \times \cdots \times \mathbf{Comp}(H_n) \longrightarrow \mathbf{Comp}(H)$.

3.2 Examples

In the following, we present some concrete examples illustrating our framework.

Example 3.1 ***(Encoder/decoder)*** *An* encoder/decoder *is usually used to guarantee certain characteristics (for example, error detection) when transmitting data across a link. A simple example of such an encoder/decoder is represented in Figure V.17. It consists of two parts:*

– *An* encoder *that takes in an incoming bit sequence and produces an encoded value which is then transmitted on the link. This encoder is considered as a component* $\mathcal{E} = (\{s_0, s_1\}, s_0, \alpha_1)$ *where the transition function* $\alpha_1 : \{s_0, s_1\} \longrightarrow (\{0,1\} \times \{s_0, s_1\})^{\{0,1\}}$ *is graphically shown in the left of Figure V.17.*

– *A* decoder *that takes the values from the link and produces the original value. This decoder is considered as a component* $\mathcal{D} = (\{q_0, q_1\}, q_0, \alpha_2)$ *where the transition function* $\alpha_2 : \{q_0, q_1\} \longrightarrow (\{0,1\} \times \{q_0, q_1\})^{\{0,1\}}$ *is graphically shown in the right of Figure V.17.*

Let us now construct the encoder/decoder as a composition of the encoder and the decoder by means of the sequential composition over a synchronous feedback. First of all, let us apply the sequential composition $\triangleright_s(\otimes(\mathcal{E}, \mathcal{D}))$ *over the synchronous feedback interface* $\mathcal{I}$ *defined for every* $(i, i') \in \mathsf{In}_1 \times \mathsf{In}_2$ *and* $(o, o') \in \mathsf{Out}_1 \times \mathsf{Out}_2$ *by:*

$$f((i, i'), (o, o')) = (i, o), \quad \pi_i((i, i')) = i \quad \text{and} \quad \pi_o((o, o')) = o'$$

We first define the cartesian product $\mathcal{C} = \otimes((\mathcal{E}, \mathcal{D}))$ *of* $\mathcal{E}$ *and* $\mathcal{D}$*. It is easy to see that* $\mathcal{C}$ *is a well-formed feedback composition over* $\mathcal{I}$*. Let us check this for* (s_0, q_0)*, we then have:*

10. H' is the signature of the synchronous feedback.
11. H' is the signature of the relaxed feedback.

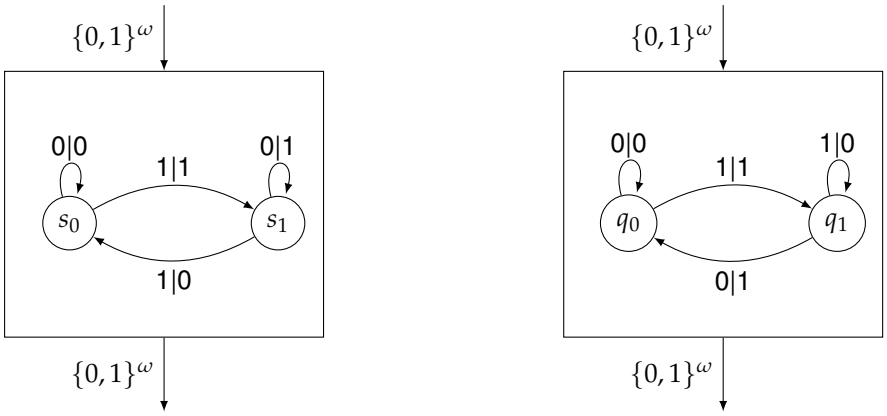


Figure V.17 – Encoder (on the left) and Decoder (on the right)

- $(0,0) \in \eta'(\alpha_{\mathcal{C}}((s_0,q_0))(f((0,0),(0,0))))_{|_1}$
- $(1,1) \in \eta'(\alpha_{\mathcal{C}}((s_0,q_0))(f((1,1),(1,1))))_{|_1}$
- $(0,0) \in \eta'(\alpha_{\mathcal{C}}((s_0,q_0))(f((0,1),(0,0))))_{|_1}$
- $(1,1) \in \eta'(\alpha_{\mathcal{C}}((s_0,q_0))(f((1,0),(1,1))))_{|_1}$

Then, we can apply the synchronous feedback operator $\circlearrowleft_{\mathcal{I}}$ on $\mathcal{C}$. This leads to a minimal component $\langle\{\mathcal{F}\}\rangle$ where $\mathcal{F} : \{0,1\}^\omega \longrightarrow \{0,1\}^\omega$ is the transfer function defined for every $x \in \{0,1\}^\omega$ and every $k, 0 \leq k < \omega$, by:

$$\mathcal{F}(x)(k) = x(k)$$

Let us explain how $\mathcal{F}$ was obtained using a running example. For this, let us consider the bit sequence $(01)^\omega$, and try to find a bit sequence $y \in \{0,1\}^\omega$ satisfying:

$$\exists (s_0,\dots,s_k,\dots) \in S \mid \forall n, 0 \leq n < \omega,\ y(n) \in \eta'_{\mathsf{Out}\times S}(\alpha(s_n)(f(x(n),y(n))))_{|_1}$$

Let us suppose that the current state and the current input are the initial state $s(n) = (s_0,q_0)$ and $x(n) = (0,0)$ respectively. There is a $y(n) = (0,0)$ such that:

$$(0,0) \in \eta'(\alpha_{\mathcal{C}}((s_0,q_0))(f((0,0),(0,0))))$$

That is to say, the component $\mathcal{C}$ reacts by updating its state to (s_0,q_0) and producing the output $(0,0)$. More precisely, the output of $\mathcal{E}$ becomes the input of $\mathcal{D}$. So, we can conclude that the input of the encoder/decoder is $\pi_1(0,0) = 0$ and its output is $\pi_o(0,0) = 0$.
Suppose next that the current input is $(1,1)$. Again, there is a $y(n) = (1,1)$ such that

$$(1,1) \in \eta'(\alpha_{\mathcal{C}}((s_0,q_0))(f((1,1),(1,1))))$$

That is to say, the component $\mathcal{C}$ reacts by updating its state to (s_1,q_1) and producing the output $(1,1)$. So, we can conclude that the input of the encoder/decoder is $\pi_1(1,1) = 1$ and its output is $\pi_o(1,1) = 1$.

Hence, the composite machine alternates states on each reaction and produces the output bit sequence $(01)^\omega$ for the input bit sequence $(01)^\omega$.

Finally, the minimal component $\langle\{\mathcal{F}\}\rangle$ that represents $\mathcal{F}$ is given by:

Example 3.2 *The purpose of this example is to shed light on how new components can be built hierarchically from elementary basic components involving various integration operators. The example is a simple model of a system that checks whether two gates are well-closed (respectively well-opened) when they receive an order to close (respectively to open). It consists of three parts: a controller $\mathcal{C}$, two gates $\mathcal{G}_1$ and $\mathcal{G}_2$ and a special component $\mathcal{O}$ testing behaviour of $\mathcal{G}_1$ and $\mathcal{G}_2$. When the controller receives an order to close the gates (i.e. when "close" button is pressed), it sends to $\mathcal{G}_1$ and $\mathcal{G}_2$ a signal "close" which is simultaneously placed to $\mathcal{G}_1$ and $\mathcal{G}_2$. Hence, each one produces either a "closed" signal or fails to do so. We assume reactions of $\mathcal{G}_1$ and $\mathcal{G}_2$ are instantaneous, i.e. they take no time to be closed or opened. Then, $\mathcal{O}$ does nothing if both $\mathcal{G}_1$ and $\mathcal{G}_2$ are well-closed or opened, and raises an alarm otherwise. It can be thought of as a checker of closing* [12] *and opening gates.*

The global model $\mathcal{S}$ of this system is then built from three basic components:

Controller *$\mathcal{C}$: it produces a signal "close" when the close button is pressed and a signal "open" when the open button is pressed (see Figure V.18). Both "close" and "open" signals are supposed to be submitted simultaneously to $\mathcal{G}_1$ and $\mathcal{G}_2$.*
In our framework, $\mathcal{C}$ is specified as the coalgebra $\mathcal{C} = (\{\text{closed}, \text{opened}\}, \text{closed}, \alpha_{\mathcal{C}})$ over the signature $(\{\text{close}, \text{open}\} \times \_)^{\{\text{buttonO},\text{buttonC}\}}$ where

$$\alpha_{\mathcal{C}} : \{\text{closed}, \text{opened}\} \times \{\text{buttonO}, \text{buttonC}\} \longrightarrow (\{\text{close}, \text{open}\} \times \{\text{closed}, \text{opened}\})$$

is defined as follows:

$$\begin{cases} \alpha_{\mathcal{C}}(\text{closed})(\text{buttonC}) = (\text{close}, \text{opened}) \\ \alpha_{\mathcal{C}}(\text{opened})(\text{buttonO}) = (\text{open}, \text{closed}) \end{cases}$$

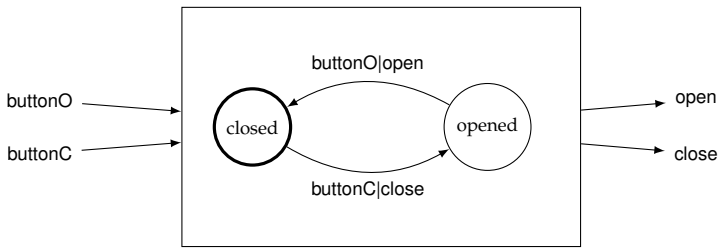


Figure V.18 – Controller system $\mathcal{C}$

Gate system *$\mathcal{G}$: it behaves as follows: when it receives the "close" signal from $\mathcal{C}$, it closes and when it receives the "open" signal from $\mathcal{C}$, it opens. Figure V.19 illustrates that behaviour.*
In our framework, $\mathcal{G}$ is specified as the coalgebra $\mathcal{G} = (\{\text{down}, \text{up}, \text{fail}\}, \text{up}, \alpha_{\mathcal{G}})$ over the signature $\mathcal{P}_{\text{fin}}(\{\text{closed}, \text{opened}, \text{abs}\} \times \_)^{\{\text{close},\text{open}\}}$ where

$$\alpha_{\mathcal{G}} : \{\text{down}, \text{up}, \text{fail}\} \times \{\text{close}, \text{open}\} \longrightarrow \mathcal{P}_{\text{fin}}(\{\text{closed}, \text{opened}, \text{abs}\} \times \{\text{up}, \text{down}, \text{fail}\})$$

is defined as follows:

$$\begin{cases} \alpha_{\mathcal{G}}(\text{up})(\text{close}) = \{(\text{closed}, \text{down}), (\text{abs}, \text{fail})\} \\ \alpha_{\mathcal{G}}(\text{down})(\text{open}) = \{(\text{opened}, \text{up})\} \end{cases}$$

12. For sake of the simplicity, we suppose that no error has occurred when closing the gates.

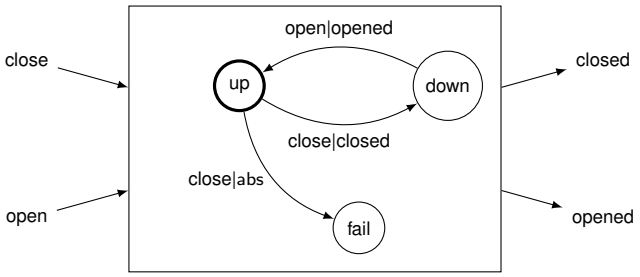


Figure V.19 – Gate system $\mathcal{G}$

Now, using our renaming operation, $\mathcal{G}_1$ and $\mathcal{G}_2$ can be seen as instances of the gate component $\mathcal{G}$. Then, $\mathcal{G}_1 = \mathcal{G}]_{r_{1in},r_{1out}}$ and $\mathcal{G}_2 = \mathcal{G}]_{r_{2in},r_{2out}}$ where both r_{1in} and r_{2in} are identities on $\{\text{close}, \text{open}\}$ and r_{1out} and r_{2out} are defined as follows:

$$\begin{array}{lll} r_{1out}: & \{\text{opened}, \text{closed}\} & \to \quad \{\text{opened}_1, \text{closed}_1\} \\ & \text{opened} & \mapsto \quad \text{opened}_1 \\ & \text{closed} & \mapsto \quad \text{closed}_1 \\ r_{2out}: & \{\text{opened}, \text{closed}\} & \to \quad \{\text{opened}_2, \text{closed}_2\} \\ & \text{opened} & \mapsto \quad \text{opened}_2 \\ & \text{closed} & \mapsto \quad \text{closed}_2 \end{array}$$

Checker system $\mathcal{O}$*: it receives the outputs of $\mathcal{G}_1$ and $\mathcal{G}_2$ and raises an alarm if there is a gate that is not completely closed.*
In our framework, $\mathcal{O}$ is specified as the following transfer function:

$$\begin{array}{lll} \mathcal{F}_{\mathcal{O}}: & \{\text{closed}_1, \text{abs}\} \times \{\text{closed}_2, \text{abs}\} & \longrightarrow \quad \{\text{alarm}, \text{abs}\} \\ & (o, o') & \mapsto \quad \begin{cases} \text{abs} & if \;\; (o, o') = (\text{closed}_1, \text{closed}_2) \\ \text{alarm} & otherwise \end{cases} \end{array}$$

Now, the global model $\mathcal{S}$ is given as a hierarchical composition of $\mathcal{C}, \mathcal{G}_1, \mathcal{G}_2$ and $\mathcal{O}$. $\mathcal{G}_1$ and $\mathcal{G}_2$ are a synchronous product composition, since their set inputs are the same, that together define a new component $\mathcal{B} = \circledast(\mathcal{G}_1, \mathcal{G}_2) = (\{b_0, b_1, b_2, b_3, b_4\}, b_0, \alpha_{\mathcal{B}})$ over the signature

$$\mathcal{P}_{\text{fin}}(\{(\text{opened}_1, \text{opened}_2), (\text{closed}_1, \text{closed}_2), (\text{closed}_1, \text{abs}), (\text{abs}, \text{closed}_2), (\text{abs}, \text{abs})\} \times \_)^{\{\text{close},\text{open}\}}$$

where $\alpha_{\mathcal{B}}$ is defined as follows:

$$\begin{cases} \alpha_{\mathcal{B}}(b_0)(\text{close}) = \{((\text{closed}_1, \text{closed}_2), b_1), ((\text{abs}, \text{abs}), b_2), ((\text{closed}_1, \text{abs}), b_3), ((\text{abs}, \text{closed}_2), b_4)\} \\ \alpha_{\mathcal{B}}(b_1)(\text{open}) = \{((\text{opened}_1, \text{opened}_2), b_0)\} \end{cases}$$

Figure V.20 illustrates graphically the component $\mathcal{B}$.

Then, $\mathcal{B}$ and $\mathcal{O}$ are a sequential composition since $\mathcal{B}$'s outputs are included into $\mathcal{O}$'s inputs. This leads to a new component $\mathcal{K} = \triangleright_s(\mathcal{B}, \mathcal{O}) = (\{k_0, k_1, k_2\}, k_0, \alpha_{\mathcal{K}})$ over the signature

$$\mathcal{P}_{\text{fin}}(\{\text{abs}, \text{alarm}\} \times \_)^{\{\text{close},\text{open}\}}$$

where $\alpha_{\mathcal{K}}$ is defined as follows:

$$\begin{cases} \alpha_{\mathcal{K}}(\text{k}_0)(\text{close}) = \{(\text{abs}, k_1), (\text{alarm}, k_2)\} \\ \alpha_{\mathcal{K}}(\text{k}_1)(\text{open}) = \{(\text{abs}, k_0)\} \end{cases}$$

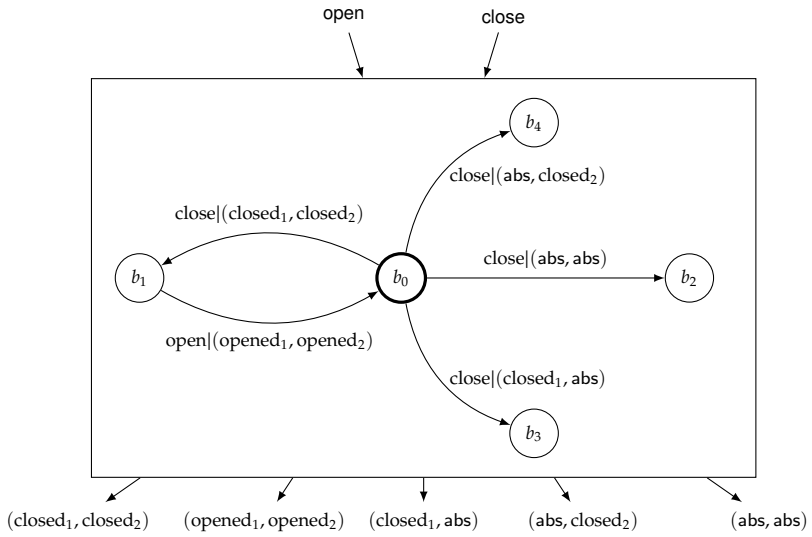


Figure V.20 – Synchronous product $\mathcal{B} = \circledast(\mathcal{G}_1, \mathcal{G}_2)$ of $\mathcal{G}_1$ and $\mathcal{G}_2$

Figure V.21 illustrates the component $\mathcal{K}$.

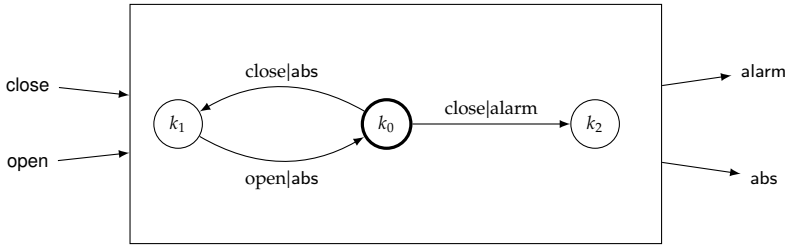


Figure V.21 – Sequential composition $\mathcal{K} = \triangleright_s(\mathcal{B}, \mathcal{O})$ of $\mathcal{B}$ and $\mathcal{O}$

Hence, the global system $\mathcal{S}$ can be given as a sequential composition of the controller $\mathcal{C}$ and $\mathcal{K}$. Thus, $\mathcal{S}$ is the component $\mathcal{S} = \triangleright_s(\mathcal{C}, \mathcal{K}) = (\{s_0, s_1, s_2\}, s_0, \alpha_{\mathcal{S}})$ over the signature $\mathcal{P}_{\text{fin}}(\{\mathsf{abs}, \mathsf{alarm}\} \times \_)^{\{\text{buttonC,buttonO}\}}$ where $\alpha_{\mathcal{S}}$ is defined as follows:

$$\begin{cases} \alpha_{\mathcal{S}}(s_0)(\text{buttonC}) = \{(\mathsf{abs}, s_1), (\mathsf{alarm}, s_2)\} \\ \alpha_{\mathcal{S}}(s_1)(\text{buttonO}) = \{(\mathsf{abs}, s_0)\} \end{cases}$$

Figure V.22 illustrates the component $\mathcal{S}$.

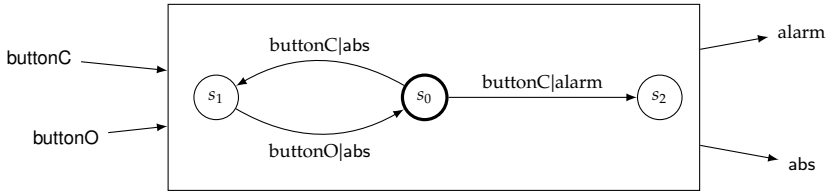


Figure V.22 – Sequential composition $\mathcal{S} = \triangleright_s(\mathcal{C}, \mathcal{K})$ of $\mathcal{C}$ and $\mathcal{K}$

Consequently, the global system $\mathcal{S}$ consists of

$$\triangleright_s(\mathcal{C}, \triangleright_s(\circledast(\mathcal{G}_1, \mathcal{G}_2), \mathcal{O}))$$

The two basic components $\mathcal{G}_1$ and $\mathcal{G}_2$ are composed together using the synchronous product and the resulting component $\mathcal{B} = \circledast(\mathcal{G}_1, \mathcal{G}_2)$ is composed sequentially with $\mathcal{O}$. Finally, the basic component $\mathcal{C}$ is composed sequentially with the result of the second composition $\mathcal{K} = \triangleright_s(\mathcal{C}, \mathcal{B})$.

Example 3.3 ***(Pedestrian crossing again)*** *We have presented in Example 1.3 the "*traffic light system*" $\mathcal{M}$ that constitutes the first part of the "*pedestrian crossing system*". In this example, we first consider the other part of the pedestrian crossing that is the "*crosswalk system*" and then show the pedestrian crossing global system obtained as a synchronous parallel composition of these two parts.*

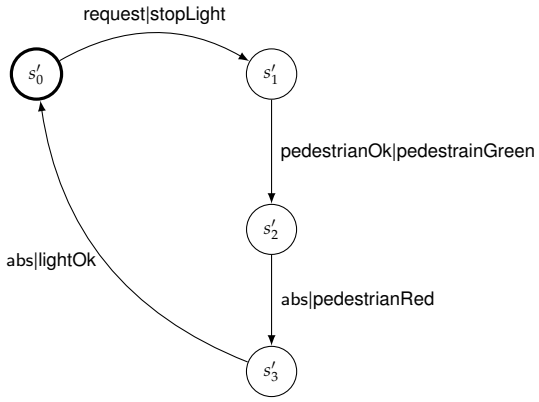


Figure V.23 – Model of a crosswalk, to be composed in a synchronous parallel composition with the traffic light model of Figure IV.4

The crosswalk system consists of two colored lights (see Figure IV.3): green and red. Illumination of the green light means that the road is vehicle-free and so pedestrians can cross safely. Illumination of the red light means there is a flow of vehicles at the road and so the pedestrians cannot cross. Such a typical crosswalk system $\mathcal{M}'$ can be modeled by the transition diagram shown in Figure V.23. The behaviour of $\mathcal{M}'$ is the following: from its initial state s'_0, when the pedestrian pushes the request button to request the green light, $\mathcal{M}'$ receives the request and goes to the s'_1 state. Then, it emits a signal to the traffic light system $\mathcal{M}$ to stop the flow of vehicles while going to the s'_2 state. When it receives the confirmation signal from the traffic light system, it illuminates the green light and sets in the pending state s'_4. Once the road is free of pedestrians, it illuminates the red light and sends a signal to the traffic light system to allow the vehicles to pass.

We model the crosswalk system as a component $\mathcal{M}' = (S', s'_0, \alpha_{\mathcal{M}'})$ over the signature $(\mathsf{Out}' \times \_)^{\mathsf{In}'}$ with $S' = \{s'_0, s'_1, s'_2\}$ is the state space, $\mathsf{In}' = \{\text{request}, \text{pedestrianOk}, \mathsf{abs}\}$ is the set of inputs and $\mathsf{Out}' = \{\text{stopLight}, \text{lightOk}, \text{pedestrianGreen}, \text{pedestrianRed}, \mathsf{abs}\}$ is the set of outputs. The transition function:

$$\alpha_{\mathcal{M}'} : S' \longrightarrow (\{\text{stopLight}, \text{lightOk}, \text{pedestrianGreen}, \text{pedestrianRed}, \mathsf{abs}\} \times S')^{\{\text{request}, \text{pedestrianOk}, \mathsf{abs}\}}$$

is defined as follows:

$$\begin{cases} \alpha_{\mathcal{M}'}(s'_0)(\text{request}) = (\text{stopLight}, s'_0) \\ \alpha_{\mathcal{M}'}(s'_0)(\text{pedestrianOk}) = (\text{pedestrianGreen}, s'_1) \\ \alpha_{\mathcal{M}'}(s'_1)(\mathsf{abs}) = (\text{pedestrianRed}, s'_2) \\ \alpha_{\mathcal{M}'}(s'_2)(\mathsf{abs}) = (\text{lightOk}, s'_0) \end{cases}$$

*The pedestrian crossing model $\mathcal{S}$ can be seen as a composition of the crosswalk component $\mathcal{M}'$ and the traffic light component $\mathcal{M}$, in which the "*stopLight*" action, the "*pedestrianOk*" action and the "*lightOk*" action are hidden as one can see in Figure V.24. The behaviour of $\mathcal{S}$ is then obtained as the synchronous parallel composition $\odot((\mathcal{M}', \mathcal{M}))$ of the individual components $\mathcal{M}'$ and $\mathcal{M}$. Output actions of one component that are in the input set actions of another component are synchronized i.e. the "*stopLight*", "*pedestrianOk*" and "*lightOk*" actions.*

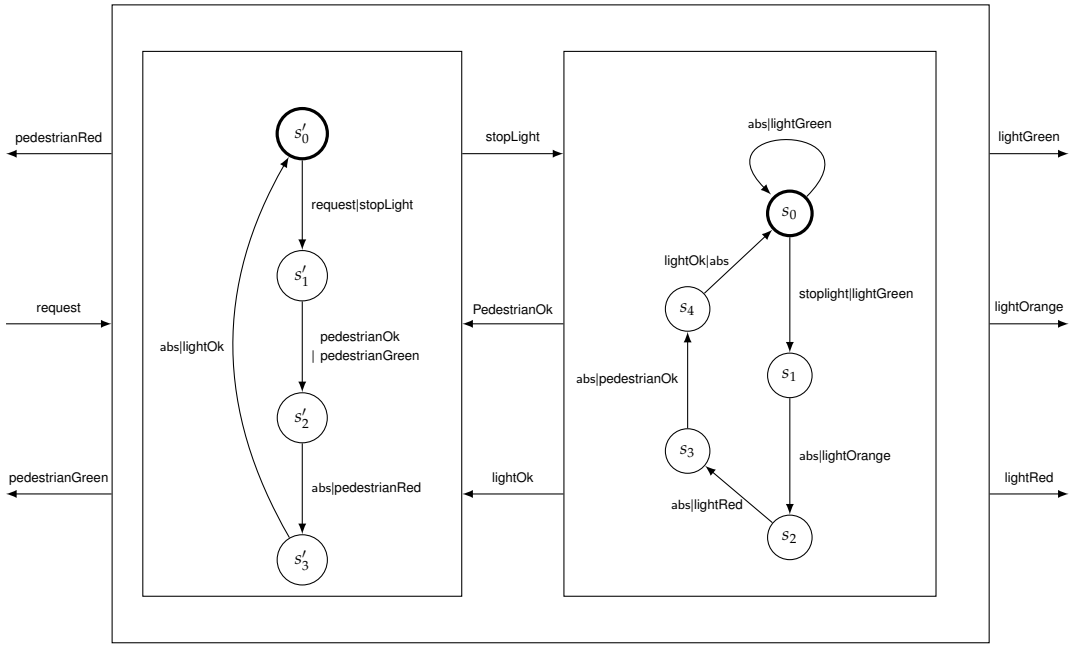


Figure V.24 – Pedestrian crossing modeling

Applying then the synchronous parallel composition $\odot$ defined in Section 2.5 on $\mathcal{M}'$ and $\mathcal{M}$ leads to a component over the signature

$$\{\text{PedestrianRed}, \text{PedestrianGreen}, \text{lightGreen}, \text{lightRed}, \text{lightOrange}, \text{abs}\} \times \_)^{\{\text{request},\text{abs}\}}$$

whose transition function is illustrated in Figure V.25.

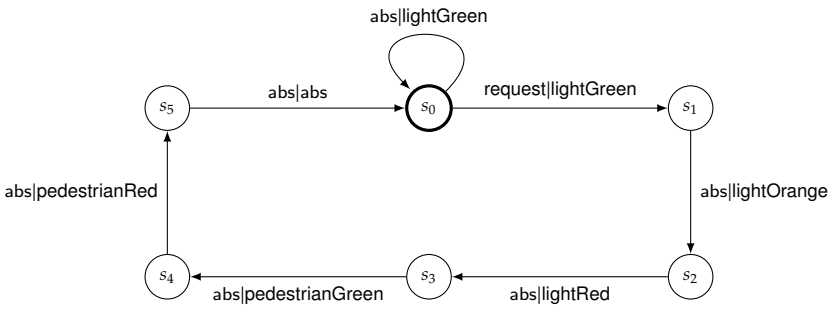


Figure V.25 – $\odot(\mathcal{M}', \mathcal{M})$

Example 3.4 *(**Level crossing**) We consider a simplified model of a* level crossing. *This model mainly consists of three parts: a single track railroad, a train, three detectors: "approach", "entry" and "exit" to detect the position of the train during its crossing of the road and the barrier. Figure V.26 illustrates a typical view of these elements.*

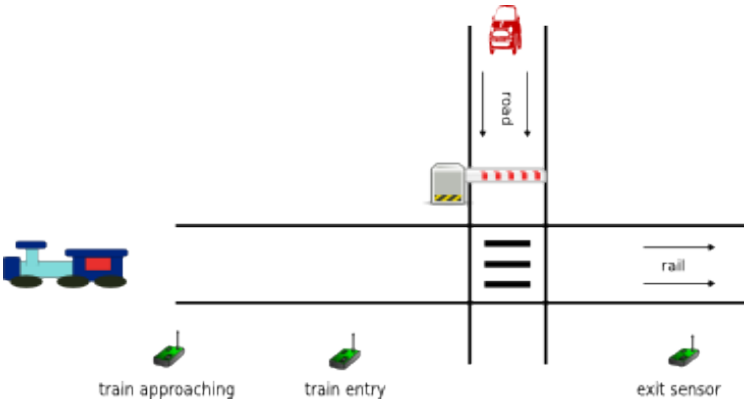


Figure V.26 – Level crossing

The behaviour of the global system of the level crossing is the following: when the "approach" detector detects an approaching train, it sends a signal to the barrier in order to go down. Once the train enters into the security zone, the "entry" detector detects the presence of the train and then sends a signal to the controller. Finally, once the train is crossed the railroad, the "exit" detector detects the train and sends a signal to the barrier in order to go back up. The model considered here does not take into consideration the errors produced during both the barrier raising and lowering in order to avoid any complications. It focuses only on safety properties such as "there is a train in the security zone while the barrier is not completely closed".

The system of the level crossing as it is explained above, is then built from two basic components:

*A **controller** $\mathcal{C}$ that produces a signal "close" when a train approaches, a signal "entry" when a train enters into the security zone and a signal "open" when a train exits the railroad. Both "close" and "open" signals are supposed to be submitted to the barrier. The controller is supposed to raise an alarm if there is a train in the security zone and the barrier is not completely closed.*
In our framework, $\mathcal{C}$ is specified as the coalgebra $\mathcal{C} = (\{\text{in}, \text{out}, \text{preparing}, \text{alarm}, \text{safe}\}, \text{out}, \alpha_{\mathcal{C}})$ over the signature

$$(\{\text{close}, \text{open}, \text{alarm}, \text{abs}\} \times \_)^{\{\text{approach,exit,entry,closed}\}}$$

where $\alpha_{\mathcal{C}} : \{\text{in}, \text{out}, \text{preparing}, \text{alarm}, \text{safe}\} \times \{\text{approach}, \text{exit}, \text{entry}, \text{closed}\} \longrightarrow$

$$(\{\text{close}, \text{open}, \text{alarm}, \text{abs}\} \times \{\text{in}, \text{out}, \text{preparing}, \text{alarm}, \text{safe}\})$$

is defined as follows:

$$\begin{cases} \alpha_{\mathcal{C}}(\text{out})(\text{approach}) = (\text{close}, \text{preparing}) \\ \alpha_{\mathcal{C}}(\text{preparing})(\text{closed}) = (\text{abs}, \text{safe}) \\ \alpha_{\mathcal{C}}(\text{preparing})(\text{entry}) = (\text{alarm}, \text{alarm}) \\ \alpha_{\mathcal{C}}(\text{safe})(\text{entry}) = (\text{abs}, \text{in}) \\ \alpha_{\mathcal{C}}(\text{in})(\text{exit}) = (\text{open}, \text{out}) \end{cases}$$

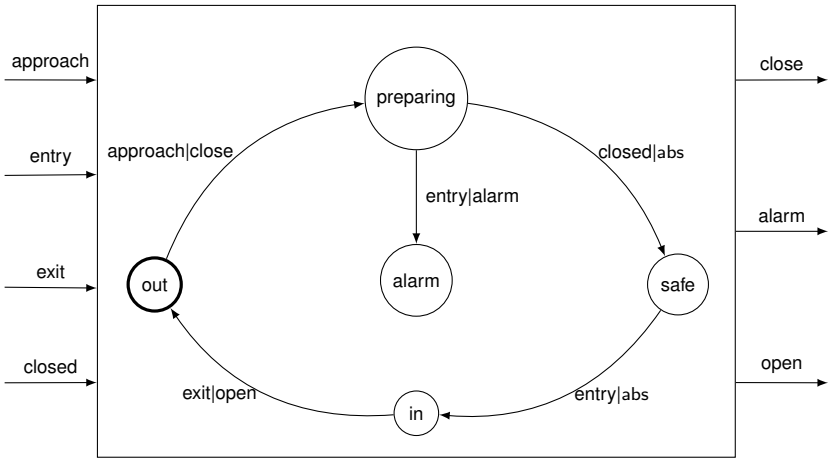


Figure V.27 – Controller system $\mathcal{C}$

Figure V.27 illustrates graphically the behaviour of the controller.

*A **barrier** system $\mathcal{B}$ that behaves as follows: when it receives the "close" signal from $\mathcal{C}$, it begins to lower and when it receives the "open" signal from $\mathcal{C}$, it begins to rise. We assume that the opening of the barrier is done instantaneously to make the example representation simple.*
In our framework, $\mathcal{B}$ is specified as the coalgebra $\mathcal{B} = (\{\text{up}, \text{closing}, \text{down}\}, \text{up}, \alpha_{\mathcal{B}})$ over the signature $(\{\text{closed}, \text{abs}\} \times \_)^{\{\text{close},\text{open},\text{abs}\}}$ where

$$\alpha_{\mathcal{B}} : \{\text{up}, \text{closing}, \text{down}\} \times \{\text{close}, \text{open}, \text{abs}\} \longrightarrow (\{\text{closed}, \text{abs}\} \times \{\text{up}, \text{closing}, \text{down}\})$$

is defined as follows:

$$\begin{cases} \alpha_{\mathcal{B}}(\text{up})(\text{close}) = (\text{abs}, \text{closing}) \\ \alpha_{\mathcal{B}}(\text{closing})(\text{abs}) = (\text{closed}, \text{down}) \\ \alpha_{\mathcal{B}}(\text{down})(\text{open}) = (\text{abs}, \text{up}) \end{cases}$$

Figure V.28 illustrates graphically the behaviour of $\mathcal{B}$.

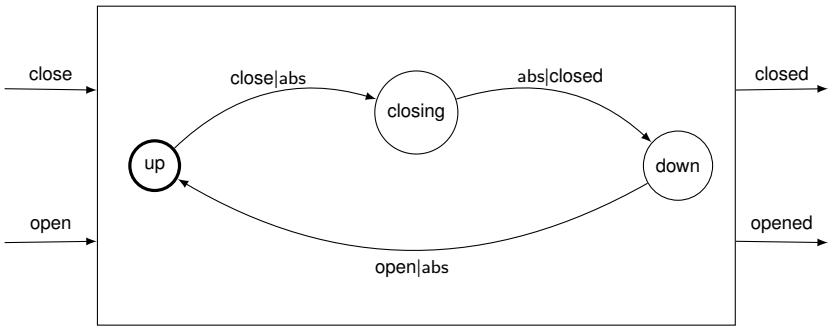


Figure V.28 – Barrier system $\mathcal{B}$

Now, the model $\mathcal{S}$ of the level crossing system is given as a synchronous parallel composition defined in Section 2.5 of $\mathcal{C}$ and $\mathcal{B}$. This leads to a new component

$$\mathcal{S} = \odot(\mathcal{C}, \mathcal{B}) = (\{s_0, s_1, s_2, s_3, s_4\}, s_0, \alpha_{\mathcal{S}})$$

over the signature:

$$(\{\mathsf{alarm}, \mathsf{abs}\} \times \_)^{\{\mathsf{approach},\mathsf{entry},\mathsf{exit},\mathsf{abs}\}}$$

where

$$\alpha_{\mathcal{S}} : \{s_0, s_1, s_2, s_3, s_4\} \times \{\mathsf{approach}, \mathsf{entry}, \mathsf{exit}, \mathsf{abs}\} \longrightarrow (\{\mathsf{alarm}, \mathsf{abs}\} \times \{s_0, s_1, s_2, s_3, s_4\})$$

is defined as follows:

$$\begin{cases} \alpha_{\mathcal{S}}(s_0)(\mathsf{approach}) = (\mathsf{abs}, s_1) \\ \alpha_{\mathcal{S}}(s_1)(\mathsf{entry}) = (\mathsf{alarm}, s_3) \\ \alpha_{\mathcal{S}}(s_1)(\mathsf{abs}) = \{\mathsf{abs}, s_2) \\ \alpha_{\mathcal{S}}(s_2)(\mathsf{entry}) = (\mathsf{abs}, s_4) \\ \alpha_{\mathcal{S}}(s_4)(\mathsf{exit}) = (\mathsf{abs}, s_0) \end{cases}$$

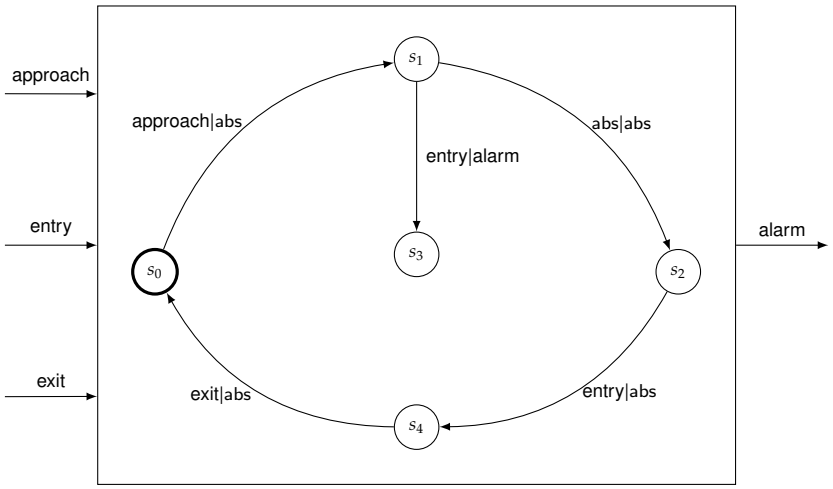


Figure V.29 – Crossing level global model $\mathcal{S}$

3.3 Compositionality

An important question we must address concerns compositionality: is the behaviour of a system the composition of its components' behaviours? In our framework, this will be expressed as follows: let op be a complex operator of arity n, $\mathcal{C}_1, \ldots, \mathcal{C}_n$ be n components and $\mathcal{C} = op(\mathcal{C}_1, \ldots, \mathcal{C}_n)$, then

$$\mathsf{beh}_{\mathcal{C}}(init) = \overline{op}(\mathsf{beh}_{\mathcal{C}_1}(init_1), \ldots, \mathsf{beh}_{\mathcal{C}_n}(init_n)) \tag{V.9}$$

where $init$ (resp. $init_i, i = 1, \ldots, n$) is the initial state of $\mathcal{C}$ (resp. $\mathcal{C}_i$) and $\overline{op}$ is the adaptation of op on sets of transfer functions. Before establishing Equation V.9, we first need to define complex operators $\overline{op}$ on behaviours. Components' behaviours being sets of transfer functions, $\overline{op}$ has to be defined on a set of transfer functions. Moreover, it has to respect the same induction structure as op. We first have to adapt the cartesian product and the feedback on components' behaviours.

Definition 3.2 ***(Cartesian product on behaviours $\otimes_f$)***
Let $H_1 = T(\mathsf{Out}_1 \times \_)^{\mathsf{In}_1}$ and $H_2 = T(\mathsf{Out}_2 \times \_)^{\mathsf{In}_2}$ be two signatures. Let Γ_1 and Γ_2 be two sets of transfer functions over H_1 and H_2 respectively. Then, $\Gamma_1 \otimes_f \Gamma_2$ is the set:

$$\Gamma_1 \otimes_f \Gamma_2 = \{\mathcal{F}_1 \times \mathcal{F}_2 \mid \mathcal{F}_1 : {\mathsf{In}_1}^{\omega} \longrightarrow {\mathsf{Out}_1}^{\omega}, \mathcal{F}_2 : {\mathsf{In}_2}^{\omega} \longrightarrow {\mathsf{Out}_2}^{\omega}\}$$

It is obvious to prove that the cartesian product of two transfer functions is a transfer function.

Definition 3.3 ***(Relaxed feedback on transfer function)*** *Let $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$ be a signature and $\mathcal{I} = (f, \pi_i, \pi_o)$ be a feedback interface over H. Let $\mathcal{F} : \mathsf{In}^\omega \longrightarrow \mathsf{Out}^\omega$ be a transfer function. Let us define for every $x \in \mathsf{In}^\omega$, the couple $(\hat{x}, y_{\hat{x}}) \in \mathsf{In}^\omega \times \mathsf{Out}^\omega$ by induction on ω as follows:*

- *$\hat{x} = x(0)$ and $y_{\hat{x}}(0) = \mathcal{F}(x)(0)$*
- *$\forall n, 0 < n < \omega,\ \hat{x}(n) = f(x(n), y_{\hat{x}}(n-1))$, $y_{\hat{x}}(n) = \mathcal{F}(\overline{x})(n)$ where $\overline{x} \in \mathsf{In}^\omega$ is any dataflow such that $\forall j \leq n,\ \overline{x}(j) = \hat{x}(j)$.*

Then, $\hookleftarrow_{\mathcal{I}_f}(\mathcal{F}) : \mathsf{In}'^\omega \longrightarrow \mathsf{Out}'^\omega$ is the mapping that associates to $x' \in \mathsf{In}'^\omega, y' \in \mathsf{Out}'^\omega$ such that there exists $x \in \mathsf{In}^\omega$ satisfying:

$$\forall i < \omega, x'(i) = \pi_i(\hat{x}(i)) \text{ and } y'(i) = \pi_o(y_{\hat{x}}(i))$$

Let us observe that Definition 3.3 is noticeably similar to Definition 1.3 except that the choice of $y_{\hat{x}}(n)$ is unique in Definition 3.3 because directly giving by the transfer function $\mathcal{F}$.

$\hookleftarrow_{\mathcal{I}_f}(\mathcal{F})$ needs some conditions on projections π_i and π_o to be a transfer function. Indeed, π_i and π_o are surjective but by no means they are supposed to be injective. This can then question the causality conditions of $\hookleftarrow_{\mathcal{I}_f}(\mathcal{F})$. Imposing π_i and π_o to be injective would lead to condition which is too strong (π_i and π_o would then be bijective) and which is seldom satisfied (e.g. the sequential composition defined in Section 2.1). Here, we propose a weaker condition that is satisfied by most of the integration operators based on feedback (all those defined in this thesis).

Assumption 1: $\forall x_1, x_2 \in \mathsf{In}^\omega, \forall j, j \leq n, \pi_i(x_1(j)) = \pi_i(x_2(j)) \Longrightarrow$

$$\begin{cases} \pi_o(\mathcal{F}(x_1)(0)) = \pi_o(\mathcal{F}(x_2)(0)) & \text{if } j = 0 \\ \pi_o(\mathcal{F}(f(x_1(j), \mathcal{F}(\hat{x}_1)(j-1))) = \pi_o(\mathcal{F}(f(x_2(j), \mathcal{F}(\hat{x}_2)(j-1))) & \text{otherwise} \end{cases}$$

Proposition 3.1 *$\hookleftarrow_{\mathcal{I}_f}(\mathcal{F}) : \mathsf{In}'^\omega \longrightarrow \mathsf{Out}'^\omega$ is a transfer function.*

Proof *Let $\mathcal{F} : \mathsf{In}^\omega \longrightarrow \mathsf{Out}^\omega$ be a transfer function over H and $\hookleftarrow_{\mathcal{I}_f}(\mathcal{F}) : \mathsf{In}'^\omega \longrightarrow \mathsf{Out}'^\omega$ be the function defined in Definition 3.3. Let $x'_1, x'_2 \in \mathsf{In}'^\omega$ be two inputs dataflows for $\hookleftarrow_{\mathcal{I}_f}(\mathcal{F})$ and let us prove that if for every $n, 0 \leq n \leq \omega, x'_1(n) = x'_2(n)$, then $\hookleftarrow_{\mathcal{I}_f}(\mathcal{F})(x'_1(n)) = \hookleftarrow_{\mathcal{I}_f}(\mathcal{F})(x'_2(n))$.*

By induction over ω:

- ***Basic Step:*** *$n = 0$*

  By definition, $x'_1, x'_2 \in \mathsf{In}'^\omega$, then there exists $x_1, x_2 \in \mathsf{In}^\omega$ such that $x'_1(0) = \pi_i(x_1(0))$ and $x'_2(0) = \pi_i(x_2(0))$, and $\hookleftarrow_{\mathcal{I}_f}(\mathcal{F})(x'_1(0)) = \pi_o(\mathcal{F}(x_1(0)))$. By hypothesis, since $x'_1(0) = \pi_i(x_1(0))$ and $x'_2(0) = \pi_i(x_2(0))$, then $\pi_i(x_1(0)) = \pi_i(x_2(0))$. Then, by Assumption 1, we have that $\pi_o(\mathcal{F}(x_1)(0)) = \pi_o(\mathcal{F}(x_2)(0))$. Hence, $\hookleftarrow_{\mathcal{I}_f}(\mathcal{F})(x'_1(0)) = \pi_o(\mathcal{F}(x_2(0)))$ which by definition equals to $\hookleftarrow_{\mathcal{I}_f}(\mathcal{F})(x'_2(0))$.

- ***Induction Step:***

  By definition of $\hookleftarrow_{\mathcal{I}_f}(\mathcal{F})(x'_1(n+1))$, we know there exists $(\hat{x}_1, \mathcal{F}(\hat{x}_1)) \in \mathsf{In}^\omega \times \mathsf{Out}^\omega$ such that $\forall k, 1 \leq k \leq n+1,\ x'_1(k) = \pi_i(\hat{x}_1(k))$ and $\hookleftarrow_{\mathcal{I}_f}(\mathcal{F})(x'_1(k)) = \pi_o(\mathcal{F}(\hat{x}_1)(k))$ where $\hat{x}_1(k) = f(x(k), \mathcal{F}(\hat{x}_1)(k-1))$.

  By definition of $\hookleftarrow_{\mathcal{I}_f}(\mathcal{F})(x'_2(n+1))$, we know there exists $(\hat{x}_2, \mathcal{F}(\hat{x}_2)) \in \mathsf{In}^\omega \times \mathsf{Out}^\omega$ such that $\forall k, 1 \leq k \leq n+1,\ x'_2(k) = \pi_i(\hat{x}_2(k))$ and $\hookleftarrow_{\mathcal{I}_f}(\mathcal{F})(x'_2(k)) = \pi_o(\mathcal{F}(\hat{x}_2)(k))$ where $\hat{x}_2(k) = f(x(k), \mathcal{F}(\hat{x}_2)(k-1))$.

By hypothesis, we know that $\forall k, 0 \leq k \leq n,\ x'_1(k) = x'_2(k) \Longrightarrow \hookleftarrow_{\mathcal{I}_f}(\mathcal{F})(x'_1(k)) = \hookleftarrow_{\mathcal{I}_f}(\mathcal{F})(x'_2(k))$. *It remains to prove that if* $x'_1(n+1) = x'_2(n+1)$, *then* $\hookleftarrow_{\mathcal{I}_f}(\mathcal{F})(x'_1(n+1)) = \hookleftarrow_{\mathcal{I}_f}(\mathcal{F})(x'_2(n+1))$.
Since $\forall k, 1 \leq k \leq n+1,\ x'_1(k) = \pi_i(\hat{x}_1(k)), x'_2(k) = \pi_i(\hat{x}_2(k))$ *and* $x'_1(k) = x'_2(k)$, *then by Assumption 1,* $\forall k, 1 \leq k \leq n+1.\ \pi_o(\mathcal{F}(\hat{x}_1)(n+1)) = \pi_o(\mathcal{F}(\hat{x}_2)(n+1))$. *This last result then yield* $\hookleftarrow_{\mathcal{I}_f}(\mathcal{F})(x'_1(n+1)) = \hookleftarrow_{\mathcal{I}_f}(\mathcal{F})(x'_2(n+1))$.

End

Definition 3.4 ***(Well-formed feedback composition for transfer function)*** *Let* $\mathcal{I} = (f, \pi_i, \pi_o)$ *be a feedback interface over a signature* H. *Let* $\mathcal{F} : \mathsf{In}^\omega \longrightarrow \mathsf{Out}^\omega$ *be a transfer function. The synchronous feedback composition of* $\mathcal{F}$ *over* $\mathcal{I}$ *is* ***well-formed*** *if, and only if*

$$\forall x \in \mathsf{In}^\omega,\ (\forall n < \omega, \hat{x}(n) = f(x(n), \mathcal{F}(x)(n))) \Longrightarrow \mathcal{F}(\hat{x}) = \mathcal{F}(x)$$

Definition 3.5 ***(Synchronous feedback for transfer functions)*** *Let* $\mathcal{I} = (f, \pi_i, \pi_o)$ *be a feedback interface over a signature* H. *Let* $\mathcal{F} : \mathsf{In}^\omega \longrightarrow \mathsf{Out}^\omega$ *be a transfer function.* $\circlearrowleft_{\mathcal{I}_f}(\mathcal{F}) : \mathsf{In}'^\omega \longrightarrow \mathsf{Out}'^\omega$ *is the mapping that associates to* $x' \in \mathsf{In}'^\omega, y' \in \mathsf{Out}'^\omega$ *such that there exists* $x \in \mathsf{In}^\omega$ *satisfying*

$$\forall i < \omega, x'(i) = \pi_i(f(x(i), \mathcal{F}(x)(i)))\ \ and\ \ y'(i) = \pi_o(\mathcal{F}(f(x(i), \mathcal{F}(x)(i))))$$

Similarly to $\hookleftarrow_{\mathcal{I}_f}(\mathcal{F})$, $\circlearrowleft_{\mathcal{I}_f}(\mathcal{F})$ is a transfer function if the following assumption is satisfied by $\mathcal{F}$.

Assumption 2: $\forall x_1, x_2 \in \mathsf{In}^\omega, \forall j, j \leq n,$

$$\pi_i(x_1(j)) = \pi_i(x_2(j)) \Longrightarrow \pi_o(\mathcal{F}(f(x_1(j), \mathcal{F}(x_1)(j)))) = \pi_o(\mathcal{F}(f(x_2(j), \mathcal{F}(x_2)(j))))$$

Proposition 3.2 *$\circlearrowleft_{\mathcal{I}_f}$ is a transfer function.*

Proof *The technical proof is noticeably similar to the proof given for* $\hookleftarrow_{\mathcal{I}_f}$.
End

Let us note that both Assumption 1 and Assumption 2 are satisfied by all operators defined in this thesis that use the feedback operator. Indeed, if we take the sequential operator, we can see that these assumptions are verified according to the underlying feedback. This is also true for all other complex operators defined by the sequential operator.

Definition 3.6 ***(Feedback on behviours)*** *Let* Γ *be a set of transfer functions over a signature* $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$. *Then,* $\odot\Gamma$ *is the set of transfer functions:*

$$\odot\Gamma = \{\odot\mathcal{F} \mid \mathcal{F} : \mathsf{In}^\omega \longrightarrow \mathsf{Out}^\omega\}$$

where $\odot$ *is either* $\hookleftarrow_{\mathcal{I}_f}$ *or* $\circlearrowleft_{\mathcal{I}_f}$.

Complex operators can be easily extended to behaviours by replacing in Definition 2.1, the symbols $\otimes, \hookleftarrow_{\mathcal{I}}$ and $\circlearrowleft_{\mathcal{I}}$ by $\otimes_f, \hookleftarrow_{\mathcal{I}_f}$ and $\circlearrowleft_{\mathcal{I}_f}$, respectively. In the following, given a complex operator on components we will note $\overline{op}$ its equivalent on behaviours.

Similarly, Definition 3.1 can be easily extended to complex operators on behaviours by replacing each component $\mathcal{C}_i$ by a set of transfer functions Γ_i, and $\otimes, \hookleftarrow_{\mathcal{I}}$ and $\circlearrowleft_{\mathcal{I}}$ by $\otimes_f, \hookleftarrow_{\mathcal{I}_f}$ and $\circlearrowleft_{\mathcal{I}_f}$, respectively.

Theorm 3.1 *(Compositionality) Let op be a complex operator on components of arity n. Let $\mathcal{C}_1, \ldots, \mathcal{C}_n$ be n components. If $\mathcal{C} = op(\mathcal{C}_1, \ldots, \mathcal{C}_n)$, then*

$$\mathsf{beh}_{\mathcal{C}}(init) = \overline{op}(\mathsf{beh}_{\mathcal{C}_1}(init_1), \ldots, \mathsf{beh}_{\mathcal{C}_n}(init_n))$$

Proof *In order to prove this theorem, we need to prove the following lemmas:*

Lemma 3.1 *Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be two components over $H_1 = T(\mathsf{Out}_1 \times \_)^{\mathsf{In}_1}$ and $H_2 = T(\mathsf{Out}_2 \times \_)^{\mathsf{In}_2}$. Let $\mathcal{C} = \otimes(\mathcal{C}_1, \mathcal{C}_2)$ be the product component over $H = T((\mathsf{Out}_1 \times \mathsf{Out}_2) \times \_)^{\mathsf{In}_1 \times \mathsf{In}_2}$. Then we have:*

$$\mathsf{beh}_{\mathcal{C}_1 \otimes \mathcal{C}_2}((init_1, init_2)) = \mathsf{beh}_{\mathcal{C}_1}(init_1) \otimes_f \mathsf{beh}_{\mathcal{C}_2}(init_2)$$

Proof *By definition, $\mathsf{beh}_{\mathcal{C}_1 \otimes \mathcal{C}_2}((init_1, init_2))$ contains all the transfer functions $\mathcal{F} : (\mathsf{In}_1 \times \mathsf{In}_2)^\omega \longrightarrow (\mathsf{Out}_1 \times \mathsf{Out}_2)^\omega$ that associates to every $(x_1, x_2) \in \mathsf{In}_1 \times \mathsf{In}_2$, a $(y_1, y_2) \in \mathsf{Out}_1 \times \mathsf{Out}_2$ such that there exists an infinite sequence $((o_{11}, o_{21}), (s_{11}, s_{21})), \cdots \in (\mathsf{Out}_1 \times \mathsf{Out}_2) \times (S_1 \times S_2)$ satisfying:*

$$\forall j \geq 1, ((o_{1j}, o_{2j}), (s_{1j}, s_{2j})) \in \eta'_{(\mathsf{Out}_1 \times \mathsf{Out}_2) \times (S_1 \times S_2)}(\alpha((s_{1j-1}, s_{2j-1}))(x_1(j-1), x_2(j-1)))$$

with $(s_{10}, s_{20}) = (init_1, init_2)$, and for every $k < \omega, y_i(k) = o_i$ for $i = 1, 2$.
Hence, for $i = 1, 2$, there exists an infinite sequence $(o_{i1}, s_{i1}), \cdots \in \mathsf{Out}_i \times S_i$ satisfying

$$\forall j \geq 1, (o_{ij}, s_{ij}) \in \eta'_{\mathsf{Out}_i \times S_i}(\alpha_i(s_{ij-1})(x_i(j-1)))$$

We can then define a transfer function $\mathcal{F}_i : x_i \mapsto y_i$. Hence $\mathcal{F} = \mathcal{F}_1 \otimes_f \mathcal{F}_2$ and then

$$\mathcal{F} \in \mathsf{beh}_{\mathcal{C}_1}(init_1) \otimes_f \mathsf{beh}_{\mathcal{C}_2}(init_2)$$

By following the same reasoning, we can show that given

$$\mathcal{F}_i \in \mathsf{beh}_{\mathcal{C}_i}(init_i), \mathcal{F}_1 \otimes_f \mathcal{F}_2 \in \mathsf{beh}_{\mathcal{C}_1 \otimes \mathcal{C}_2}((init_1, init_2))$$

End

Lemma 3.2 *Let $\mathcal{C}'$ be a component over $H = T(\mathsf{Out}' \times \_)^{\mathsf{In}'}$ and $\mathcal{C} = \hookleftarrow_{\mathcal{I}}(\mathcal{C}')$ be a component over $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$. Let $\mathcal{I} = (f, \pi_i, \pi_o)$ where $f : \mathsf{In}' \times \mathsf{Out}' \longrightarrow \mathsf{In}', \pi_i : \mathsf{In}' \longrightarrow \mathsf{In}$ and $\pi_o : \mathsf{Out}' \longrightarrow \mathsf{Out}$ be a feedback interface. Then we have:*

$$\mathsf{beh}_{\hookleftarrow_{\mathcal{I}}(\mathcal{C}')}(init) = \hookleftarrow_{\mathcal{I}_f}(\mathsf{beh}_{\mathcal{C}'}(init'))$$

where init is the initial state of $\mathcal{C} = \hookleftarrow_{\mathcal{I}}(\mathcal{C}')$.

Proof *Le $\mathcal{F} \in \mathsf{beh}_{\hookleftarrow_{\mathcal{I}}(\mathcal{C}')}(init)$. By definition, $\mathcal{F}$ associates to $x' \in \mathsf{In}^\omega$, $y' \in \mathsf{Out}^\omega$ (when such y' exists) such that there exists $x \in \mathsf{In}'^\omega$ and $(\hat{x}, y_{\hat{x}}) \in \mathsf{In}'^\omega \times \mathsf{Out}'^\omega$ satisfying*

$$\forall i < \omega, x'(i) = \pi_i(\hat{x}(i)) \text{ and } y'(i) = \pi_o(y_{\hat{x}}(i))$$

By definition of $\hat{x}$ and $y_{\hat{x}}$, there exists an infinite sequence $(init', s'_1, \ldots, s'_k, \ldots) \in S'$ such that:

- *$\hat{x} = x(0)$ and $y_{\hat{x}}(0) \in \eta'_{\mathsf{Out}' \times S'}(\alpha'(init')(\hat{x}(0)))$*
- *$\forall n, 0 < n < \omega,\ \hat{x}(n) = f(x(n), y_{\hat{x}}(n-1)),\ y_{\hat{x}}(n) \in \eta'_{\mathsf{Out}' \times S'}(\alpha'(s'_n)(\hat{x}(n)))$.*

Hence, we can extract a transfer function $\mathcal{F}'$ that associates to $\hat{x}$, $y_{\hat{x}}$ such that $\leftarrow\!\!\!\!\!\!\rightarrow_{\mathcal{I}_f}(\mathcal{F}') = \mathcal{F}$, and then $\leftarrow\!\!\!\!\!\!\rightarrow_{\mathcal{I}_f}(\mathcal{F}') \in \leftarrow\!\!\!\!\!\!\rightarrow_{\mathcal{I}_f}(\mathsf{beh}_{\mathcal{C}'}(init'))$.

To prove the other inclusion, we can follow the same reasoning.
End

Lemma 3.3 *Let $\mathcal{C}'$ be a component over $H = T(\mathsf{Out}' \times \_)^{\mathsf{In}'}$ and $\mathcal{C} = \circlearrowleft_{\mathcal{I}}(\mathcal{C}')$ be a component over $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$. Let $\mathcal{I} = (f, \pi_i, \pi_o)$ where $f : \mathsf{In}' \times \mathsf{Out}' \longrightarrow \mathsf{In}'$, $\pi_i : \mathsf{In}' \longrightarrow \mathsf{In}$ and $\pi_o : \mathsf{Out}' \longrightarrow \mathsf{Out}$ be a feedback interface. Then we have:*

$$\mathsf{beh}_{\circlearrowleft_{\mathcal{I}}(\mathcal{C}')}(init) = \circlearrowleft_{\mathcal{I}_f}(\mathsf{beh}_{\mathcal{C}'}(init'))$$

where init is the initial state of $\mathcal{C} = \circlearrowleft_{\mathcal{I}}(\mathcal{C}')$.

Proof *The technical proof is noticeably similar to the proof given for $\leftarrow\!\!\!\!\!\!\rightarrow_{\mathcal{I}}$.*
End

Now, Theorem 3.1 is proven by induction on the structure of op as follows:

- ***Basic Step:*** *op is of the form $\_$. Its equivalent for sets of transfer functions is also defined by $\_(\Gamma) = \Gamma$. The conclusion is then obvious.*
- ***Induction Step:*** *Three cases have to be considered*
  - *$op = \otimes(op_1, op_2)$ with arity of op_1 is n_1, arity of op_2 is n_2 and $n_1 + n_2 = n$*

    By induction hypothesis, we have:

    (1) $\mathsf{beh}_{op_1(\mathcal{C}_1,\ldots,\mathcal{C}_{n_1})}(init) = \overline{op}_1(\mathsf{beh}_{\mathcal{C}_1}(init_1), \ldots, \mathsf{beh}_{\mathcal{C}_{n_1}}(init_{n_1}))$ where init is the initial state of $op_1(\mathcal{C}_1, \ldots, \mathcal{C}_{n_1})$.

    (2) $\mathsf{beh}_{op_2(\mathcal{C}_{n_1+1},\ldots,\mathcal{C}_n)}(init') = \overline{op}_2(\mathsf{beh}_{\mathcal{C}_{n_1+1}}(init_{n_1+1}), \ldots, \mathsf{beh}_{\mathcal{C}_n}(init_n))$ where $init'$ is the initial state of $op_2(\mathcal{C}_{n_1+1}, \ldots, \mathcal{C}_n)$.

    and by the definition of both op_1 and op_2, we have

    (3) $op_2(\mathcal{C}_{n_1+1}, \ldots, \mathcal{C}_n)$ and $op_2(\mathcal{C}'_{n_1+1}, \ldots, \mathcal{C}'_n)$ are components.

    Then, ((1) + (2) + (3) + Lemma 3.1 implies that
    $\mathsf{beh}_{op_1(\mathcal{C}_1,\ldots,\mathcal{C}_{n_1}) \otimes op_2(\mathcal{C}_{n_1+1},\ldots,\mathcal{C}_n)}((init, init')) =$
    $\overline{op}_1(\mathsf{beh}_{\mathcal{C}_1}(init_1), \ldots, \mathsf{beh}_{\mathcal{C}_{n_1+1}}(init_{n_1})) \otimes_f \overline{op}_2(\mathsf{beh}_{\mathcal{C}_{n_1+1}}(init_{n_1+1}), \ldots, \mathsf{beh}_{\mathcal{C}_n}(init_n))$
  - *op is of the form $\leftarrow\!\!\!\!\!\!\rightarrow_{\mathcal{I}}(op')$ and is of arity n.*

    Let $\mathcal{C}_1, \ldots, \mathcal{C}_n$ be n components such that $\mathcal{C}' = op'(\mathcal{C}_1, \ldots, \mathcal{C}_n)$. By induction hypothesis, $\mathsf{beh}_{\mathcal{C}'}(init') = \overline{op}'(\mathsf{beh}_{\mathcal{C}_1}(init_1), \ldots, \mathsf{beh}_{\mathcal{C}_n}(init_n))$. It remains to prove that

    $$\mathsf{beh}_{\leftarrow\!\!\!\!\!\!\rightarrow_{\mathcal{I}}(\mathcal{C}')}(init) = \leftarrow\!\!\!\!\!\!\rightarrow_{\mathcal{I}_f}(\mathsf{beh}_{\mathcal{C}'}(init'))$$

    where init is the initial state of $\mathcal{C} = \leftarrow\!\!\!\!\!\!\rightarrow_{\mathcal{I}}(\mathcal{C}')$. This last point is naturally proved by Lemma 3.2.
  - *op is of the form $\circlearrowleft_{\mathcal{I}}(op')$ and is of arity n.*

    Let $\mathcal{C}_1, \ldots, \mathcal{C}_n$ be n components such that $\mathcal{C}' = op'(\mathcal{C}_1, \ldots, \mathcal{C}_n)$. By induction hypothesis, $\mathsf{beh}_{\mathcal{C}'}(init') = \overline{op}'(\mathsf{beh}_{\mathcal{C}_1}(init_1), \ldots, \mathsf{beh}_{\mathcal{C}_n}(init_n))$. It remains to prove that

    $$\mathsf{beh}_{\circlearrowleft_{\mathcal{I}}(\mathcal{C}')}(init) = \circlearrowleft_{\mathcal{I}_f}(\mathsf{beh}_{\mathcal{C}'}(init'))$$

    where init is the initial state of $\mathcal{C} = \circlearrowleft_{\mathcal{I}}(\mathcal{C}')$. This last point is naturally proved by Lemma 3.3.

End

4 Related works

In this section, we present a brief overview of contributions which are technically close to our approach, by discussing the difference between the problematics addressed by those contributions and those addressed by our approach. There are several coalgebraic works in the literature which regard the combination of components using some sort of integration mechanism. The closet to our work is the set of integration operators proposed by *Barbosa* in [9, 31]. Four component integration operators have been proposed to reason about component-based designs: pipeline "series" operator, external choice operator, parallel composition and concurrent operator. These operators are defined as special functors in some bicategory of components. The pipeline operator is similar to our synchronous sequential operator $\triangleright_r$. The external choice operator corresponds to a composition where both components $\mathcal{C}_1$ and $\mathcal{C}_2$ are executed independently, depending on the input submitted to the integrated component: when interacting with the composed system, the environment will be allowed to choose either $\mathcal{C}_1$ or $\mathcal{C}_2$ inputs, but not both. Input then triggers the corresponding component (i.e. $\mathcal{C}_1$ or $\mathcal{C}_2$), producing the associated output. This operator is then similar to our synchronous product $\circledast$ when the intersection of input sets In_1 and In_2 is the empty set. The parallel composition is embodied in the cartesian product, and finally the concurrent operator is similar to the operator defined in Section 2.4. Thus, *Barbosa*'s operators can be all deduced from our two basic operators by choosing the suitable and the minimal combination of them. In this setting, our approach offers advantages relative to [9, 31], that is it makes larger systems as a composition of only two operators, rather than as a combination of a set of separated operators. This makes it easier to reason about system functionalities in the sense that every correct property under these basic operators is also a correct property under other complex operators. This yields general theorems that are true of all operators defined in our frameworks, and thus it is not required to be re-proved every time for a new operator described as a combination of our basic operators: the proofs of results done on our basic operators are made once and for all.

Meng in [89] redefined *Barbosa*'s operators to combine two components $\mathcal{C}_1$ and $\mathcal{C}_2$ over the signatures $(\mathsf{Out}_1 \times T(\mathsf{Out}_2 \times \_)^{\mathsf{In}})$ and $(\mathsf{Out}'_1 \times T'(\mathsf{Out}'_2 \times \_)^{\mathsf{In}'})$ respectively. Hence, the difference between *Meng*'s and *Barbosa*'s work is the form of the functor over which components are defined, and the possibility of combining components with different computational models (i.e. T and T'), rather than using a single monad.

In this chapter, we have also shown how to define larger systems by composing subsystems from two basic integration operators: product and feedback. This led us to inductively define a set of complex operators (see Definition 2.1), the semantics of which are partial functors on categories of components. This part can then be compared to works in [90, 91]. Indeed, from a set of complex operators we can easily generate an algebraic signature that can be seen as an FP-theory $\mathbb{L}$ over a basic set of sorts $S \subseteq \mathbf{Set} \times \mathbf{Set}$ where for $(\mathsf{In}, \mathsf{Out}) \in S$, In and Out denote input and output sets, respectively, and operations are complex operators (a monad T is supposed identical for every couple $(\mathsf{In}, \mathsf{Out})$ in the FP-theory $\mathbb{L}$). Outer models can then be defined along the functor $\mathbb{C} : \mathbb{L} \longrightarrow \mathbf{Cat}$ that associates to any couple $(\mathsf{In}, \mathsf{Out})$ the category $\mathbf{Comp}(H)$ with $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$ and to any operator the partial functor defined in Definition 2.1. Finally, inner models are defined by the natural transformation $X : \mathbf{1} \Longrightarrow \mathbb{C}$ where $\mathbf{1}$ is the constant functor that associates to any $S \in \mathbb{L}$ the trivial object category $\mathbf{1}$, which to any couple $(\mathsf{In}, \mathsf{Out})$ associates the final object in $\mathbf{Comp}(H)$ and to any complex operator op, the mapping on behaviours noted $[[op]]$ in [90, 91] that contains op semantics on both components and transfer functions.

The difference between our works and those mentioned above is that we have defined integration operations by composing two basic operators: product and feedback. The objective was then to demonstrate a set of general properties on these integration operators such as the

results of compositionality (see Theorem 3.1), by showing that these properties are valid for the product and feedback and are preserved by composition.

Hence, Theorem 3.1 is similar to Theorem 4.7 in [91] at least in these goals to establish a generic result of compositionality independent of a given integration operator.

There is a long list of other works addressing components composition without involving coalgebraic denotations. It is difficult to collect them all; however, the common basis of most of them is that the system is described as a structural decomposition into components (or subsystems) by separating the notion of component behaviour and that of interaction (or communication) between components, which is considered essential to overcome system design complexity [2, 3, 92]. Approaches based on so-called *model of computations*, such as [92, 93, 94, 95, 96, 97], have been proposed to connect all components of the system globally. Roughly speaking, a model of computation can be seen as a set of primitive rules that govern the interactions between the components of a system. Such rules should explicitly encompass (1) the global behaviour of a system during its execution (e.g. cyclic, reactive, concurrent or sequential way); (2) the communication protocol used to allow the system to interact with its environment (e.g. rendezvous, message passing, exchange events) and (3) the data format that can be used during communication (e.g. events, queries, flux). These approaches suffer from the disadvantage that the semantics of the model of computations used and the interactions between them are given implicitly, which makes it hard or even impossible to incorporate, beyond modeling and simulation, other techniques such as testing and verification.

Other approaches have been proposed for describing component behaviour and their coordinations, such as [98, 99, 100]. Reo [98] is a coordination language for the composition of distributed software components and services based on connectors. Primitive connectors such as synchronous channels or $FIFO$ queues are structured to build complex component connectors which exhibit complex behaviours. A number of formal models describing the behaviour of Reo connectors and their composition exists, for instance models based on constraint automata [101] or models based on coalgebraic denotations [102]. Thus, algebraic reasoning and simulation are supported for analysis. Reo does not focus on component behaviour, it presents components only by their interfaces. In this way, Reo's components can be seen as transfer functions taking input data at given moments and providing its associated output, and Reo's connectors as relations between a couple of an input data stream and a time stream $(\mathsf{In}^\omega, TS^\omega)$ and a couple of an output data stream and a time stream $(\mathsf{Out}^\omega, TS^\omega)$. Thereby, associating time to component dataflows makes Reo's connectors rich, where by imposing suitable timing constraints on them, many styles of communication can be obtained such as synchronous, asynchronous, bounded, unbounded, lossy, lossless, etc. Thus, Reo's connector can be thought of as the sequential operator defined in Section 2.1, which does not only make components connection synchronously, but also encompasses other connection aspects such as asynchronous, unbounded, etc. Hence, by extending In and Out of the[13] signature $T(\mathsf{Out} \times \_)^{\mathsf{In}}$ with a complex data structure, Reo's connectors would be embodied in our framework.

[99, 100] provides a formal composition framework for describing based-component systems with heterogeneous interactions. This framework is known as BIP: B stands for Behaviour, I for Interaction and P for Priority. Thus, as its name indicates, a BIP model (or component) is composed of three layers: a layer defines the behaviour of the component encoded as transition systems extended with variables, a layer defines the connectors (i.e. the interactions) between components executed via communication ports and a layer defines priority rules which reduce non-determinism between interactions. BIP's models can be composed to yield larger models. This is done using a binary composition operator on components which is supposed to com-

13. Extension of the signature $T(\mathsf{Out} \times \_)^{\mathsf{In}}$ over which components are defined with data structure is a part of our future work.

pose layers separately. This means that when composing two components $\mathcal{C}_1$ and $\mathcal{C}_2$, their corresponding layers are composed indifferently and separately: $\mathcal{C}_1$'s behaviour is composed with $\mathcal{C}_2$'s, $\mathcal{C}_1$'s interactions are composed with $\mathcal{C}_2$'s, and $\mathcal{C}_1$'s priority rules are composed with $\mathcal{C}_2$'s. Furthermore, BIP is able to ensure correctness-by-construction for essential system properties such as mutual exclusion, deadlock freedom and progress. It also enables formal verification. Such a representation of components and connectors allows BIP to provide multitude heterogeneous interactions such as rendezvous and broadcast communication mechanisms. We believe that these styles of communications would be described in our framework, by extending the component signature $T(\mathsf{Out} \times \_)^{\mathsf{In}}$ with complex data structure (for instance, rendez-vous style can be embodied in sequential communication, broadcast in sequential following by synchronous product).

5 Conclusion

In this chapter, we have proposed a generic framework for modeling complex modern systems viewed as state-based systems. We have shown how a basic component (see Chapter IV, Definition 1.1) can be combined with another by means of some integration operators to yield a bigger, more complicated component. We have then defined two generic integration operators for combining the behaviour of components: an extended version of the well-known cartesian product, and the feedback operator. The feedback operator relies on three mappings: a mapping f to specify how components are linked and which parts of their interfaces are involved in the composition process, and two mappings π_i and π_o that allow us to hide inputs and outputs involved in the feedback composition process and thus help both encapsulation and compositionality. We have shown that other integration operators such as the sequential operator, the double sequential operator, the synchronous product, the concurrent operator and the synchronous parallel composition operator can be considered as compositions of these two basic operators by a suitable choice of f, π_i and π_o. In this setting, we can see both the cartesian product and the feedback operator as two patterns from which other integration operators can be deduced as sequences of composition patterns. These patterns can also be used to build larger patterns. Our two basic operators are then minimal operators used to derive operators representing an interaction between components by choosing a suitable and a minimal combination. Such a combination is given according to the semantic that we want to associate to the integration operator of interest i.e. how the components communicate and share their actions. Hence, the objective is to demonstrate a set of general properties on these integration operators such as the results of compositionality, by showing that these properties are valid for the product and the feedback and are preserved by composition. Thereby, every correct property under the basic operators is also a correct property under complex operators.

Part III

Validation of component-based systems by testing

This part provides the second main contribution of this thesis. It presents a formal compositional theory for testing complex-software systems viewed as component-based systems. Our approach in the first part was to define a generic formalism dedicated to modeling systems, generally state-based formalisms. The work proposed here can then be seen as a proposal for a generic theory of conformance testing. It intends to contribute to the following topics:

– The development of a conformance testing approach enabling us to ensure correctness of the components defined in Chapter IV. This approach is mainly inspired from the theory of conformance testing developed by *A. Touil* and *al.* in [44, 53].

– The definition of a compositional testing framework enabling us to ensure the correctness of systems obtained as an assembling of a set of components, as was shown in Chapter V. The underlying idea consists in establishing correctness of the global system by using correctness of each component.

– The definition of a framework enabling us to strengthen the correctness of each component involved in a global system, by choosing suitable test purposes for them. The underlying idea is to use a projection mechanism, as in [48], to identify from any trace tr of the global system, the trace of any component involved in tr. These projected traces can be then seen as test cases that should be tested on individual components.

This part consists of three chapters. The first one presents an overview of the conformance testing theory. The second one introduces our conformance testing theory allowing us to test components separately. The third one is devoted to defining our compositional testing approach based on projection mechanisms.

Chapter VI

Conformance testing theory: a general overview

1 Formal Method in Conformance Testing 132

1.1 General principle 132

1.2 The meaning of conformance 133

1.3 Formal framework for conformance testing 134

Conformance testing [27, 26] is a technique for checking the functional correctness of an implementation with respect to its specification by means of experiments on the implementation. It consists in deriving test cases algorithmically from a system specification, executing them on the real system and finally making sure that the latter behaves correctly by comparing its outputs with those required in the specification.

A common methodology and framework for the specification and execution of conformance testing for implementations of standard communication protocols such as *ISO* protocols, *ISDN* protocols, etc. was proposed by the *International Organization for Standardization* (*ISO*). This methodology is known as the *international standard* IS-9646: "Conformance Testing Methodology and Framework" [50, 51]. The goal of the standard IS-9646 originally was to unify the process of developing methods for conformance testing between protocols or open systems interconnection (*OSI*) and their specifications. But, it has rapidly turned out that this standard can also be suitable to test reactive systems [103, 104]. It is consequently now considered as the basis of conformance testing where it allows us to define how to specify conformance tests and to provide guidance to developers of test systems.

Nevertheless, the standard IS-9646 is limited in practice to automatized testing due to the absence of a formal description of its elements. Concepts are indeed written in natural language which makes the automation of the conformance testing process hard. Hence, this limitation activated the research and development of a formal framework, in which conformance testing concepts, such as conformance requirement, conformance meaning, correctness of an implementation, test cases, test execution, verdict, etc. were defined in a formal setting. This led to a joint project between *ISO* and the *International Telecommunication Union* (*ITU*) called "Formal Methods in Conformance Testing" (*FMCT*) [52]. This project was the main topic of *Tretmans*'s thesis [27] whose goal was the formalization of the testing methodology IS-9646, giving a formal interpretation to most concepts in this standard.

In this chapter, we outline the main concepts of conformance testing which are introduced in the standard IS-9646, and formalized by *Tretmans* in [27]. These concepts will be the theoretical background of formal testing, which serves us as a fundamental basis to define our method for automatic generation of abstract [1] tests for generic components.

1 Formal Method in Conformance Testing

1.1 General principle

The process of conformance testing consists mainly of three phases that are shown in Figure VI.1.

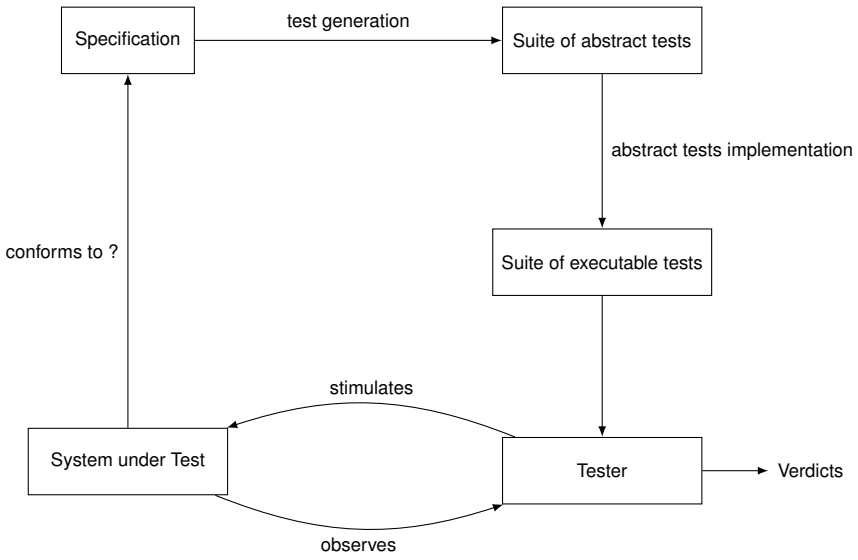


Figure VI.1 – Conformance testing process

The first phase is called *test generation* (or *test derivation*) and consists in deriving *test cases* algorithmically from a formal description of the system's behaviour using an algorithm of test case generation. A collection of generated test cases is called a *test suite*. Test cases are indeed not directly executable on the implementation because they are developed at some abstract level independently of any implementation of the system. Thus, they are called *abstract tests*. The second phase is known as *test implementation* and consists in transforming the generated test cases during the first phase into concrete tests which therefore can be run on the real system. In fact, a mechanism that maps abstract tests to concrete ones is needed. This mechanism is usually referred to as an *adapter*. The last phase is called *test execution* and consists in executing concrete tests on a particular implementation. The underlying idea is to provide test cases to the implementation and then observe its outputs which should be compared with the expected outputs indicated in the test case. If the outputs do not match the specified outputs, a verdict Fail is assigned to the test indicating non-conformance between the implementation and the specification. Otherwise, a verdict Pass is assigned to the test indicating the conformance between the implementation and the specification (only for the submitted test case).

1. It is important to notice that all concepts introduced in this chapter are given at a generic level, i.e. they are independent of any particular formal method.

Note that in this thesis, we will not talk about the phase of test implementation. In fact, as the test cases generated from the first phase are abstract, they must be then made concrete before executing them on the implementation. As well, when a test is executed, outputs from the implementation, being concrete, tests have to be transformed into abstract tests before comparing them with the expected outputs that are specified in the abstract test cases. We will not focus on this phase, and as you will see in the next section, we consider implementations as black boxes in order to deal with them by a formal reasoning.

1.2 The meaning of conformance

1.2.1 Specification model

A *formal specification* describes system behaviour using a specialized description formalism. In general, it is a formal representation that captures the properties of a system precisely and unambiguously.
In the following part of this section, to be independent of any specification formalism, we will note SPECS the set of all possible formal specifications independently of any formalism, and spec a specification belonging to SPECS.

1.2.2 Implementation model

An *implementation under test* generally consists of a combination of hardware and software. It usually has physical connectors or interfaces to communicate with its environment. As previously, we will note the set of all possible implementations by IMPS and an implementation (for instance Java program or hardware components) belonging to IMPS by iut. An implementation is then a concrete executable object we cannot deal with it by a formal reasoning. The only way to observe its behaviour is to interact via its interfaces, submitting inputs and observing outputs. Hence, a formal description for such an implementation is needed to build a conformance testing theory whose aim is to check whether the behaviour of a real implementation is correct with respect to a formal specification. Hence, every implementation $\mathsf{iut} \in \mathsf{IMPS}$ has to be modeled by a formal object $\mathsf{m_{iut}}$ called a *model of* iut. The universe of the models of all implementations under test is denoted by MODS. Consequently, we have the following *testing hypothesis* [105]:

$$\forall \mathsf{iut} \in \mathsf{IMPS}, \exists \mathsf{m_{iut}} \in \mathsf{MODS},\ \forall i \in \mathsf{In}, Out(\mathsf{iut}, i) = Out(\mathsf{m_{iut}}, i)$$

where $Out(\mathsf{iut}, i)$ (respectively $Out(\mathsf{m_{iut}}, i)$) is the output yielded by iut (respectively $\mathsf{m_{iut}}$) for the input i.

Note that it is not assumed that the model of an implementation is known, only its existence is required.

1.2.3 Conformance relation

The theory of conformance testing defines the conformance of an implementation to a specification thanks to conformance relations. The objective of these relations is to provide a way to specify conformance of an implementation with its specification. Several kinds of relations have been proposed according to both test purposes and application domains. For instance, models described by specialized description languages such as LOTOS [106] or SDL [107], or those directly described as operational formalisms such as finite state machine or labeled transition systems have different implementation relations embodying the conformance.

The conformance is formally expressed as a relation between the class of implementations IMPS and the class of specifications SPECS. This relation, the so-called *implementation relation*, is denoted by $\mathsf{imp} \subseteq \mathsf{MODS} \times \mathsf{SPECS}$ and expressed as follows:

An implementation $\mathsf{iut} \in \mathsf{IMPS}$ *is in conformance to a specification* $\mathsf{spec} \in \mathsf{SPECS}$ *if the existing model* $\mathsf{m_{iut}} \in \mathsf{MODS}$ *of* iut *is* imp*-related to* spec*, i.e.* iut *conforms to* spec *iff* $\mathsf{m_{iut}}$ imp spec.

A specification can have many implementations conforming to it and then many conforming implementation models. The different relations between IMPS, MODS and SPECS are depicted in Figure VI.2. For a specification $\mathsf{spec} \in \mathsf{SPECS}$ and an implementation relation imp, one has the set $\mathsf{I_{iut}} \subseteq \mathsf{IMPS}$ of all implementations that can be modeled by models in $\mathsf{M_{iut}} \subseteq \mathsf{MODS}$. Therefore, $\mathsf{I_{iut}}$ represents the set of all implementations that implement the specification spec correctly according to imp, and $\mathsf{M_{iut}}$ represents the set of all models that conform to spec in MODS. Thus, $\mathsf{M_{iut}}$ is given by $\{\mathsf{m} \in \mathsf{MODS} \mid \mathsf{m\ imp\ spec}\}$. Hence, an implementation $\mathsf{iut} \in \mathsf{IMPS}$ conforms to the specification $\mathsf{spec} \in \mathsf{SPECS}$ if it is modeled by $\mathsf{m_{iut}}$ belonging to the set $\mathsf{M_{iut}}$, and the model $\mathsf{m_{iut}}$ implements spec according to the implementation relation imp.

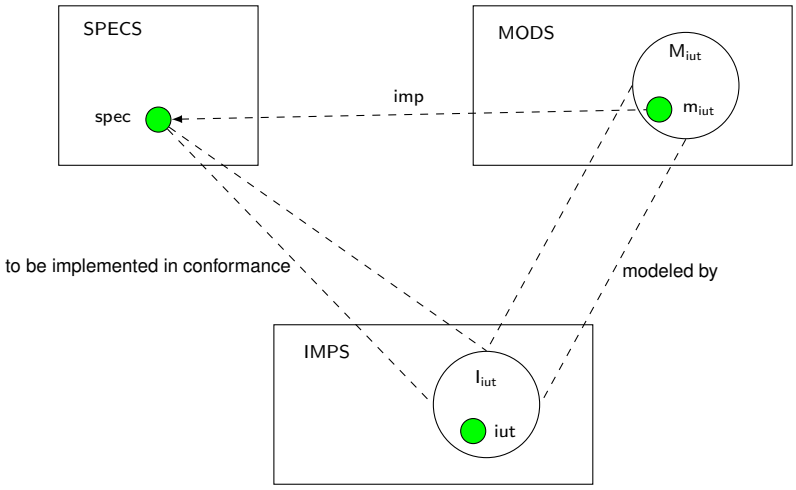


Figure VI.2 – Relations between IMPS, MODS and SPECS

1.3 Formal framework for conformance testing

1.3.1 Test execution

The behaviour of an implementation under test $\mathsf{iut} \in \mathsf{IMPS}$ is checked by means of test experiments on it: we choose an appropriate input and submit it to the iut. Then, we observe the reaction of the iut and compare it with the one expected in its specification $\mathsf{spec} \in \mathsf{SPECS}$. This comparison gives rise to a verdict about the correctness of the concrete implementation iut with respect to the specification spec. A specification of each such experiment is called a *test case*. The set of all possible test cases is denoted by TESTS and a test case belonging to TESTS by tc.

When specifications are written in formal languages, test cases can then be generated automatically using algorithms for test case derivation from the specification. Given an implementation relation imp and an implementation under test $\mathsf{iut} \in \mathsf{IMPS}$, an algorithm of test case generation then provides a set of test cases $\mathsf{TC} \subseteq \mathsf{TESTS}$ for a specification $\mathsf{spec} \in \mathsf{SPECS}$ of iut.

This process of derivation of tests is known as *test case generation*. An execution of a test case is then the process of stimulating an implementation under test $\mathsf{iut} \in \mathsf{IMPS}$ by:

- executing the specified test events of a test case $\mathsf{tc} \in \mathsf{TESTS}$;
- observing the produced reactions from the given iut;
- generating a test verdict based on these reactions.

We define $\mathsf{exec}(\mathsf{tc}, \mathsf{iut})$ as the concrete execution of a test case on a real implementation leading to a subset of observations. Now, to continue our formal reasoning, we need to formalize $\mathsf{exec}(\mathsf{tc}, \mathsf{iut})$, then we introduce the observation function:

$$\mathsf{obs} : \mathsf{TESTS} \times \mathsf{MODS} \longrightarrow \mathcal{P}(\mathsf{OBS})$$

where OBS denotes the domain of all possible observations from the iut.

This function computes, for each test case $\mathsf{tc} \in \mathsf{TESTS}$ and each concrete implementation under test $\mathsf{iut} \in \mathsf{IMPS}$ modeled by $\mathsf{m}_{iut} \in \mathsf{MODS}$, the observations in OBS that result from executing tc on iut i.e. a subset of observations $\mathsf{O} \subseteq \mathsf{OBS}$. Hence, $\mathsf{obs}(\mathsf{tc}, \mathsf{m}_{\mathsf{iut}})$ models the execution of test cases $\mathsf{exec}(\mathsf{tc}, \mathsf{iut})$ formally.

Each observation obtained after a test execution is assigned to a verdict which may vary depending on the test case tc. This gives rise to a function:

$$\mathsf{verdict}_{\mathsf{tc}} : \mathsf{OBS} \longrightarrow \{\mathsf{Fail}, \mathsf{Pass}\}$$

A Fail verdict means that a non-conformance between the implementation iut and its specification spec is detected (the behaviour of the implementation observed by the tester is not the one expected in the specification). A Pass verdict means that the implementation iut behaves in conformance to the specification spec for the given observation i.e. for the executed test case. [2]

1.3.2 Test case properties

So far we have only seen how test cases are generated from specification and how they are executed on implementations. But, we have not studied the coherence between both notions of test cases execution and conformance applied to an implementation under test and its formal specification. For this purpose, two properties have to be studied: *correctness* and *completeness*.

- **Correctness**: this property states that if an implementation iut is in conformance to a specification spec then it passes all generated tests.
- **Completeness**: this property states that if the implementation passes all generated test cases, then it conforms to its specification.

Intuitively the correctness property is achievable for practical testing. It allows us to ensure the conformance of an implementation with respect to its specification, but it may not be able to detect implementations that are not in conformance to a specification. That is because some non-conforming implementations may pass the set of test cases. On the contrary, the completeness property allows us to exactly distinguish between all conformant and non-conformant implementations. However since a test is not exhaustive, this means we cannot reach the set of all tests generated from the specification, this property is only achievable in theory.

In order to formalize the property of correctness and completeness, we introduce the following two notations:

2. There exist another verdicts. For instance, the inconclusive verdict, noted Inconc, that is used if the implementation iut behaves correctly according to the specification spec, but its responses do not satisfy the test purpose.

– *passes*: means that an implementation iut modeled by $\mathsf{m_{iut}} \in \mathsf{MODS}$ passes a test case tc successfully. This is formalized by:

$$\mathsf{iut}\ \ \textit{passes}\ \ \mathsf{tc} \quad =_{\text{def}} \quad \mathsf{verdict}(\mathsf{obs}(\mathsf{tc}, \mathsf{m_{iut}})) = \mathsf{Pass}$$

– $fails$: means that an implementation iut modeled by $\mathsf{m_{iut}} \in \mathsf{MODS}$ does not pass a test case tc. This is a situation of failure.

$$\mathsf{iut}\ \ \textit{fails}\ \ \mathsf{tc} \quad =_{\text{def}} \quad \neg(\mathsf{iut}\ \ \textit{passes}\ \ \mathsf{tc})$$

These notions can be extended to a set of test cases $\mathsf{TC} \subseteq \mathsf{TESTS}$ as follows:

$$\mathsf{iut}\ \ \textit{passes}\ \ \mathsf{TC} \quad =_{\text{def}} \quad \forall \mathsf{tc} \in \mathsf{TC}, \mathsf{iut}\ \ \textit{passes}\ \ \mathsf{tc}$$

$$\mathsf{iut}\ \ \textit{fails}\ \ \mathsf{TC} \quad =_{\text{def}} \quad \exists \mathsf{tc} \in \mathsf{TC}, \mathsf{iut}\ \ \textit{fails}\ \ \mathsf{tc}$$

Definition 1.1 ***(Correctness and Completeness)*** *Let* $\mathsf{iut} \in \mathsf{IMPS}$ *be an implementation and* $\mathsf{spec} \in \mathsf{SPECS}$ *be its specification. Let* TC *be a set of generated test cases. Then we have:*

– TC *is* ***correct*** *if:*

$$\forall \mathsf{iut} \in \mathsf{IMPS},\ \mathsf{iut}\ \ \mathsf{imp}\ \ \mathsf{spec} \quad \Longrightarrow \quad \mathsf{iut}\ \ \textit{passes}\ \ \mathsf{TC}$$

– TC *is* ***complete*** *if:*

$$\forall \mathsf{iut} \in \mathsf{IMPS},\ \mathsf{iut}\ \ \mathsf{imp}\ \ \mathsf{spec} \quad \Longleftrightarrow \quad \mathsf{iut}\ \ \textit{passes}\ \ \mathsf{TC}$$

The following table describes the required elements for conformance testing:

Physic elements	implementation	$\mathsf{iut} \in \mathsf{IMPS}$
	test case execution	$\mathsf{exec}(\mathsf{tc}, \mathsf{iut})$
Formal elements	specification	$\mathsf{spec} \in \mathsf{SPECS}$
	implementation model	$\mathsf{m_{iut}} \in \mathsf{MODS}$
	implementation relation	$\mathsf{imp} \subseteq \mathsf{MODS} \times \mathsf{SPECS}$
	test case	$\mathsf{tc} \in \mathsf{TESTS}$
	observations	OBS
	execution model of test case	$\mathsf{obs} : \mathsf{TESTS} \times \mathsf{MODS} \longrightarrow \mathcal{P}(\mathsf{OBS})$
	verdict	$\mathsf{verdict_{tc}} : \mathsf{OBS} \longrightarrow \{\mathsf{Fail}, \mathsf{Pass}\}$
Assumptions	test hypothesis	$\mathsf{m_{iut}}$ models iut
		$\mathsf{obs}(\mathsf{tc}, \mathsf{m_{iut}})$ models $\mathsf{exec}(\mathsf{tc}, \mathsf{iut})$

Table VI.1 – Conformance testing elements

Chapter VII

Testing of components

1 Conformance relation 138
1.1 Specification model 138
1.2 Implementation model 138
1.3 Conformance 139
2 Finite computation tree 144
2.1 Formal definition 144
2.2 Unfolding algorithm 146
3 Test Purpose 149
4 Test generation guided by test purposes 150
4.1 Preliminaries 153
4.2 Inferences rules 154
4.3 Example 156
4.4 Properties 158
5 Instantiating of the approach 161

In this chapter, we develop a formal theory of conformance testing which allows us to test our components defined in the first part of this thesis. The work presented here is mainly inspired by the formal testing theory developed by *A. Touil* et *al.* in [44, 53]. In our conformance testing theory, behaviours of specifications and implementations under test are considered as coalgebras of the functor $T(\mathsf{Out} \times \_)^{\mathsf{In}}$ presented in Chapter IV. The conformance relation we consider is an adapted version of the well-known relation *ioco* proposed in [46], and that is defined as a partial inclusion of implementation traces into specification ones. Our work uses the notion of a test purpose as it was defined in [44, 53]. In [44, 53], a test purpose provides an operative way to extract test cases by selecting from the specification model, described [1] as an *IOSTS*, the functionalities that we want to test. This is done by relying on a symbolic execution technique. In our approach, since we do not handle data but just values in both In and Out, we use a simple unfolding technique to define our test purposes instead of a symbolic execution technique. Then, our test purposes are directly derived from specifications using unfolding and marking algorithms, and look like labeled trees capturing all specification traces for a given length. We also propose in this chapter a test case generation algorithm as in [44, 53]. The underlying idea consists in choosing a possible input i and submitting it to the implementation under test, and then observing the outputs produced from it and compare them with the possible outputs in the specification. Hence, the only way to observe the reaction of the system under

1. *IOSTS* stands for Input-Output Symbolic Transition system. It is a variant of *IOLTS* including and handling explicitly the data system [44, 53].

test is through sequences of simulations-observations. In other words, systems under test are considered as black boxes. In this sense, we are in a testing framework similar to both [44, 53] and [39, 108].

1 Conformance relation

In this section, we examine how we can define the conformance of the implementation of a component to its specification. In order to compare the behaviour of the implementation to the specification, we need to consider both as components over a same signature. However, the behaviour of the implementation is unknown and can only be observed through its interface. We therefore need a conformance relation between what we can observe on the implementation and what the specification allows.

To define the conformance between the implementation iut and its specification spec, a formal relation of conformance between iut and spec is classically given between the models of iut and spec. However, the specification spec of a system is the formal description of its behaviour. On the contrary, the implementation iut of a system is an executable component, which is considered as a black box [109, 110]. Hence, modeling iut requires some assumptions that we state in the following.

1.1 Specification model

The conformance testing is a black-box test technique i.e. it is only based on a description of system functionalities in terms of its inputs and outputs. Such a description does not make reference to the internal structure of a system under test. It only contains the desired behaviours that stand only for what the system should do, not how it is done. Then, the first step to define conformance testing theory is to give a specification model of the system in which both its input and output are well represented and the internal behaviour is not considered. Furthermore, this specification model has to make clear the difference between the input and the output of the system due to the fundamental role of this distinction in practical testing process [27, 37, 38, 45, 67]. Indeed, the inputs are the actions used by the tester to stimulate the system under test while the outputs are the expected reactions observed after the stimulation. From this point of view, our components defined over the signature $T(\mathsf{Out} \times \_)^{\mathsf{In}}$ give answers to these testing requirements where they explicitly differentiate input and output actions.

Definition 1.1 *(**Specification model**) A **specification** of a system $\mathcal{S}$ is modeled as a component* $\mathsf{spec} = (S, init, \alpha)$ *over a signature* $T(\mathsf{Out} \times \_)^{\mathsf{In}}$.

1.2 Implementation model

An implementation iut is commonly a reactive program intending to interact permanently with its environment. During the test process, it is assumed that the source code of the implementation is not available and null knowledge about it is provided. That means it is considered a black box [109, 110], whose internal structure cannot be directly accessed. We interact with it through its interface, by providing inputs to stimulate it and observing its behaviour through its outputs. Then, to be able to treat the implementation iut, we make the following two assumptions about it:

1. The implementation iut can be modeled as a coalgebra $(S', \mathsf{init}', \alpha')$ over the signature $T(\mathsf{Out}' \times \_)^{\mathsf{In}'}$ with $\mathsf{In} \subseteq \mathsf{In}'$ and $\mathsf{Out} \subseteq \mathsf{Out}'$ (In and Out are the input and output sets of

the specification respectively). This assumption is imposed to allow the specification spec to accept all responses of the implementation.

We also denote by iut the coalgebra modeling the implementation to avoid any excessive denotations;

2. The iut is *input-enabled*, i.e. at any state, it must produce answers for all inputs provided by the environment.

The following definition formalizes these two assumptions:

Definition 1.2 *(*iut ***model****) Let* spec *be a specification over* $T(\mathsf{Out} \times \_)^{\mathsf{In}}$. *A* ***system under test*** *or* ***implementation*** *of* spec *is a component* $(S', \mathsf{init}', \alpha')$ *over the signature* $T(\mathsf{Out}' \times \_)^{\mathsf{In}'}$ *where* α' *is considered as a total function:*

$$\forall (s', i') \in S' \times \mathsf{In}', \exists (o', s'') \in \mathsf{Out}' \times S' \text{ such that } (o', s'') \in \eta'_{\mathsf{Out}' \times S'}(\alpha(s')(i'))$$

and $\mathsf{In} \subseteq \mathsf{In}'$ *and* $\mathsf{Out} \subseteq \mathsf{Out}'$.

1.3 Conformance

1.3.1 An overview

The notion of conformance is usually based on the comparison between the behaviour of a specification and an implementation using a conformance relation. The goal of this relation is to specify what the conformance of an implementation is to its specification. Several kinds of relations have been proposed in the literature. They differ mainly in both the formalism used to model system behaviour and the testing aspects considered. Let us informally review some of them.

The original conformance testing relation proposed for finite state machines (FSM) is defined as the testing equivalence of states whose goal is to determine the equivalence of two machines [111]. Two state machines are said to be *equivalent* if they produce exactly the same sequence of outputs when offered the same sequence of inputs. There is a list of other conformance relations that can be found in the literature. The definitions of these relations depend mainly on the underlying properties of the finite state machines we use. Table VII.1 reviews some of them without going into details. For more detailed explanations, see [112, 113, 114, 115].

Relation	Informal definition	Properties
Equivalence $\cong$	equality of traces set	complete deterministic, or complete nondeterministic
Quasi Equivalence $\sqsupseteq$	for each input sequence of spec, spec and iut produce the same output sequences	deterministic or nondeterministic
Reduction $\leq$	trace inclusion	complete nondeterministic
Quasi reduction $\preceq$	for each input sequence of spec, iut produces only output sequences of spec	nondeterministic

Table VII.1 – Examples of conformance relations

It turns out that the conformance relations to test state equivalence are too strong, in practice, for conformance testing. There is a number of common assumptions (e.g. specification is

strongly connected, minimized or complete) that are usually made in the literature to make test processes at all possible [111, 65, 116, 117]. Test generation algorithms based on them are also expensive in time and memory [65, 111, 118, 119, 120, 121], contrary to test cases generation techniques for inclusion relations (e.g. reduction and quasi reduction relations) [112, 113, 114, 115].

The test relations proposed for labeled transition systems (*LTS*) are usually equivalence and preorder relations relying on the notion of observable behaviours. Many works have been done on establishing the relations between *LTSs*. The relation *trace preorder* $\leq_{tr}$ requires the inclusion of sets of traces. That means an implementation iut may show only behaviour which is specified in the specification. The *testing preorder* $\leq_{te}$ [122, 123] means that if the implementation iut makes a trace which corresponds to a computation of iut after which no more action is possible, then the specification spec has to make the same trace. The *conf* [124] is a variant of $\leq_{te}$ in which all possible observations (i.e. Σ^*) are restricted to only traces contained in the specification. In other words, it requires that the implementation behaves according to a specification, but allows behaviours on which the specification puts no constraint. The *refusal preorder* $\leq_{rf}$ [125] is a variant of $\leq_{tr}$. The main difference between them is that $\leq_{rf}$ is able to detect possible deadlock states i.e. states from which the system cannot go further. This is done by extending the definition of a labeled transition system with refusal transitions.

Testing methods for *LTS* models are based on symmetric communication between the system and its environment. Both environment and system actions are indeed treated in the same way. There is no notion of input or output. However, it turns out such a classification of actions into inputs and outputs leads to a closer link to testing process reality [27, 37, 38, 45, 67]. This is due to the fact that outputs (respectively inputs) have to be considered as actions that are initiated by, and under control of the system (respectively that are initiated by, and under control of the environment). That distinction between input and output actions then has a fundamental role in testing practice in which the tester chooses an input action i, provides it to the implementation under test, and then observes output actions produced by the implementation after i. We refer to the state of arts of the thesis [38, 126, 127] as well as to the articles [40, 128] for further details.

Hence, most of the relations based on *LTS* have been extended to *IOLTS* models. Both the testing preorder $\leq_{te}$ and the refusal preorder $\leq_{rf}$ were redefined to allow to take into account inputs and outputs of systems [46, 129]. The $conf$ relation was also adapted for the *IOLTS* models, and called $ioconf$ [40, 46]. Indeed, the relation $ioconf$ is similar to $conf$, but distinguishes inputs from outputs, and restricts all possible observations to the traces of the specification. It checks only whether a given implementation does what it should do, without regard to unspecified behaviours. The implementation then has the freedom to do more that what is specified. The relation *ioco* [40, 46] is similar to *ioconf*, but it uses suspension traces (i.e. traces generated from suspension models whose quiescence[2] is specified) instead of proper traces (i.e. traces generated from models whose quiescence is not handled) to check the conformance between iut and spec. There are many other types of relations [68, 82, 130].

The relations *conf*, *ioconf* and *ioco* have received much attention by the community of formal testing because they have shown their suitability for conformance testing and automatic test derivation [40]. The reason is that the objective of conformance testing is mainly to check whether the implementation behaves as required by the specification i.e. to check if the implementation does what it should do. Hence, a conformance relation has to allow implementations not only to do what is specified, but also to do more than what is specified (for instance, when

2. The word "quiescence" is used to refer to blocking situations in states.

an annoyed user hits or kicks the coffee machine, or does other strange things that we are not usually considered in the specification). This requirement of testing conformance is well satisfied by *conf*, *ioconf* and *ioco* contrary to other relations [68, 122, 123, 125, 130] that require testing behaviours that are not in the specification i.e. the implementation does not have the freedom to produce outputs for any input not considered in the specification.

Since we are dealing with components with input and output and quiescence is implicitly defined in our component models, we choose *ioco* and extend it to fit our framework taking into account that other relations previously presented could also have been defined in our framework. There are several extensions to *ioco* according to both the transition system type and the aspect considered to be tested. For instance, *sioco* for symbolic transition systems [43], *sioco* for input-output symbolic transition systems [47], *tioco* for timed labeled transition systems [131], *cspio* for *CSP* process algebra [132], *dioco* for distributed systems [133], *uicoco* for hybrid system [134], etc.

1.3.2 Definition

We redefine the *ioco* conformance testing relation that we will call here *cioco*[3] in terms of components as defined in Chapter IV. We make some modifications to the original definition of *ioco* to fit our component definition. That is, after each trace *tr* of a specification spec, instead of considering that the possible outputs of the corresponding implementation iut after executing *tr* on it is a subset of the possible outputs of spec, we consider that the corresponding implementation iut, after executing *tr* on it and then submitting any input *i* of the specification to it, does not produce outputs that are not allowed by spec.

The formal definition of *cioco* uses the following definition:

Definition 1.3 *(Out* after (tr, i)*) Let $\mathcal{C}$ be a component over $T(\mathsf{Out} \times \_)^{\mathsf{In}}$. Let tr be a finite trace of $\mathcal{C}$ and $i \in \mathsf{In}$. The set of the possible outputs for input i after executing tr on $\mathcal{C}$ is:*

$$Out(\mathcal{C} \text{ after } (tr, i)) = \{o \quad | \quad tr.\langle i|o\rangle \in Trace(\mathcal{C})\}$$

Hence, the relation *cioco* is formally redefined in terms of coalgebras as follows:

Definition 1.4 ***(cioco)*** *Let* spec *be the component over the signature $T(\mathsf{Out} \times \_)^{\mathsf{In}}$ and* iut *be the component over $T(\mathsf{Out}' \times \_)^{\mathsf{In}'}$ such that* iut *satisfies the assumptions stated in Definition 1.2.* ***cioco*** *is defined as follows:*

$$\mathsf{iut} \textbf{ cioco } \mathsf{spec} \iff \forall tr \in Trace(\mathsf{spec}), \forall i \in \mathsf{In}, Out(\mathsf{iut} \text{ after } (tr, i)) \subseteq Out(\mathsf{spec} \text{ after } (tr, i))$$

Example 1.1 *Consider the specification* spec *of the coffee machine illustrated in Figure IV.1 and three implementations under test* $\mathsf{iut}_1, \mathsf{iut}_2$ *and* iut_3 *depicted in Figure VII.1a, Figure VII.1b and Figure VII.1c respectively. Then, one has:*

- (iut_1 *cioco* spec) *as after executing any finite trace of* spec *on* iut*, the outputs of* iut_1 *are included into the outputs of* spec*, when any specified input of* spec *is submitted to* iut*. For instance, after the finite trace $tr = \langle \mathsf{coin}|\mathsf{abs}, \mathsf{coffee}|\mathsf{served}, \mathsf{coin}|\mathsf{abs}\rangle$ of* spec*, for the "*coffee*" input, one has:*

$$Out(\mathsf{iut}_1 \text{ after } (tr, \mathsf{coffee})) = \{\mathsf{served}, \mathsf{refund}\}$$

$$\subseteq$$

$$Out(\mathsf{spec} \text{ after } (tr, \mathsf{coffee})) = \{\mathsf{served}, \mathsf{refund}\}$$

3. *c* for component

– $\neg(\mathsf{iut}_2$ *cioco* $\mathsf{spec})$. *This is because the output* "refund" *from the state* FAILED *of* iut_2 *after the finite trace* $\langle \mathsf{coin}|\mathsf{abs}, \mathsf{coffee}|\mathsf{refund}\rangle$ *is not allowed by* spec *when* iut_2 *receives the input* "repair" *(see the red transition* repair|refund*), while the specification only allows the output* abs*:*

$$Out(\mathsf{iut}_2 \text{ after } (\langle \mathsf{coin}|\mathsf{abs}, \mathsf{coffee}|\mathsf{refund}\rangle, \mathsf{repair})) = \{\mathsf{refund}\}$$

$$\notin$$

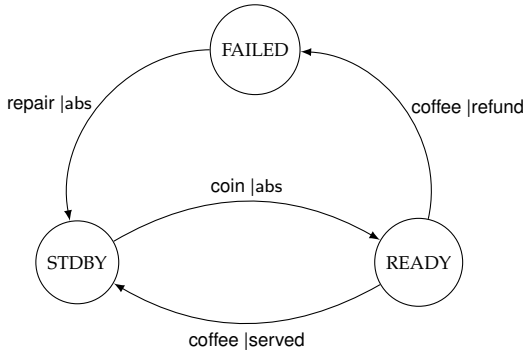
$$Out(\mathsf{spec} \text{ after } (\langle \mathsf{coin}|\mathsf{abs}, \mathsf{coffee}|\mathsf{refund}\rangle, \mathsf{repair})) = \{\mathsf{abs}\}$$

– $\neg(\mathsf{iut}_3$ *cioco* $\mathsf{spec})$. *This is because the output* abs *of* iut_3 *after the trace* $\langle \mathsf{coin}|\mathsf{abs}\rangle$ *is not allowed by the specification* spec *when* iut_3 *receives the input* "coffee" *(see the red transition* coffee|abs*):*

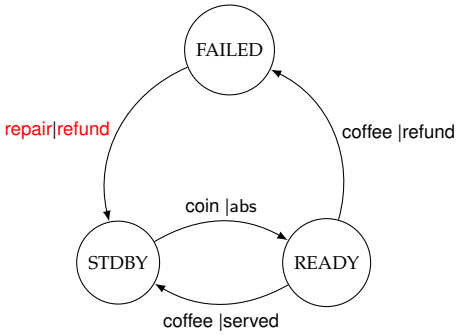
$$Out(\mathsf{iut}_3 \text{ after } (\langle \mathsf{coin}|\mathsf{abs}\rangle, \mathsf{coffee})) = \{\mathsf{served}, \mathsf{refund}, \mathsf{abs}\}$$

$$\notin$$

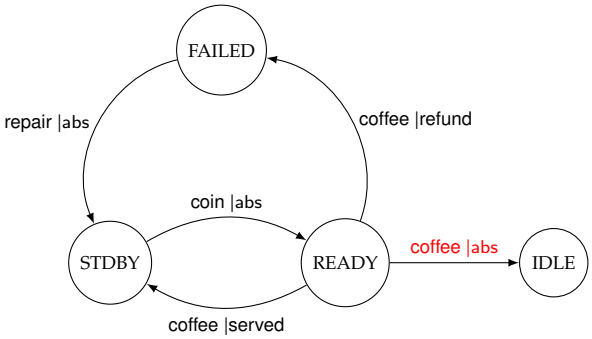
$$Out(\mathsf{spec} \text{ after } (\langle \mathsf{coin}|\mathsf{abs}\rangle, \mathsf{coffee})) = \{\mathsf{served}, \mathsf{refund}\}$$



(a) Implementation iut_1 in conformance to the specification spec



(b) Implementation iut_2 not in conformance to the specification spec



(c) Implementation iut_3 not in conformance to the specification spec

Figure VII.1 – Illustration of *cioco*

Our definition of the relation *cioco* is generic enough to encompass the different *ioco* relations defined in formalisms instances of our framework. Let us show that for *IOLTS* formalism. First of all, let us recall the formal definition of *ioco* in the context of *IOLTS* model:

Definition 1.5 ***(ioco)*** *Let* spec *be an IOLTS and* iut *be an input complete IOLTS, where the alphabets of* iut *and* spec *are compatible, i.e.* $\Sigma^{!}_{\mathsf{spec}} \subseteq \Sigma^{!}_{\mathsf{iut}}$ *and* $\Sigma^{?}_{\mathsf{spec}} \subseteq \Sigma^{?}_{\mathsf{iut}}$ *then:*

$$\mathsf{iut}\ \textit{ioco}\ \mathsf{spec} =_{\text{def}} \forall tr \in \textit{Trace}(\mathsf{spec}),\ \textit{Out}(\mathsf{iut}\ \mathsf{after}\ tr) \subseteq \textit{Out}(\mathsf{spec}\ \mathsf{after}\ tr)$$

where given an IOLTS $\mathcal{M} = (S, init, \Sigma_\tau, \mathsf{Tr})$,

- $\textit{Trace}(\mathcal{M}) = \{(a_0, \ldots, a_n) \mid \exists (s_0, \ldots, s_{n+1}) \in S \text{ with } s_0 = init \text{ and } s_i \xrightarrow{a_i} s_{i+1}, i \leq n\}$
- $\textit{Out}(\mathcal{M}\ \mathsf{after}\ tr)$ *is defined for* $\sigma \in \textit{Trace}(\mathcal{M})$ *by* $\{o \mid \sigma.o \in \textit{Trace}(\mathcal{M}) \text{ and } o \in \Sigma^{!}_{\mathcal{M}}\}$

By using the transformation defined in Definition 1.3 of Chapter IV, we have the following proposition:

Proposition 1.1 *Let* $\mathcal{M}_1$ *and* $\mathcal{M}_2$ *be two suspension IOLTSs. Let* $\phi(\mathcal{M}_1)$ *and* $\phi(\mathcal{M}_2)$ *be the components obtained after transforming* $\mathcal{M}_1$ *and* $\mathcal{M}_2$ *in our framework. Then, we have:*

$$\mathcal{M}_1\ \textit{ioco}\ \mathcal{M}_2 \quad \text{iff} \quad \phi(\mathcal{M}_1)\ \textit{cioco}\ \phi(\mathcal{M}_2)$$

Proof

$\Longrightarrow$

Let $(\mathcal{M}_1\ \textit{ioco}\ \mathcal{M}_2), tr \in \textit{Trace}(\phi(\mathcal{M}_2))$, *i an input of* $\phi(\mathcal{M}_2)$ *and* $o \in \textit{Out}(\phi(\mathcal{M}_1)\ \mathsf{after}\ (tr, i))$ *and let us prove that* $o \in \textit{Out}(\phi(\mathcal{M}_2)\ \mathsf{after}\ (tr, i))$.
$o \in \textit{Out}(\phi(\mathcal{M}_1)\ \mathsf{after}\ (tr, i))$ *implies that* $tr.\langle i|o\rangle \in \textit{Trace}(\phi(\mathcal{M}_1))$. *By Corollary 2.1 (in Chapter IV), we have that* $tr.\langle i|o\rangle \in \phi_t(\textit{Trace}(\mathcal{M}_1))$. *Then,*

$$\exists tr'.a \in \textit{Trace}(\mathcal{M}_1) \mid tr.\langle i|o\rangle = \phi_t(tr').\phi_t(a) \tag{VII.1}$$

Similarly, $tr \in \textit{Trace}(\phi(\mathcal{M}_2))$ *implies that* $tr' \in \textit{Trace}(\mathcal{M}_2)$.
Since $\mathcal{M}_1\ \textit{ioco}\ \mathcal{M}_2$, *then* $a \in \textit{Out}(\mathcal{M}_2\ \mathsf{after}\ tr')$. *That amounts to say* $tr'.a \in \textit{Trace}(\mathcal{M}_2)$. *Thus,* $\phi_t(tr'.a) \in \phi_t(\textit{Trace}(\mathcal{M}_2))$. *Hence, by Corollary 2.1 (in Chapter IV) and Equation VII.1, we can conclude that* $tr.\langle i|o\rangle \in \textit{Trace}(\phi(\mathcal{M}_2))$. *Consequently,* $o \in \textit{Out}(\phi(\mathcal{M}_2)\ \mathsf{after}\ (tr, i))$.

$\Longleftarrow$

Let $(\phi(\mathcal{M}_1)\ \textit{cioco}\ \phi(\mathcal{M}_2)), tr \in \textit{Trace}(\mathcal{M}_2)$ *and* $o \in \textit{Out}(\mathcal{M}_1\ \mathsf{after}\ tr)$ *and let us prove that* $o \in \textit{Out}(\mathcal{M}_2\ \mathsf{after}\ tr)$.
$o \in \textit{Out}(\mathcal{M}_1\ \mathsf{after}\ tr)$ *implies that* $tr.o \in \textit{Trace}(\mathcal{M}_1)$ *and* $tr \in \textit{Trace}(\mathcal{M}_2)$. *Then,* $\phi_t(tr).\phi_t(o) \in \phi_t(\textit{Trace}(\mathcal{M}_1))$ *and* $\phi_t(tr) \in \phi_t(\textit{Trace}(\mathcal{M}_2)$. *By Corollary 2.1 (in Chapter IV), we have then that* $\phi_t(tr).\langle \mathsf{abs}_? | o\rangle \in \phi(\textit{Trace}(\mathcal{M}_1))$ *and* $\phi_t(tr) \in \textit{Trace}(\phi(\mathcal{M}_2))$. *But we know by hypothesis that* $(\phi(\mathcal{M}_1)\ \textit{cioco}\ \phi(\mathcal{M}_2))$, *then* $o \in \textit{Out}(\phi(\mathcal{M}_2)\ \mathsf{after}\ (\phi_t(tr), \mathsf{abs}_?))$. *That implies* $\phi_t(tr).\langle \mathsf{abs}_? | o\rangle \in \textit{Trace}(\phi(\mathcal{M}_2))$. *Then, by Corollary 2.1 (in Chapter IV) we have that* $\phi_t(tr).\langle \mathsf{abs}_? | o\rangle \in \phi_t(\textit{Trace}(\mathcal{M}_2))$. *Hence, by applying* ϕ_t^{-1}, $tr.o\rangle \in (\textit{Trace}(\mathcal{M}_2))$. *Consequently, we have that* $o \in \textit{Out}(\mathcal{M}_2\ \mathsf{after}\ tr)$.
End

Similarly to *ioconf*, verifying that a Mealy machine implementation iut *quasi-reduction*[4] a Mealy machine specification spec is not the same as checking whether *cioco* holds for the components resulting by instancing iut and spec in our framework. Figure VII.2 shows a counterexample: the implementation iut conforms to the specification spec *w.r.t. quasi-reduction*, that is because for all input sequences of spec, iut'outputs are allowed by spec. For instance, after the input

4. See Table VII.1 for *quasi-reduction* definition.

sequence $\langle i_1, i_2, i_3, i_4 \rangle$, the output o_5 of iut is specified. On the other hand, iut does not conforms to spec *w.r.t. cioco*. That is because, after the trace $\langle i_1|o_1, i_2|o_3, i_3|o_4 \rangle$, there exists an input i_4 such that $Out(\text{iut after } (\langle i_1|o_1, i_2|o_3, i_3|o_4 \rangle, i_4) = \{o_6\} \not\subseteq Out(\text{spec after } (\langle i_1|o_1, i_2|o_3, i_3|o_4 \rangle, i_4) = \{o_5\}$.

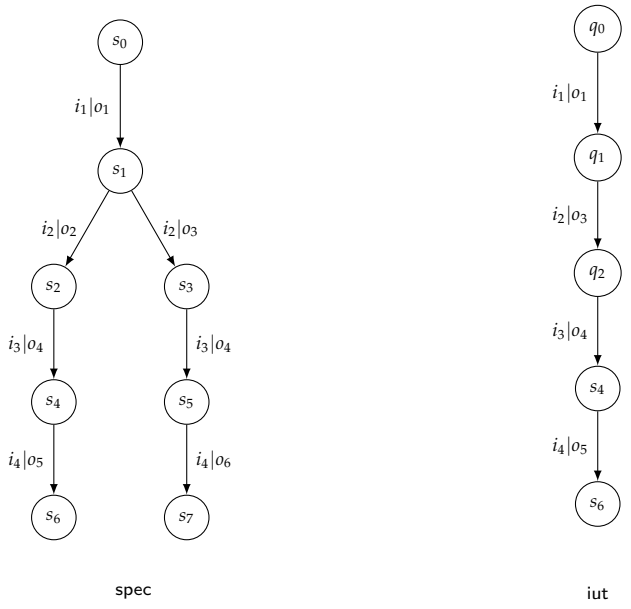


Figure VII.2 – Counter example for *quasi-reduction*

2 Finite computation tree

A component $\mathcal{C} = (S, init, \alpha)$ over the signature $T(\mathsf{Out} \times \_)^{\mathsf{In}}$ can be unfolded into a tree. Intuitively, this tree contains all the information about the possible executions of the component $\mathcal{C}$. It will form the cornerstone of the definition of test purposes in Section 3.

2.1 Formal definition

In this subsection, we define the finite computation tree of a component $\mathcal{C}$ which captures all its finite traces.

Definition 2.1 *($\mathcal{C}$-**paths**) Let $\mathcal{C} = (S, s_0, \alpha)$ be a component over $T(\mathsf{Out} \times \_)^{\mathsf{In}}$. A $\mathcal{C}$-**path** is defined by two finite sequences of states and inputs $(s_0, \ldots, s_n)$ and $(i_0, \ldots, i_{n-1})$ such that:*

$$\forall j, 1 \leq j \leq n, s_j \in \eta'_{\mathsf{Out} \times S}\big(\alpha(s_{j-1})(i_{j-1})\big)_{|_2}$$

Definition 2.2 *(**Finite computation tree of component**) Let (S, s_0, α) be a component over the signature $T(\mathsf{Out} \times \_)^{\mathsf{In}}$. The **finite computation tree** of depth n of $\mathcal{C}$, noted $FCT(\mathcal{C}, n)$, is the coalgebra $(S_{FCT}, s^0_{FCT}, \alpha_{FCT})$ defined by:*

- *S_{FCT} is the whole set of $\mathcal{C}-$*paths
- *s^0_{FCT} is the initial $\mathcal{C}-$*path $\langle s_0, () \rangle$
- *α_{FCT} is the mapping which for every $\mathcal{C}-$*path $\langle (s_0, \ldots, s_n), (i_0, \ldots, i_{n-1}) \rangle$ *and every input $i \in \mathsf{In}$ associates $\eta'^{-1}_{\mathsf{Out} \times S_{FCT}}(\Gamma)$ where Γ is the set:*

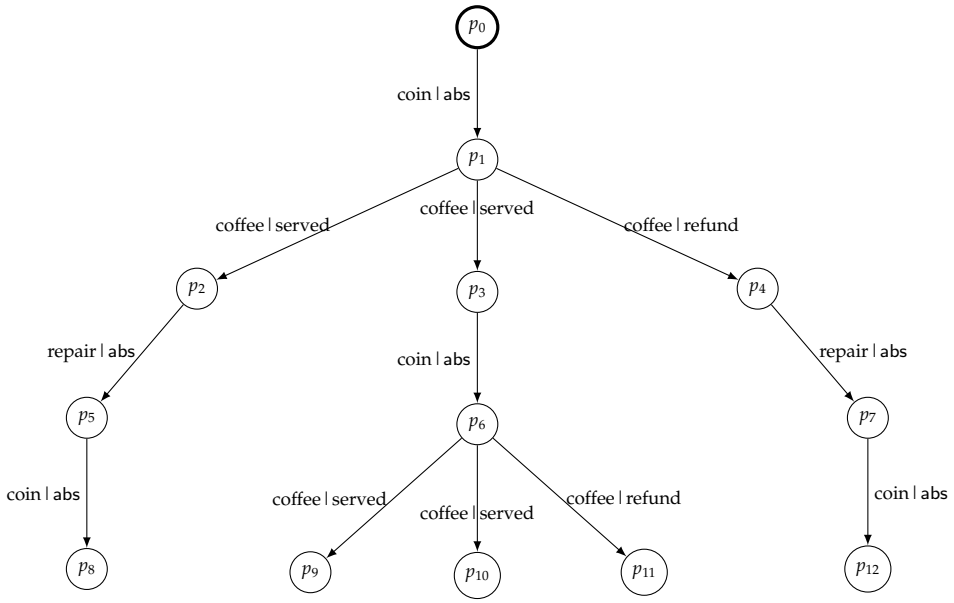
$$\Gamma = \Big\{ \big(o, \langle (s_0, \ldots, s_n, s'), (i_0, \ldots, i_{n-1}, i) \rangle\big) \mid (o, s') \in \eta'_{\mathsf{Out} \times S}\big(\alpha(s_n)(i)\big) \Big\}$$

In this definition, S_{FCT} is the set of the nodes of the tree. s^0_{FCT} is the root of the tree. Each node is represented by the unique $\mathcal{C}$-path $\langle (s_0, \ldots, s_n), (i_0, \ldots, i_{n-1}) \rangle$ which leads to it from the root:

$$s_0 \xrightarrow{i_0} s_1 \xrightarrow{i_1} \ldots \xrightarrow{i_{n-2}} s_{n-1} \xrightarrow{i_{n-1}} s_n$$

α_{FCT} gives, for each node p and for each input i, the set of nodes Γ that can be reached from p when the input i is submitted to the component.

Example 2.1 *Figure VII.3 gives the finite computation tree of depth* 4 *of the coffee machine* $\mathcal{M}$ *whose specification is shown in Figure IV.1.*



$p_0 = \langle \text{STDBY}, () \rangle$
$p_1 = \langle (\text{STDBY}, \text{READY}), \text{coin} \rangle$
$p_2 = \langle (\text{STDBY}, \text{READY}, \text{FAILED}), (\text{coin}, \text{coffee}) \rangle$
$p_3 = \langle (\text{STDBY}, \text{READY}, \text{STDBY}), (\text{coin}, \text{coffee}) \rangle$
$p_4 = \langle (\text{STDBY}, \text{READY}, \text{FAILED}), (\text{coin}, \text{coffee}) \rangle$
$p_5 = \langle (\text{STDBY}, \text{READY}, \text{FAILED}, \text{STDBY}), (\text{coin}, \text{coffee}, \text{repair}) \rangle$
$p_6 = \langle (\text{STDBY}, \text{READY}, \text{STDBY}, \text{READY}), (\text{coin}, \text{coffee}, \text{coin}) \rangle$
$p_7 = \langle (\text{STDBY}, \text{READY}, \text{FAILED}, \text{STDBY}), (\text{coin}, \text{coffee}, \text{repair}) \rangle$
$p_8 = \langle (\text{STDBY}, \text{READY}, \text{FAILED}, \text{STDBY}, \text{READY}), (\text{coin}, \text{coffee}, \text{repair}, \text{coin}) \rangle$
$p_9 = \langle (\text{STDBY}, \text{READY}, \text{STDBY}, \text{READY}, \text{STDBY}), (\text{coin}, \text{coffee}, \text{coin}, \text{coffee}) \rangle$
$p_{10} = \langle (\text{STDBY}, \text{READY}, \text{STDBY}, \text{READY}, \text{FAILED}), (\text{coin}, \text{coffee}, \text{coin}, \text{coffee}) \rangle$
$p_{11} = \langle (\text{STDBY}, \text{READY}, \text{STDBY}, \text{READY}, \text{FAILED}), (\text{coin}, \text{coffee}, \text{coin}, \text{coffee}) \rangle$
$p_{12} = \langle (\text{STDBY}, \text{READY}, \text{FAILED}, \text{STDBY}, \text{READY}), (\text{coin}, \text{coffee}, \text{repair}, \text{coin}) \rangle$

Figure VII.3 – Finite computation tree for the coffee machine

It is easy to notice that a component $\mathcal{C}$ and its finite computation tree $FCT(\mathcal{C})$ share the same trace semantics i.e. $Trace(\mathcal{C}) = Trace(FCT(\mathcal{C}))$. It is therefore equivalent to study a component or its finite computation tree in the context of our work.

2.2 Unfolding algorithm

In this subsection, we show that Definition 2.2 is computable by showing how to build a tree $FCT(\mathcal{C})$ algorithmically that captures all the possible finite computation paths of the component $\mathcal{C} = (S, s_0, \alpha)$ over the signature $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$. This tree can be thought of as a data structure representing the component computations obtained after a finite unfolding of $\mathcal{C}$. Starting from the initial state s_0 of $\mathcal{C}$, a tree t containing all the elementary paths is built by running a depth-first search (DFS) as well as a set $\mathbb{H}(\mathcal{C})$ containing the heads (i.e. the first explored state) of all elementary circuits which exist in $\mathcal{C}$. To be more accurate, we assign a unique name to each explored state and we also maintain the original state in $\mathcal{C}$ for every explored state in $\mathbb{H}(\mathcal{C})$. The $FCT(\mathcal{C})$ is therefore built recursively by binding to t in every state in $\mathbb{H}(\mathcal{C})$, the tree obtained by applying the depth-first search to its original state in $\mathcal{C}$. This means that if the component has a circuit, then its unfolding is an infinite tree. To prevent this algorithm from running indefinitely (the unfolding contains infinitely many visits to each state), we can enrich it with the option of specifying the maximum time to go through the circuit. Briefly, unfolding the component using this algorithm gives rise to a tree that contains isomorphic subtrees, describing redundant computations, and each state can be reached via a unique execution from the initial state. It allows us to capture all possible behaviours of the component at once. If we need to examine a subset of the behavior of the system, then we use a stopping (or coverage) criteria which determines the maximum number of passes through each circuit of $\mathcal{C}$.

Hence, our mechanism of unfolding can be described by two algorithms. The first one, called Enumerate, is a simple depth-first search that gives rise, as results, to the tree containing all the elementary paths of $\mathcal{C}$, and the first explored state of every circuit existing in $\mathcal{C}$. The second algorithm, called Unfolding, allows one to construct recursively the entire computation tree taking into account how many times we go through every circuit of $\mathcal{C}$.

Algorithm 1: Enumerate$(\mathcal{C}, (S', \mathsf{init}', \alpha'), s, LS, HC, P)$

while $\exists i \in \mathsf{In}, (o, s') \in \mathsf{Out} \times S \mid (o, s') \in \eta'(\alpha(s)(i))$ **do**
  $nb' \leftarrow$ `GiveNumber`(s');
  `Add`(S', nb');
  $\alpha'(\mathsf{init}')(i) \leftarrow \alpha'(\mathsf{init}')(i) \cup \eta'^{-1}(\{(o, nb')\})$;
  if `Color`$(s') = W$ *or* R **then**
    `Push`(P, s');
    `ModifyColor`(LS, s', G);
    `Enumerate`$(\mathcal{C}, (S', nb', \alpha'), s', LS, HC, P)$;
  else `Add`$(HC, (s', nb'))$;
`Pop` (P);
`ModifyColor`(LS, s, R);
return $((S', \mathsf{init}', \alpha'), HC)$

Proposition 2.1 Enumerate *runs in* $\Theta(mN)$ *while* Unfolding *in* $\Theta(mNnb^n)$ *with:*

- *m is the number of states of the component to be unfolded;*
- *N is the cardinality of the input set* In*;*
- *nb is the number of circuits detected in $\mathcal{C}$;*
- *n is the number of passes of the algorithm through each circuit.*

Proof

Algorithm 2: Unfolding$(\mathcal{C}, s, n)$

input : a component $\mathcal{C} = (S, \mathsf{init}, \alpha)$ over $T(\mathsf{Out} \times \_)^{\mathsf{In}}$, a state $s \in S$ and positive number n

output: a finite tree $FCT(\mathcal{C}, n)$ containing all paths of $\mathcal{C}$ outgoing from s. Each one of these paths does not go through more n circuit

initialization ;

$(S', \mathsf{init}', \alpha') \leftarrow$ `CreateEmptyComponent();`

$LS \leftarrow$ `CreateEmptyListOfColoredStates();`

$P \leftarrow$ `CreateEmptyStack();`

$HC \leftarrow$ `CreateEmptyListOfStateState();`

`Push`(P, s);

for $state \in S$ **do**

  `Add`$(LS, (state, W))$

$nb \leftarrow$ `GiveNumber`(s);

$\mathsf{init}' \leftarrow nb$;

`Add`(S', nb);

`ModifyColor`(LS, s, G);

$(S', \alpha'), HC \leftarrow$ `Enumerate`$(\mathcal{C}, (S', \alpha'), s, LS, HC, P)$;

if $n == 1$ **then**

  return $(S', \mathsf{init}', \alpha'))$

else

  for $(x, nb) \in HC$ **do**

    $(S'', \mathsf{init}'', \alpha'') \leftarrow$ `Unfolding`$(\mathcal{C}, x, n-1)$;

    $nb \leftarrow \mathsf{init}''$;

    $S' \leftarrow S' \cup S''$;

    $$\alpha'(y)(i) \mapsto \begin{cases} \alpha'(y)(i) & if \quad y \in S' \\ \alpha''(y)(i) & otherwise \end{cases}$$

Enumerate*: Let $T(m)$ be the number of elementary operations*[5] *necessary to enumerate all elementary paths of a component $\mathcal{C}$ over $T(\mathsf{Out} \times \_)^{\mathsf{In}}$ whose state number is m. We can easily see that* Enumerate *satisfies the following recurrence:*

$$T(m) = \begin{cases} 3 + 8 \times N & if \qquad m = 1 \\ 3 + 8 \times N + T(m-1) & otherwise \end{cases}$$

To guess at a solution, let's try unrolling the recurrence, by substituting it into itself as follows:

$$\begin{aligned} T(m) &= T(m-1) + 3 + 8 \times N \\ T(m-1) &= T(m-2) + 3 + 8 \times N \\ &\vdots \\ T(m-(m-2)) &= T(1) + 3 + 8 \times N \end{aligned}$$

Then, $T(m)$ satisfies the recurrence:

$$T(m) = m(3 + 8 \times N)$$

Consequently, Enumerate *runs in $\Theta(mN)$.*

Unfolding*: Let $T(n)$ be the number of elementary operations necessary to unfolding a component $\mathcal{C}$ over $T(\mathsf{Out} \times \_)^{\mathsf{In}}$ with n as the number of times that the algorithm goes through each circuit in $\mathcal{C}$. We can easily see that* Unfolding *satisfies the following recurrence:*

$$T(n) = \begin{cases} 9 + m + m \times N & if \qquad n = 1 \\ (9 + m + m \times N) + 6 \times nb \times T(n-1) & otherwise \end{cases}$$

To guess at a solution, let's try unrolling the recurrence, by substituting it into itself as follows:

$$\begin{aligned} T(n) &= 6 \times nb \times T(n-1) + 9 + m + m \times N \\ T(n-1) &= 6 \times nb \times T(n-2) + 9 + m + m \times N \\ &\vdots \\ T(n-(n-2)) &= 6 \times nb \times T(1) + 9 + m + m \times N \end{aligned}$$

Then, $T(n)$ satisfies the recurrence:

$$T(n) = (9 + m + m \times N) \sum_{i=0}^{n} (6 \times nb)^i$$

Consequently, Enumerate *runs in $\Theta(mNnb^n)$.*
End

Example 2.2 *Figure VII.4 shows a graphical representation of a component $\mathcal{C}$ (on the left side) and its unfolding $FCT(\mathcal{C}, 2)$ as a finite computation tree (on the right side).* Unfolding$(\mathcal{C}, \mathsf{init}, 2)$ *calls* Enumerate *which in turn returns a tree t containing all the elementary paths of $\mathcal{C}$ (t is colored red) as well as the set containing the couple $(s_1, 5)$. s_1 is the first state involved in the circuit $(s_1 s_2 s_3 s_1)$ and 5 is the node associated to s_1 in $FCT(\mathcal{C}, 2)$. Since we intend to build a tree of depth 2, then* Unfolding *glues in 5 the tree t' (colored blue) built by unfolding $\mathcal{C}$ in s_1.*

5. Operations such that Add, $\cup$, GiveNumber, etc. are considered as elementary operations i.e. they run in $\Theta(1)$.

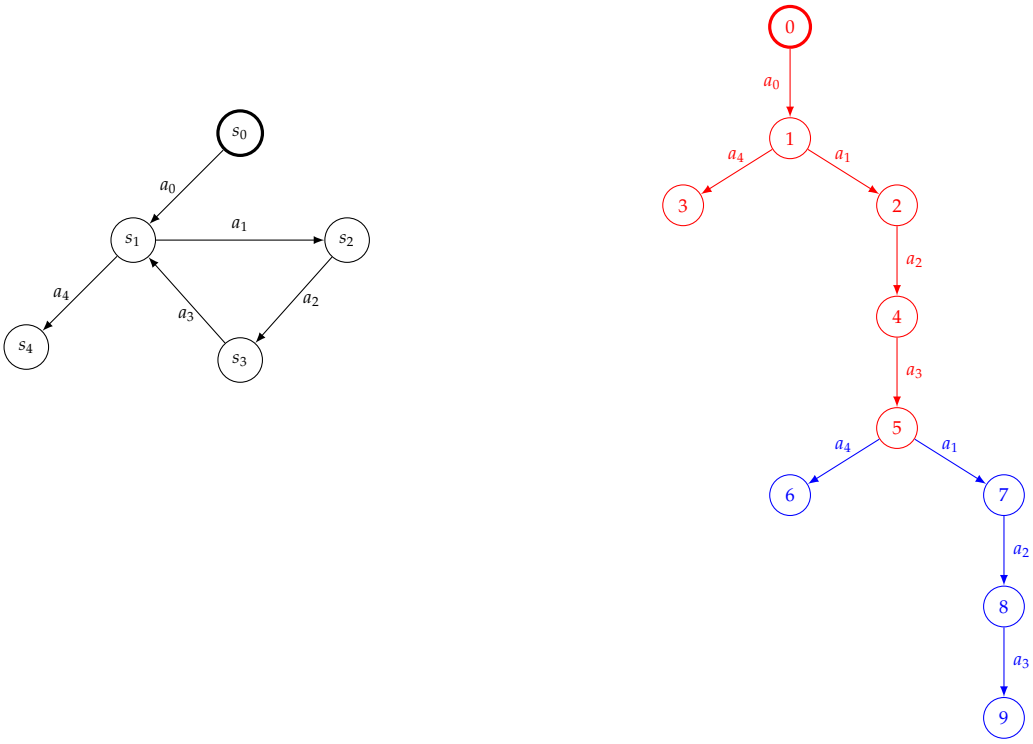


Figure VII.4 – Example of finite computation tree

3 Test Purpose

As previously mentioned, our model of a specification is described as a coalgebra over $T(\mathsf{Out} \times \_)^{\mathsf{In}}$. Such a model usually contains a growth of exponential states which makes the testing process difficult even impossible to be implemented. To cope with this problem, test purposes can be used. A test purpose is a description of the part of the specification that we want to test and for which test cases are to be generated. In other words, it narrows down the model of the specification into smaller ones from which test cases are later generated. In [39, 135], test purposes are described independently of the model of the specification. Then, a synchronous product is done between the test purpose and the specification in order to keep only the paths accepted by the specification. In [44, 53], test purposes are deduced from the specification by construction. More precisely, a test purpose is considered as a finite symbolic subtree of the symbolic execution tree generated from a specification *IOSTS* using a symbolic execution technique. Then, leaves of this tree are labeled by accept and intermediate nodes are labeled by skip (states leading to states from which it is possible to go to accepting states). All other nodes of the tree not belonging to the test are then labeled by a special label $\odot$.

In order to guide the test derivation process in our approach, we prefer, as in [44, 53], to describe test purposes by selecting the part of the specification that we want to explore. We therefore build the finite computation tree $FCT(\mathcal{C})$ of the component $\mathcal{C}$, and consider a test purpose as a tagged finite computation tree of the specification. The leaves of the *FCT* which correspond to paths that we want to test are tagged accept. All internal nodes on such paths are tagged skip, and all other nodes are tagged $\odot$.

In summary, these labels in trees serve either to accept, or to reject the behaviours of specifications as follows: a path that is a part of the test purpose is called *path purpose* and noted $\mathcal{X}$. The last node of the path in question is tagged accept. All other nodes of this path are tagged

skip. They correspond to the fact that it is still possible to transmit additional input i to the implementation and receive its output o to reach an accepted node. Finally, all other nodes that are not part of $\mathcal{X}$ are tagged $\odot$.

Definition 3.1 ***(Test Purpose)*** *Let $FCT(\mathcal{C}, n)$ be the finite computation tree of depth n associated to a component $\mathcal{C}$. A* ***test purpose*** TP *for $\mathcal{C}$ is a mapping $\mathsf{TP} : S_{FCT} \longrightarrow \{\mathsf{accept}, \mathsf{skip}, \odot\}$ such that:*

- *there exists a $\mathcal{C}-$*path *$p \in S_{FCT}$ such that $\mathsf{TP}(p) = \mathsf{accept}$*
- *if $\mathsf{TP}(\langle(s_0, \ldots, s_n), (i_0, \ldots, i_{n-1})\rangle) = \mathsf{accept}$, then:*

  $\forall j, 1 \leq j \leq n-1, \mathsf{TP}(\langle(s_0, \ldots, s_j), (i_0, \ldots, i_{j-1})\rangle) = \mathsf{skip}$
- $\mathsf{TP}(\langle s_0, ()\rangle) = \mathsf{skip}$
- *if $\mathsf{TP}(\langle(s_0, \ldots, s_n), (i_0, \ldots, i_{n-1})\rangle) = \odot$, then:*

  $\mathsf{TP}(\langle(s_0, \ldots, s_n, s'_{n+1}, \ldots, s'_m), (i_0, \ldots, i_{n-1}, i'_n, \ldots, i'_{m-1})\rangle) = \odot$
  for all $m > n$ and for all $(s'_j)_{n<j\leq m}$ and $(i'_k)_{n\leq k<m}$

In order to build a test purpose on a finite computation tree, we choose the leaves of the tree that we accept as correct finite behaviours and tag them with accept. We then tag every node which represents a prefix of an accepted behaviour with skip. The other nodes, which lead to behaviours that we do not want to test, are tagged with $\odot$.

Note a test purpose can be characterized by one path purpose $\mathcal{X}$, two path purposes $\mathcal{X}$, even all the finite computation tree.

In the following, we use the notation TP to refer to an arbitrary test purpose.

Example 3.1 *From the finite computation tree $FCT(\mathcal{M}, 4)$ shown in Figure VII.3, one defines three test purposes:*

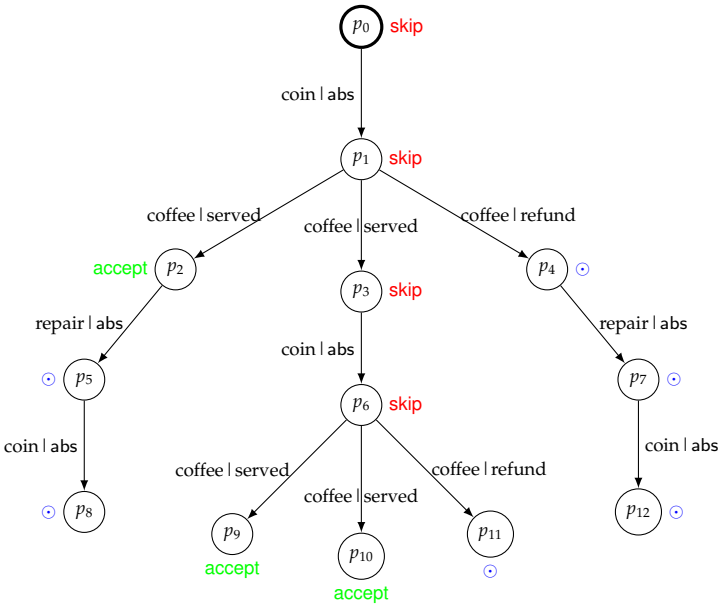
TP_1*: this test purpose allows us to ignore the behaviours of $\mathcal{M}$ related to failure and repair and to concentrate on its interaction with a user. When the machine fails, we reach node p_4, p_5 or p_{11} which are tagged with $\odot$. This indicates that we are not interested in further behaviour from these nodes. p_2, p_9 and p_{10} are tagged with* accept *because they are nodes corresponding to the expected behaviour. All nodes leading from the root p_0 to these nodes are tagged with* skip*.*

TP_2*: this test purpose describes the behaviours of $\mathcal{M}$ where it delivers coffee and goes into a blocked state. When the machine fails after delivering coffee, we only reach node p_5 which we tag with* accept*. Then, p_0, p_1 and p_2 are tagged* skip *and all other nodes are tagged with $\odot$.*

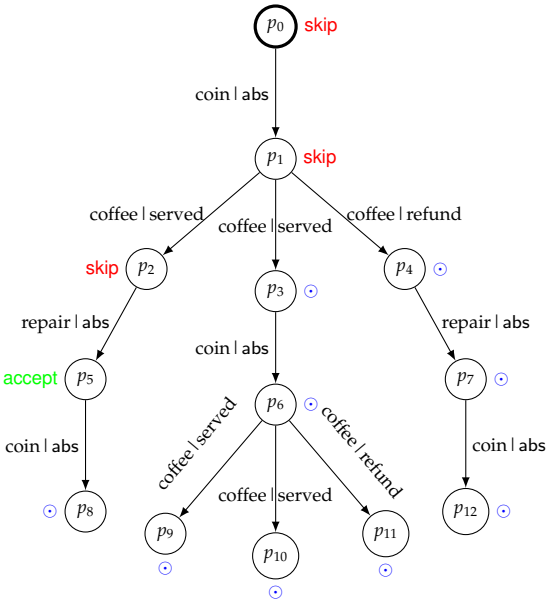
TP_3*: this test purpose describes the behaviours of $\mathcal{M}$ where it fails to deliver coffee to the user but refunds him. When the machine refunds the user and goes into a blocked state, we reach node p_4 or p_{11} which we tag with* accept*. Then, p_0, p_1, p_3 and p_6 are tagged* skip *and all other nodes are tagged with $\odot$.*

4 Test generation guided by test purposes

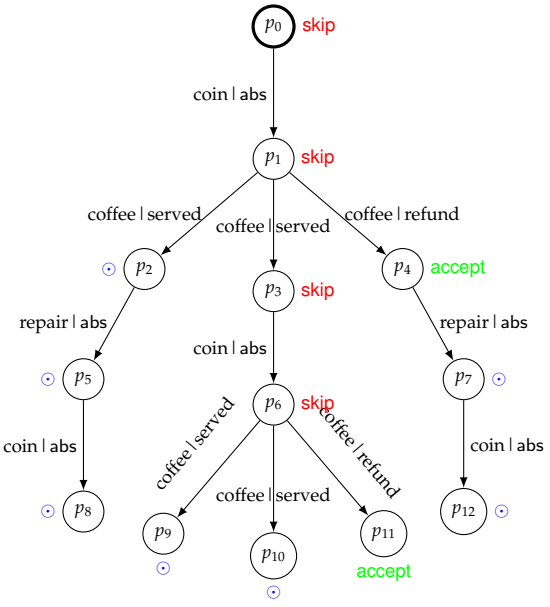
Conformance testing is based on a conformance relation between the component modeling the implementation iut and the component denoting the specification spec. The conformance relation we use is *cioco*, which requires that the iut's outputs are among the outputs allowed by spec for every specified input sequence. A test execution is then the process of feeding the iut with a test case, observing its response and giving some test verdicts.



(a) Test purpose TP_1 with three path purposes $\mathcal{X}$



(b) Test purpose TP_2 with one path purpose $\mathcal{X}$



(c) Test purpose TP_3 with two path purposes $\mathcal{X}$

Figure VII.5 – Test purposes of the coffee machine

There exist several different techniques to generate test cases systematically from a specification in such a way these test cases can distinguish correct implementations, and incorrect ones. These techniques differ by both specification model and conformance relation used for testing. They can be classified in two main categories: *random* techniques and techniques *guided by test purposes*. Random techniques, such as in [46, 27], are used to generate test cases based on non-deterministic choices. This is done by exploring the specification randomly and selecting new actions until a certain user-defined bound on steps is reached, or no match is found between the outputs of the implementation and of the specification. Techniques guided by test purposes, such as in [39, 44], are based on the definition of a test purpose which is used to allow the tester to select a property to be tested. In [39], a test case can be generated by computing the synchronized product of the specification and the test [6] purpose. In [44], test cases are generated from a finite symbolic tree obtained according to a test purpose which can be either chosen manually by the user or computed automatically by means of inclusion [7] criteria.

Similarly to [44], we propose an approach for test case selection according to a test purpose. The advantage of the testing theory proposed in [44] is that it is based on the conformance relation *ioco* that we adopt in our framework and as previously stated, has received much attention by the community of formal testing thanks to its suitability for both conformance testing and automatic test derivation. Furthermore, test generation algorithms proposed in [44] are simple in their implementation and efficient in their execution.

Figure VII.6 illustrates graphically the elements of the approach. A test case is generated from the finite computation tree $FCT(\mathsf{spec})$ of a specification spec enriched with a test purpose TP. It is considered as a sequence of input-output actions built progressively by interacting with the implementation iut, and which examines one of the behaviours of the specification spec selected by the test purpose TP (i.e. the last node of the path is tagged accept). The underlying idea to building such a sequence is the following: we choose an input action i according to the interactions with the iut previously computed and the set of reachable states that can lead to accepting states of TP. Then, the reaction (output) o received from the implementation is compared to the specified ones, and depending on the result of this comparison, our algorithm continues its computation, or stops by generating a verdict. We therefore distinguish four verdicts:

- **Fail**: means that the output o does not match the specification, and then the interaction sequence does not form the trace of any path purpose. Hence, the goal of the test case is not reached;

- **Pass**: means that no observable difference between the specification and the implementation is detected;

- **Inconc**: means that no error is detected but, the test purpose is not achieved;

- **WeakPass**: means that the implementation behaves correctly but, due to the fact of non-deterministic specification, we are not sure whether the test purpose has been achieved.

6. Test purposes are usually supposed to be given by an expert.
7. It is stated in [44], that inclusion criterion can be used to answer industrial needs where engineers are not always able to define which behaviour they want to test.

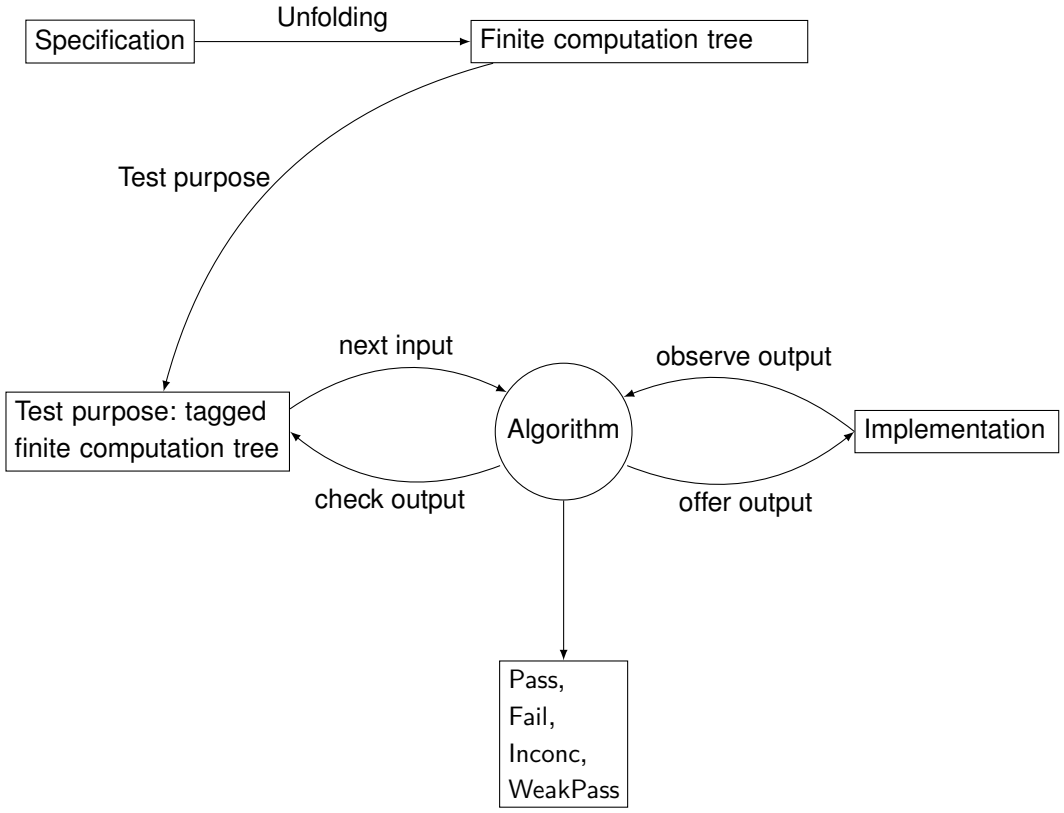


Figure VII.6 – General view of the algorithm

4.1 Preliminaries

In this section, we introduce some notations and definitions that will be used in describing our algorithm for generating conformance tests for components.

As mentioned above, a test case is a sequence generated by a test purpose TP interacting with iut. This is denoted by:

$$[ev_0, ev_1, \ldots, ev_n | V]$$

where for all $j \in [0, \ldots, n]$, $ev_j = i|o$ is an elementary input-output with $i \in \mathsf{In}, o \in \mathsf{Out}$ and $V \in \{\mathsf{Fail}, \mathsf{Pass}, \mathsf{Inconc}, \mathsf{WeakPass}\}$.

We note $stimobs(i|o)$ the output o from iut when stimulating it with input i.

In order to compute the set of reachable states that lead to accept states after a given input-output sequence, we define a current set of states denoted by CS that contains a subset of the states of the test purpose. It is initialized to the initial state of TP. We also introduce three functions to help explore TP by selecting paths that lead to accept states:

- $Next(CS, ev)$ is the set of directly reachable states from the current set of states CS after executing ev;
- $NextSkip(CS, ev)$ is the set of states in $Next(CS, ev)$ which are labeled by skip;
- $NextPass(CS, ev)$ is the set of states in $Next(CS, ev)$ which are labeled by accept.

More formally, these three sets are defined as follows:

Definition 4.1 *Let* $\mathsf{TP} : S_{FCT} \longrightarrow \{\mathsf{accept}, \mathsf{skip}, \odot\}$ *be a test purpose for a component* $\mathcal{C}$, $ev = \langle i|o \rangle$ *an event, and* S' *a subset of* S_{FCT}*:*

- $Next(S', ev) = \bigcup_{s' \in S'} (\{s \mid (o, s) \in \eta'_{Out \times S_{FCT}}(\alpha_{FCT}(s')(i))\})$;
- $NextSkip(S', ev) = Next(S', ev) \bigcap \mathsf{TP}(S')_{|skip}$;
- $NextPass(S', ev) = Next(S', ev) \bigcap \mathsf{TP}(S')_{|accept}$.

with $\mathsf{TP}(S')_{|\mathsf{tag}} = \{s' \in S' \mid \mathsf{TP}(s') = \mathsf{tag}\}$

4.2 Inferences rules

We define our test case generation algorithm as a set of inferences rules. Each rule states that under certain conditions on the next observation of output action from iut or the next stimulation of iut by an input action, the algorithm either performs an exploration of the other states of TP, or stops by generating a verdict.

We structure these rules as

$$\frac{CS}{Results}\ cond(ev)$$

where

- CS is a set of current states;
- $Results$ is either a set of current states or a verdict;
- $cond(ev)$ is a set of conditions including $stimobs(ev)$.

Each rule must be read as follows:

Given the current set of states CS, if $cond(ev)$ *is satisfied, then the algorithm may achieve a step of execution, with ev as input-output elementary sequence.*

This algorithm can be seen as an exploration of the finite computation tree starting from the initial state. It switches between sending stimuli to the implementation and waiting for output of the implementation according to the inference rules as long as a verdict is not reached. We distinguish two kinds of inference rules : *exploring* rules and *diagnosis* rules. The first kind is applied to pursue the computation of the sequence as long as *Result* is a set of states. The second kind leads to a verdict and stops the algorithm.

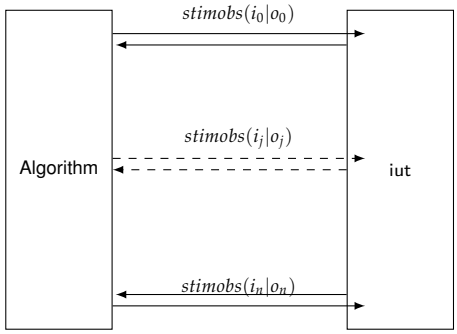


Figure VII.7 – Communication between the iut and the algorithm

Rule 0 Initialization rule [8]: $\frac{}{\{s^0_{FCT}\}}$

Rule 1 Exploration of other states: the emission o after a stimulation by i on the iut is compatible with the test purpose but no accept is reached.

$$\frac{CS}{Next(CS,ev)}\ stimobs(ev),\ NextSkip(CS,ev) \neq \emptyset, NextPass(CS,ev) = \emptyset$$

Rule 2 Generation of the verdict Fail: the emission from the iut is not expected with regards to the specification.

$$\frac{CS}{\mathsf{Fail}}\ stimobs(ev),\ Next(CS,ev) = \emptyset$$

Rule 3 Generation of the verdict Inconc: the emission from the iut is specified but not compatible with the test purpose.

$$\frac{CS}{\mathsf{Inconc}} stimobs(ev), \begin{cases} Next(CS,ev) \neq \emptyset, \\ \\ NextSkip(CS,ev) = NextPass(CS,ev) = \emptyset \end{cases}$$

Rule 4 Generation of the verdict Pass: all next states directly reachable from the set of current set are *accept* ones.

$$\frac{CS}{\mathsf{Pass}}\ stimobs(ev),\ NextPass(CS,ev) = Next(CS,ev),\ Next(CS,ev) \neq \emptyset$$

Rule 5 Generation of the verdict WeakPass: some of the next states are labelled by *accept*, but not all of them.

$$\frac{CS}{\mathsf{WeakPass}} stimobs(ev), \begin{cases} NextPass(CS,ev) \subset Next(CS,ev), \\ \\ NextPass(CS,ev) \neq \emptyset \end{cases}$$

8. This rule is involved only once when starting the algorithm.

Let us note that three forms of ev are distinguished: $i|o$, $i|\epsilon_o$ and $\epsilon_i|o$. Hence, each of these rules except rule 0 can be used in several ways according to the form of ev:

- $stimobs(ev = i|o)$ means o is produced by iut when it is stimulated with i;
- $stimobs(ev = i|\epsilon_o)$ means the stimulation of iut with i does not produce any output;
- $stimobs(\epsilon_i|o)$ means o is produced spontaneously by iut.

These possibilities for ev therefore give rise to a generic algorithm that can be applied to a wide variety of state-based systems ([46, 39, 65]) by choosing the appropriate monad T and input and output sets.

4.3 Example

We illustrate the algorithm previously proposed with concrete examples. We consider the test purposes TP_1 and TP_2 defined in Figure VII.5a and Figure VII.5b respectively, and show how test cases can be obtained by applying the rules presented in Section 4.2. Let us first recall that the algorithm uses the following notation:

$$CS \xrightarrow[\textbf{rule}]{event} CS'$$

where:

- $event$ denotes the current element of the considered trace, and is of the form $input|output$;
- **rule** stands for the rule applied to get the next set of states CS'.

Test cases for TP_1

Fail To get the verdict Fail, we consider the following trace:

$$[\mathsf{coin}|\mathsf{abs}, \mathsf{coffee}|\mathsf{served}, \mathsf{coin}|\mathsf{refund} \mid \mathsf{Fail}]$$

The algorithm is applied as follows:

$$\xrightarrow[\textbf{rule 0}]{} CS_0 = \{p_0\} \xrightarrow[\textbf{rule 1}]{\mathsf{coin}|\mathsf{abs}} CS_1 = \{p_1\} \xrightarrow[\textbf{rule 1}]{\mathsf{coffee}|\mathsf{served}} CS_2 = \{p_2, p_3\}$$

$$CS_2 \xrightarrow[\textbf{rule 2}]{\mathsf{coin}|\mathsf{refund}} \mathsf{Fail}$$

The verdict Fail is due to the following equality:

$$Next(CS_2, \mathsf{coin}|\mathsf{refund}) = \emptyset$$

Inconc To get the verdict Inconc, we consider the following trace:

$$[\mathsf{coin}|\mathsf{abs}, \mathsf{coffee}|\mathsf{served}, \mathsf{coin}|\mathsf{abs}, \mathsf{coffee}|\mathsf{refund} \mid \mathsf{Inconc}]$$

The algorithm is applied as follows:

$$\xrightarrow[\textbf{rule 0}]{} CS_0 = \{p_0\} \xrightarrow[\textbf{rule 1}]{\mathsf{coin}|\mathsf{abs}} CS_1 = \{p_1\} \xrightarrow[\textbf{rule 1}]{\mathsf{coffee}|\mathsf{served}} CS_2 = \{p_2, p_3\}$$

$$CS_2 \xrightarrow[\textbf{rule 1}]{\text{coin}|\text{abs}} CS_3 = \{p_6\} \xrightarrow[\textbf{rule 3}]{\text{coffee}|\text{refund}} \mathsf{Inconc}$$

The verdict Inconc is due to the following two equalities:

- $Next(CS_3, \text{coffee}|\text{refund}) = \{p_{11}\} \neq \emptyset$
- $NextPass(CS_3, \text{coffee}|\text{refund}) = NextSkip(CS_3, \text{coffee}|\text{refund}) = \emptyset$

Pass To get the verdict Pass, we consider the following trace:

$$[\text{coin}|\text{abs}, \text{coffee}|\text{served}, \text{coin}|\text{abs}, \text{coffee}|\text{served} \mid \mathsf{Pass}]$$

The algorithm is applied as follows:

$$\xrightarrow[\textbf{rule 0}]{} CS_0 = \{p_0\} \xrightarrow[\textbf{rule 1}]{\text{coin}|\text{abs}} CS_1 = \{p_1\} \xrightarrow[\textbf{rule 1}]{\text{coffee}|\text{served}} CS_2 = \{p_2, p_3\}$$

$$CS_2 \xrightarrow[\textbf{rule 1}]{\text{coin}|\text{abs}} CS_3 = \{p_6\} \xrightarrow[\textbf{rule 4}]{\text{coffee}|\text{served}} \mathsf{Pass}$$

The verdict Pass is due to the following equality:

$$NextPass(CS_3, \text{coffee}|\text{served}) = Next(CS_3, \text{coffee}|\text{served}),\ Next(CS_3, \text{coffee}|\text{served}) \neq \emptyset$$

WeakPass To get the verdict WeakPass, we consider the following trace:

$$[\text{coin}|\text{abs}, \text{coffee}|\text{served} \mid \mathsf{WeakPass}]$$

The algorithm is applied as follows:

$$\xrightarrow[\textbf{rule 0}]{} CS_0 = \{p_0\} \xrightarrow[\textbf{rule 1}]{\text{coin}|\text{abs}} CS_1 = \{p_1\} \xrightarrow[\textbf{rule 5}]{\text{coffee}|\text{served}} \mathsf{WeakPass}$$

The verdict WeakPass is due to the two equalities:

- $NextPass(CS_1, \text{coffee}|\text{served}) \subset Next(CS_1, \text{coffee}|\text{served})$
- $NextPass(CS_1, \text{coffee}|\text{served}) = \{p_2\} \neq \emptyset$

Test cases for TP_2

Fail To get the verdict Fail, we consider the following trace:

$$[\text{coin}|\text{abs}, \text{coffee}|\text{served}, \text{repair}|\text{refund} \mid \mathsf{Fail}]$$

The algorithm is applied as follows:

$$\xrightarrow[\textbf{rule 0}]{} CS_0 = \{p_0\} \xrightarrow[\textbf{rule 1}]{\text{coin}|\text{abs}} CS_1 = \{p_1\} \xrightarrow[\textbf{rule 1}]{\text{coffee}|\text{served}} CS_2 = \{p_2, p_3\}$$

$$CS_2 \xrightarrow[\textbf{rule 2}]{\text{repair}|\text{refund}} \mathsf{Fail}$$

The verdict Fail is due to the following equality:

$$Next(CS_2, \text{repair}|\text{refund}) = \emptyset$$

Inconc To get the verdict Inconc, we consider the following trace:

$$[\text{coin}|\text{abs}, \text{coffee}|\text{served}, \text{coin}|\text{abs} \mid \text{Inconc}]$$

The algorithm is applied as follows:

$$\xrightarrow[\textbf{rule 0}]{} CS_0 = \{p_0\} \xrightarrow[\textbf{rule 1}]{\text{coin}|\text{abs}} CS_1 = \{p_1\} \xrightarrow[\textbf{rule 1}]{\text{coffee}|\text{served}} CS_2 = \{p_2, p_3\}$$

$$CS_2 \xrightarrow[\textbf{rule 3}]{\text{coin}|\text{abs}} \text{Inconc}$$

The verdict Inconc is due to the following two equalities:

- $Next(CS_2, \text{coin}|\text{abs}) = \{p_6\} \neq \emptyset$
- $NextPass(CS_2, \text{coin}|\text{abs}) = NextSkip(CS_3, \text{coin}|\text{abs}) = \emptyset$

Pass To get the verdict Pass, we consider the following trace:

$$[\text{coin}|\text{abs}, \text{coffee}|\text{served}, \text{repair}|\text{abs} \mid \text{Pass}]$$

The algorithm is applied as follows:

$$\xrightarrow[\textbf{rule 0}]{} CS_0 = \{p_0\} \xrightarrow[\textbf{rule 1}]{\text{coin}|\text{abs}} CS_1 = \{p_1\} \xrightarrow[\textbf{rule 1}]{\text{coffee}|\text{served}} CS_2 = \{p_2, p_3\}$$

$$CS_2 \xrightarrow[\textbf{rule 4}]{\text{repair}|\text{abs}} \text{Pass}$$

The verdict Pass is due to the following equality:

$$NextPass(CS_2, \text{repair}|\text{abs}) = Next(CS_2, \text{repair}|\text{abs}),\ Next(CS_2, \text{repair}|\text{abs}) \neq \emptyset$$

WeakPass There is no test cases ending by the verdict WeakPass for TP_2.

4.4 Properties

A test case informs us of the conformance of the implementation to its specification. The non-existence of a Fail verdict leads to a conformance, and any non-conformance should be detected by a test case ending by a Fail verdict. In order to study the coherence between the notion of conformance applied to an implementation under test and its specification, and the notion of a test case generated by our algorithm, we denote by $\mathbb{CS}$ and $\mathbb{EV}$ respectively the whole set of current state sets and the whole set of input-output elementary sequences used during the application of the set of inference rules on an implementation iut according to a test purpose TP. We then introduce a transition system whose states are the sets of current states and four special states labeled by the verdicts. Two states are linked by a transition labeled by an input-output elementary sequence. This transition system is formally defined as follows:

Definition 4.2 ***(Execution)*** *Let* TP *be a test purpose for a specification* spec*, let* iut *be an implementation, let* $\mathbb{CS}$ *be the whole set of current state sets and let* $\mathbb{EV}$ *be the whole set of input-output elementary sequences. Then,* ***the execution of the test generation algorithm*** *on* iut *according to* TP *denoted by* $TS(\mathsf{TP}, \mathsf{iut})$ *(see its explanation in Section 4.2) is the coalgebra* (S_{TS}, α_{TS}) *over the signature* $(\_)^{\mathbb{EV}}$ *defined by:*

- $S_{TS} = \mathbb{CS} \cup \mathbb{V}$ *where* $\mathbb{V}$ *is the set whose elements are* Fail, Pass, Inconc *and* WeakPass*;*
- α_{TS} *is the mapping which for every* $CS \in \mathbb{CS}$ *and for every* $ev \in \mathbb{EV}$ *is defined as follows:*

$$\alpha_{TS}(CS)(ev) = \begin{cases} Next(CS,ev) & \text{if} \quad NextSkip(CS,ev) \neq \emptyset, NextPass(CS,ev) = \emptyset \\ \mathsf{Fail} & \text{if} \quad Next(CS,ev) = \emptyset \\ \mathsf{Inconc} & \text{if} \quad NextSkip(CS,ev) = NextPass(CS,ev) = \emptyset \\ & \quad \text{and } Next(CS,ev) \neq \emptyset \\ \mathsf{Pass} & \text{if} \quad Next(CS,ev) = NextPass(CS,ev) \\ & \quad \text{and } Next(CS,ev) \neq \emptyset \\ \mathsf{WeakPass} & \text{if} \quad NextPass(CS,ev) \subsetneq Next(CS,ev) \\ & \quad \text{and } NextPASS(CS,ev) \neq \emptyset \end{cases}$$

With this definition, test cases are sets of possible traces which can be observed during an execution of $TS(\mathsf{TP}, \mathsf{iut})$, and lead to a verdict state.

Definition 4.3 ***(Test cases)*** *Let* $TS(\mathsf{TP}, \mathsf{iut}) = (S_{TS}, \alpha_{TS})$ *be the execution of the test generation algorithm on* iut *according to* TP*. A* ***test case*** *for* TP *is a sequence* $[ev_0, \ldots, ev_n | V]$ *for which there is a sequence of states* $s_0, \ldots, s_n \in \mathbb{CS}$ *with* $\forall j, 0 \leq j < n, s_{j+1} = \alpha_{TS}(s_j)(ev_j)$*, and there is a verdict state* $V \in \mathbb{V}$ *such that* $V = \alpha_{TS}(s_n)(ev_n)$*.*

We note $st(\mathsf{TP}, \mathsf{iut})$ *the set of all possible test cases for* TP*.*

We can now introduce the notation:

$$vdt(\mathsf{TP}, \mathsf{iut}) = \{V \mid \exists ev_0, \ldots, ev_n, [ev_0, \ldots, ev_n | V] \in st(\mathsf{TP}, \mathsf{iut})\}$$

Theorm 4.1 ***(Correctness and completeness)*** *For any specification* spec *and any* iut*:*

- ***Correctness:*** *If* iut *conforms to* spec*, then for any test purpose* TP*,* $\mathsf{Fail} \notin vdt(\mathsf{TP}, \mathsf{iut})$*.*
- ***Completeness:*** *If* iut *does not conform to* spec*, then there exists a test purpose* TP *such that* $\mathsf{Fail} \in vdt(\mathsf{TP}, \mathsf{iut})$*.*

Proof
Proof of the correctness: *Let* $\mathsf{spec} = (S, s_0, \alpha)$ *be a specification over a signature* $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$ *and* $FCT = (S_{FCT}, s^0_{FCT}, \alpha_{FCT})$ *be its finite computation tree. Let us prove the correctness using the contraposition principle. This means that to prove:*

$$\text{if iut conforms to spec, for any test purpose TP, } \mathsf{Fail} \notin vdt(\mathsf{TP}, \mathsf{iut}).$$

we have to prove:

$$\text{if there exists a test purpose TP such that } \mathsf{Fail} \in vdt(\mathsf{TP}, \mathsf{iut}), \text{ then}$$

$$\text{iut does not conform } \textit{w.r.t cioco} \text{ to spec.}$$

More precisely, according to the definition of cioco, we have to prove that:

$$\text{there exists a finite trace } tr \in Trace(FCT), \text{an input } i \in \mathsf{In} \text{ such that}$$

$$Out_{\mathsf{iut}}(\mathsf{iut} \text{ after } (tr, i)) \not\subseteq Out_{FCT}(FCT \text{ after } (tr, i))$$

This is proved by the following proposition:

Proposition 4.1 *If there exists a test purpose* TP *such that* $[i_0|o_0, \ldots, i_n|o_n|\mathsf{Fail}] \in st(\mathsf{TP}, \mathsf{iut})$*, then:*

1. $\langle i_0|o_0, i_1|o_1, \ldots, i_{n-1}|o_{n-1}\rangle \in Trace(FCT)$.
2. $i_n \in \mathsf{In}$
3. $o_n \in Out_{\mathsf{iut}}(\mathsf{iut} \text{ after } (\langle i_0|o_0, \ldots, i_{n-1}|o_{n-1}\rangle, i_n))$.
4. $o_n \notin Out_{\mathsf{spec}}(\mathsf{spec} \text{ after } (\langle i_0|o_0, \ldots, i_{n-1}|o_{n-1}\rangle, i_n))$.

First of all, let us denote $\langle i_0|o_0 \ldots i_{n-1}|o_{n-1}\rangle$ *by* $\langle ev_0 \ldots ev_{n-1}\rangle$.

Proof of (1).
In order to show that the sequence $\langle i_0|o_0 \ldots i_{n-1}|o_{n-1}\rangle \in Trace(FCT)$*, we are going to reason on the way of computation of this sequence by using the inference rules. First of all, let* $TS(\mathsf{TP}, \mathsf{iut})$ *be the execution of the test generation algorithm and* $st(\mathsf{TP}, \mathsf{iut})$ *be the set of generated test cases. Since* $[i_0|o_0, \ldots, i_n|o_n|\mathsf{Fail}] \in st(\mathsf{TP}, \mathsf{iut})$*, then*

$$\text{there exists } \forall j, 0 \leq j < n, S_j \in \mathbb{CS} \text{ such that } S_0 = \{s^0_{TS}\}, S_{j+1} = \alpha_{TS}(S_j)(ev_j) \text{ and } \mathsf{Fail} = \alpha_{TS}(S_n)(ev_n)$$

Hence, for every $j, 0 \leq j < n, S_{j+1}$ *which equals to* $Next(S_j, ev_j)$ *is not empty by Definition 4.2. Hence, by Definition 4.1, for every* $j, 0 \leq j < n$*, every state belonging into* S_{j+1} *is a state of FCT. This means that for every* $j, 0 \leq j < n$*, every state* $s \in S_j$ *is related to a state* $s' \in S_{j+1}$ *by* ev_j*. Then, the sequence* $\langle ev_0 \ldots ev_j \ldots ev_{n-1}\rangle \in Trace(FCT)$.

Proof of (2).
We have proved above that $\langle i_0|o_0 \ldots i_{n-1}|o_{n-1}\rangle \in Trace(FCT)$ *and* $S_n \neq \emptyset$.
We have that $[i_0|o_0 \ldots i_n|o_n|\mathsf{Fail}] \in st(\mathsf{TP}, \mathsf{iut})$ i.e. *submitting the input* i_n *to the implementation under test will produce the output* o_n *that is not specified in* $FCT(\mathcal{C})$*. Then, it is clear that* $i \in \mathsf{In}$ *is an input of* $FCT(\mathsf{spec})$.

Proof of (3).
It is obvious because $[i_0|o_0, i_1|o_1, \ldots, i_n|o_n|\mathsf{Fail}] \in st(\mathsf{TP}, \mathsf{iut})$.

Proof of (4).
We know that $\langle i_0|o_0 \ldots i_{n-1}|o_{n-1}\rangle \in Trace(FCT)$ *and* $S_n \neq \emptyset$*. We have that* $[i_0|o_0 \ldots i_n|o_n|\mathsf{Fail}] \in st(\mathsf{TP}, \mathsf{iut})$ i.e. *applying* $i_n|o_n$ *has to lead to a* Fail *verdict. This means that* $\alpha_{TS}(S_n)(i_n|o_n) = \mathsf{Fail}$*. Hence by Definition 4.2,* $Next(S_n, i_n|o_n)$ *has to be empty. But we know that* $Next(S_n, i_n|o_n) \subseteq S_{FCT}$*. Hence,* $\langle i_0|o_0, \ldots, i_{n-1}|o_{n-1}, i_n|o_n\rangle$ *does not belong to* $Trace(FCT)$.

Proof of the completeness : *Let* $\mathsf{spec} = (S, s_0, \alpha)$ *be a specification over a signature* $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$ *and* $FCT = (S_{FCT}, s^0_{FCT}, \alpha_{FCT})$ *be its finite computation tree. Let us prove that the completeness holds.*

For this, let us assume that iut *does not conform to* spec *and let us prove that there exists a test purpose* TP *such that there exists* $[ev_0, \ldots, ev_n|\mathsf{Fail}] \in st(\mathsf{TP}, \mathsf{iut})$.
First of all, iut *does not conform to* spec. *According to the definition of cioco, there exists a trace* $tr = \langle ev_0 \ldots ev_{n-1} \rangle \in Trace(FCT)$ *and an input* $i \in \mathsf{In}$ *such that*

$$Out_{\mathsf{iut}}(\mathsf{iut} \text{ after } (tr, i)) \nsubseteq Out_{FCT}(FCT \text{ after } (tr, i))$$

i.e. *there exists an output* o'_n *of* iut *such that*
- $o'_n \in Out_{\mathsf{iut}}(\mathsf{iut} \text{ after } (tr, i_n))$;
- $o'_n \notin Out_{FCT}(FCT \text{ after } (tr, i_n))$.

That means:

$$\langle ev_0, \ldots, ev_{n-1}, i_n|o'_n \rangle \in Trace(\mathsf{iut}) \tag{VII.2}$$

and

$$\langle ev_0, \ldots, ev_{n-1}, i_n|o'_n \rangle \notin Trace(FCT) \tag{VII.3}$$

Since $i_n \in \mathsf{In}$, *then there also exists an output* o_n *such that* $o_n \in Out_{FCT}(FCT \text{ after } (tr, i_n))$ *i.e.*

$$\langle ev_0, \ldots, ev_{n-1}, i_n|o_n \rangle \in Trace(FCT) \tag{VII.4}$$

Let us denote $\langle i_n|o_n \rangle$ *by* ev_n *and* $\langle i_n|o'_n \rangle$ *by* ev'_n.

Now, let us denote by TP *a test purpose of FCT such that there exists a state* $s \in S_{FTC}$ *such that s belongs to the set of reachable states from the initial state of FCT after executing the trace* $\langle ev_0 \ldots ev_{n-1} ev_n \rangle$ *on FCT, and* $\mathsf{TP}(s) =$ ***accept*** i.e. $\langle ev_0 \ldots ev_{n-1} ev_n \rangle$ *forms a path of* TP. *Let us prove that there exists* $[ev_0 \ldots ev_{n-1} ev'_n|\mathsf{Fail}] \in st(\mathsf{TP}, \mathsf{iut})$. *For this, it is enough to show that*

$$\exists (S_j)_{0 \leq j \leq n} \text{ such that } \forall j, 0 \leq j < n, S_{j+1} = \alpha_{TS}(S_j)(ev_j) \in \mathbb{CS} \text{ and } \mathsf{Fail} = \alpha_{TS}(S_n)(ev'_n)$$

We have that $\langle ev_0 \ldots ev_{n-1} \rangle \in Trace(FCT)$, *then, for every* $j, 0 \leq j < n, S_j$ *exists because for every* $j, 1 \leq j < n, \alpha_{TS}(S_j)(ev_j) = Next(S_j, ev_j)$ *and* $S_0 = \{s^0_{FCT}\}$. *Thus, what remains is to prove that there is a verdict state* Fail *such that* $\mathsf{Fail} = \alpha_{TS}(S_n)(ev_n)$.
By Equation VII.3, $\langle ev_0 \ldots ev'_n \rangle \notin Trace(FCT)$ *and by Equation VII.2* $\langle ev_0 \ldots ev_n \rangle \in Trace(\mathsf{iut})$, *hence* $Next(S_n, ev'_n) = \emptyset$, *and consequently* $\alpha_{TS}(S_n)(ev_n) = \mathsf{Fail}$.
End

5 Instantiating of the approach

From the genericity of our framework, the testing technique proposed in this chapter, can also be applicable to any-state formalisms which are instances of our framework. Figure VII.8 illustrates the different steps to generate correct test cases for any model instance of our framework.
First, we define the function ϕ which transforms the model in which a specification is given (for example, an *IOLTS* or Mealy automaton) into our framework. This function has to be bijective to allow us to go back to the original formalism. This transformation is illustrated on the left side in Figure VII.8. Second, we need to prove the equivalence between the conformance relation **rel** used between iut and spec and the *cioco* defined on their images $\phi(\mathsf{iut})$ and $\phi(\mathsf{spec})$. Third, we apply our proposed algorithm for test case derivation on $\phi(\mathsf{spec})$ to generate correct test cases. Finally, we transform the obtained test cases into test cases described in the original formalism. Hence, testing systems, whose specification models can be viewed as instances of our component definition, require us to define the bijective transformation ϕ and to prove that the following property holds:

$$\mathsf{iut} \ \mathbf{rel} \ \mathsf{spec} \iff \phi(\mathcal{M}_1) \ cioco \ \phi(\mathcal{M}_2)$$

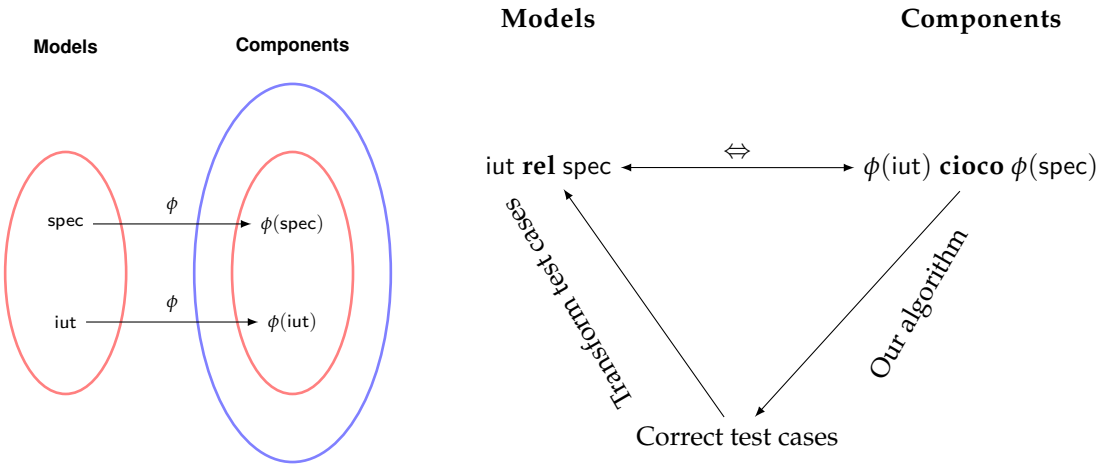


Figure VII.8 – Instantiating of the algorithm

Chapter VIII

Integration Testing

1 Compositional testing 164
1.1 Compositional testing with cioco 164
1.2 Compositionality for cartesian product 168
1.3 Compositionality for feedback operators 168
1.4 Compositionality for complex operator 172
2 Test purposes for sub-systems 173
2.1 Sub-systems and projection 174
2.2 System-based test purposes 175
3 Related works 177

In Chapter VII, we have shown how component implementations are tested to be *cioco*-correct to their component specifications separately. This is classically known as *unitary testing*. The present chapter intends to validate the complex systems made by assembling a set of state-based components. This is classically known as *compositional testing* (or *component based testing*). Due to the growing complexity of the space state of the complex systems, it is difficult even impossible to use the black-box testing approach that we proposed in Chapter VII. Therefore, there is a need to systematically derive test cases based on the structure of the complex system. We believe that it is natural and easier to test a system by testing only its subsystems.

Hence, in this chapter we intend to contribute in two ways:

1. By defining a compositional testing approach. The main idea is to test an integrated system assuming that its underlying components have already been tested in isolation and are correct [30]. The operators used to compose components are assumed to be well-implemented and to preserve their specifications. Thus, the problem of compositional testing that we address can be seen as follows: if single components of a system conform to their specifications, can we conclude that the whole system is in conformance to its specification? As a consequence of positive answer to this question, we can test the global system by testing in isolation its components that may be done at various steps of development and potentially developed by different teams.
2. By strengthening the quality of components by taking into account their involvement in the global system that encapsulates them. The main idea consists in showing how to strengthen the correctness of each component involved in a global system, by choosing suitable test purposes for them. This will be done by defining a projection mechanism that, from a behaviour of the global system, will help to generate test purposes capturing behaviours of sub-systems that typically occur in the whole system [48].

In Section 1, we study the compositional testing problem in our framework by explaining and formalizing this problem. In Section 2, we study the projection mechanism by showing how to define test purposes from global behaviour of a system and how to project them on any sub-system of it. Finally, in Section 3, comparisons with existing works close to our modeling framework will be done.

1 Compositional testing

Compositional testing consists in testing communicating components that have been tested separately. It aims to guarantee the correctness of the integration of a set of components $\mathcal{S} = op(\mathcal{C}_1, \ldots, \mathcal{C}_n)$ from the correctness of each components $\mathcal{C}_i$ in isolation where op is the integration operator of interest. Thus, such a compositional testing theory provides a way to test the integrated system only by testing its sub-systems i.e. there is no need to re-test its conformance correction. It is formally expressed as follows:

Given implementation models $\mathsf{iut}_1, \ldots, \mathsf{iut}_n$ *and their specifications* $\mathsf{spec}_1, \ldots, \mathsf{spec}_n$

$$\forall i, 1 \leq i \leq n, (\mathsf{iut}_1 \ \mathbf{rel} \ \mathsf{spec}_1), \ldots, (\mathsf{iut}_n \ \mathbf{rel} \ \mathsf{spec}_n) \Longrightarrow op(\mathsf{iut}_1, \ldots, \mathsf{iut}_n) \ \mathbf{rel} \ op(\mathsf{spec}_1, \ldots, \mathsf{spec}_n)$$

where **rel** *and* op *denote the conformance relation and the integration operator of interest respectively.*

Hence, once this property is verified, the correctness of the integrated system is obtained from the correctness of the individual components. To test the integrated system, it is not necessary to consider it as a whole, but it is enough to consider its sub-systems and test them separately. Indeed, the contraposition of this property is the following:

$$\neg\Big(op(\mathsf{iut}_1, \ldots, \mathsf{iut}_n) \ \mathbf{rel} \ op(\mathsf{spec}_1, \ldots, \mathsf{spec}_n)\Big) \Longrightarrow \exists i, 1 \leq i \leq n, \neg(\mathsf{iut}_i \ \mathbf{rel} \ \mathsf{spec}_i)$$

Thus, by looking at this new property, we can easily see that non-correctness of the integrated system under test $op(\mathsf{iut}_1, \ldots, \mathsf{iut}_n)$ implies that at least one of its components $\mathsf{iut}_1, \ldots, \mathsf{iut}_n$ is incorrect. In other words, that means to test $op(\mathsf{iut}_1, \ldots, \mathsf{iut}_n)$, it suffices to test $\mathsf{iut}_1, \ldots, \mathsf{iut}_n$ in isolation.

1.1 Compositional testing with cioco

In this subsection, we study the compositional testing problem in our framework. Then, we intend to address the following question:

Given that the components $\mathcal{C}_1, \ldots, \mathcal{C}_n$ *over the signatures* $H_1 = T(\mathsf{Out}_1 \times \_)^{\mathsf{In}_1}, \ldots, H_n = T(\mathsf{Out}_n \times \_)^{\mathsf{In}_n}$ *respectively, are cioco-correct*[1] *separately, may we conclude that their integration* $\mathcal{C} = op(\mathcal{C}_1, \ldots, \mathcal{C}_n)$ *using a complex operator*[2] op *is also cioco-correct?*

The response to this question amounts to first addressing both for cartesian product and feedback operator, and then by showing *cioco* correctness is stable for its composition. Hence, in this following we intend to give answers to the following three questions:

Question 1:

$$\text{Given } (\mathsf{iut}_k \ \textit{cioco} \ \mathsf{spec}_k) \text{ for } k = 1, 2,$$
$$\text{is it the case of } \otimes(\mathsf{iut}_1, \mathsf{iut}_2) \ \textit{cioco} \ \otimes(\mathsf{spec}_1, \mathsf{spec}_2)?$$

1. For instance using our conformance testing approach defined in Chapter VII.
2. See Chapter V, for the definition of a complex operator.

Question 2:

$$\text{Given } (\mathsf{iut}\ \textit{cioco}\ \mathsf{spec}), \text{ is it the case of } \hookleftarrow_{\mathcal{I}}(\mathsf{iut})\ \textit{cioco}\ \hookleftarrow_{\mathcal{I}}(\mathsf{spec})?$$

Question 3:

$$\text{Given } (\mathsf{iut}\ \textit{cioco}\ \mathsf{spec}), \text{ is it the case of } \circlearrowleft_{\mathcal{I}}(\mathsf{iut})\ \textit{cioco}\ \circlearrowleft_{\mathcal{I}}(\mathsf{spec})?$$

In the following subsections we will show that the answer to **Question 1** is positive without imposing any conditions i.e. *cioco* is naturally compositional for the cartesian product. However, the answer to both **Question 2** and **Question 3**, in general, is negative. To a get positive answer, the specification should be input-enabled. In other words, compositionality does not hold for *cioco* with respect to the feedback operators, unless the specification model is input-enabled. Without this condition, even if both iut_1 and iut_2 are *cioco*-correct, the resulting implementation obtained by means of feedback operators may not be.

Example 1.1 *To illustrate our compositional testing, we consider two components*[3] *of a coffee machine: a "*money component*" $\mathcal{M}$ that handles the inserted coins and "*drink component*" $\mathcal{D}$ that produces the drinks.*

Figure VIII.1 illustrates the architecture of these two components.

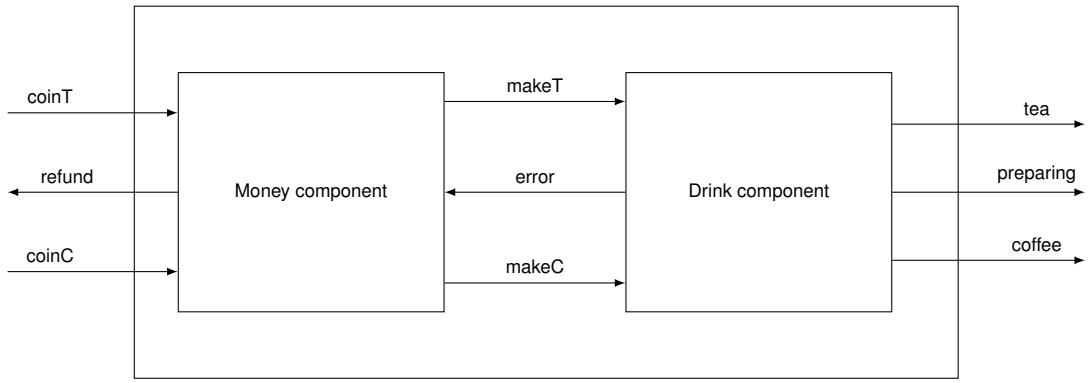


Figure VIII.1 – Architecture of a coffee machine in components

We use the following specifications and implementations of $\mathcal{M}$ and $\mathcal{D}$:

Money component specification $\mathsf{spec}_{\mathcal{M}}$***:***

when it receives a coffee coin "coinC" (resp. a tea coin "coinT") from the user, it gives an order "makeC" (resp. "makeT") to the drink component $\mathcal{D}$ to make coffee (resp. tea).

Drink component specification $\mathsf{spec}_{\mathcal{D}}$***:***

when it receives the order "makeC" (resp. "makeT") to make coffee (resp. tea) from the money component $\mathcal{M}$, if there is nothing wrong during the drink preparation process, it directly delivers the coffee (resp. tea) to the user, or else it sends an error message to the money component in order to refund the user.

3. This example is inspired from the example presented in [30].

Money component implementation $\mathsf{iut}_{\mathcal{M}}$***:***

it behaves as the money component[4] *specification* $\mathsf{spec}_{\mathcal{M}}$*, but in addition it does some extra functionalities, that is if an error occurs during the drink preparation process, it refunds the inserted coin to the user.*

Drink component implementation $\mathsf{iut}_{\mathcal{D}}$***:***

it behaves exactly as the drink component specification $\mathsf{spec}_{\mathcal{D}}$*.*

In our framework, $\mathsf{spec}_{\mathcal{M}}, \mathsf{iut}_{\mathcal{M}}, \mathsf{spec}_{\mathcal{D}}$ *and* $\mathsf{iut}_{\mathcal{D}}$ *are modeled as follows:*

- $\mathsf{spec}_{\mathcal{M}}$ *is the coalgebra* $(\{q_0\}, q_0, \alpha_1)$ *over the signature*

$$(\{\text{makeC}, \text{makeT}\} \times \_)^{\{\text{coinC},\text{coinT}\}}$$

  where α_1 *is depicted in Figure VIII.2a.*

- $\mathsf{iut}_{\mathcal{M}}$ *is the coalgebra* $(\{q'_0, q'_1\}, q'_0, \alpha'_1)$ *over the signature*

$$(\{\text{makeC}, \text{makeT}, \text{refund}\} \times \_)^{\{\text{coinC},\text{coinT},\text{error}\}}$$

  where α'_1 *is depicted in Figure VIII.2c.*

- $\mathsf{spec}_{\mathcal{D}}$ *is the coalgebra* $(\{s_0, s_1, s_2, s_3, s_4\}, s_0, \alpha_2)$ *over the signature*

$$(\{\text{error}, \text{tea}, \text{coffee}, \text{preparing}\} \times \_)^{\{\text{makeC},\text{makeT}\}}$$

  where α_2 *is depicted in Figure VIII.2b.*

- $\mathsf{iut}_{\mathcal{D}}$ *is the coalgebra* $(\{s'_0, s'_1, s'_2, s'_3, s'_4\}, s'_0, \alpha'_2)$ *over the signature*

$$(\{\text{error}, \text{tea}, \text{coffee}, \text{preparing}\} \times \_)^{\{\text{makeC},\text{makeT}\}}$$

  where α'_2 *is depicted in Figure VIII.2d.*

The components $\mathcal{M}$ *and* $\mathcal{D}$ *may communicate separately (for instance* $\mathcal{D}$ *may execute the transition labeled with* $\mathsf{abs}|\text{coffee}$ *while* $\mathcal{M}$ *does nothing) or jointly in synchronization (for instance when* $\mathcal{M}$ *execute the transition labeled with* $\text{coinC}|\text{makeC}$*,* $\mathcal{M}$ *receives instantaneously the output* makeC *and then produces the output* coffee*). Then, the suitable composition of* $\mathcal{M}$ *and* $\mathcal{D}$ *is the synchronous parallel composition* $\odot$ *defined in Chapter V, Section 2.5.*

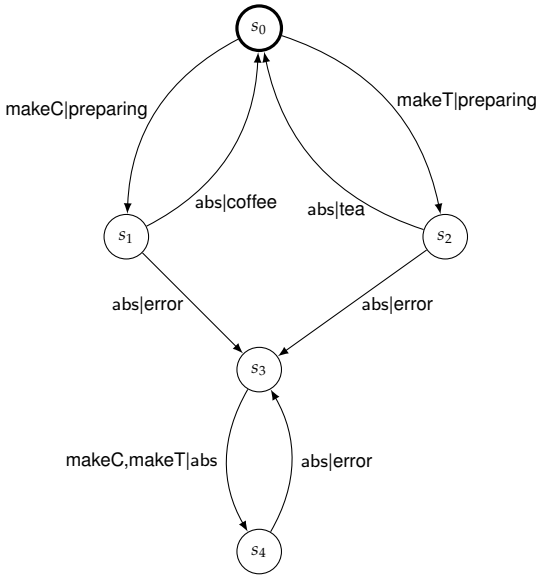
As far as the compositional testing is concerned, we have that

$$(\mathsf{iut}_{\mathcal{M}} \textit{ cioco } \mathsf{spec}_{\mathcal{M}}) \textit{ and } (\mathsf{iut}_{\mathcal{D}} \textit{ cioco } \mathsf{spec}_{\mathcal{D}})$$

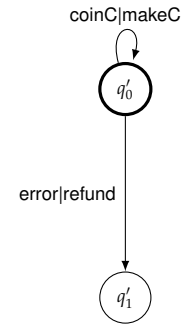
Our goal is to know if this is enough to ensure whether the global implementation $\odot(\mathsf{iut}_{\mathcal{M}}, \mathsf{iut}_{\mathcal{D}})$ *is in conformance with respect to cioco to the global specification* $\odot(\mathsf{spec}_{\mathcal{M}}, \mathsf{spec}_{\mathcal{D}})$*. Hence, to test* $\odot(\mathsf{iut}_{\mathcal{M}}, \mathsf{iut}_{\mathcal{D}})$*, it suffices to test if* $(\mathsf{iut}_{\mathcal{M}} \textit{ cioco } \mathsf{spec}_{\mathcal{M}})$ *and* $(\mathsf{iut}_{\mathcal{D}} \textit{ cioco } \mathsf{spec}_{\mathcal{D}})$*. An answer to this question is given later in this chapter.*

4. For the sake of readability, input completeness (implementations) are not depicted in Figure VIII.2c and Figure VIII.2d.

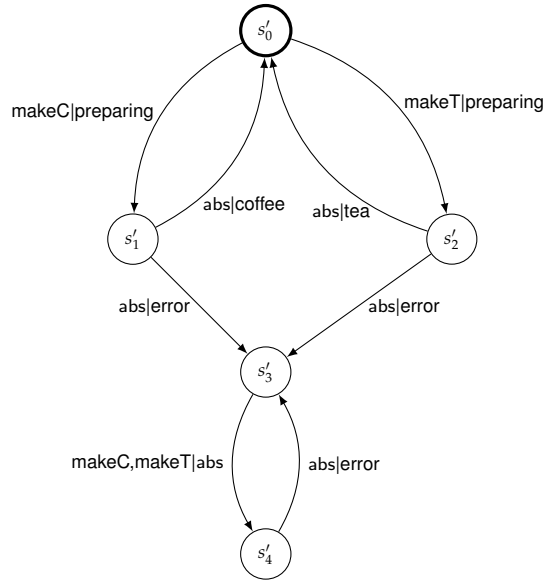
(a) money component specification $\mathsf{spec}_{\mathcal{M}}$



(b) drink component specification $\mathsf{spec}_{\mathcal{D}}$



(c) money component implementation $\mathsf{iut}_{\mathcal{M}}$



(d) drink component implementation $\mathsf{iut}_{\mathcal{D}}$

Figure VIII.2 – Illustration of *cioco*'s compositionality

1.2 Compositionality for cartesian product

We show here that *cioco* is naturally preserved by the cartesian product.

Theorm 1.1 *Let* $H_1 = T(\mathsf{Out}_1 \times \_)^{\mathsf{In}_1}$ *and* $H_2 = T(\mathsf{Out}_2 \times \_)^{\mathsf{In}_1}$ *be two signatures.*
Let $H = T((\mathsf{Out}_1 \times \mathsf{Out}_2) \times \_)^{\mathsf{In}_1 \times \mathsf{In}_2}$ *be the cartesian product interface for* H_1 *and* H_2. *Let* $\mathsf{iut}_j, \mathsf{spec}_j \in \mathbf{Comp}(H_j)$ *for* $j = 1, 2$ *and* $\otimes((\mathsf{iut}_1, \mathsf{iut}_2))$, $\otimes((\mathsf{spec}_1, \mathsf{spec}_2)) \in \mathbf{Comp}(H)$. *Then, we have:*

$$\left.\begin{array}{l} \mathsf{iut}_1 \textit{ cioco } \mathsf{spec}_1 \\ \\ \mathsf{iut}_2 \textit{ cioco } \mathsf{spec}_2 \end{array}\right\} \Longrightarrow \otimes((\mathsf{iut}_1, \mathsf{iut}_2)) \textit{ cioco } \otimes((\mathsf{spec}_1, \mathsf{spec}_2))$$

Proof *Let us assume that:*

$$(\mathsf{iut}_1 \textit{ cioco } \mathsf{spec}_1) \text{ and } (\mathsf{iut}_2 \textit{ cioco } \mathsf{spec}_2)$$

and let us then prove that:

$$\otimes((\mathsf{iut}_1, \mathsf{iut}_2)) \textit{ cioco } \otimes((\mathsf{spec}_1, \mathsf{spec}_2))$$

Let us use the contradiction principle. For this, let us assume that

$$\neg(\otimes((\mathsf{iut}_1, \mathsf{iut}_2)) \textit{ cioco } \otimes((\mathsf{spec}_1, \mathsf{spec}_2)))$$

i.e. there exists a finite trace $tr = \langle (i_1, i_1')|(o_1, o_1'), \ldots, (i_n, i_n')|(o_n, o_n')\rangle \in Trace(\otimes((\mathsf{spec}_1, \mathsf{spec}_2)))$ *and* $(i, i') \in \mathsf{In}_1 \times \mathsf{In}_2$ *such that there exists an output* $(o, o') \in \mathsf{Out}_1 \times \mathsf{Out}_2$ *among the outputs obtained after executing* $(tr, (i, i'))$ *on* $\otimes((\mathsf{iut}_1, \mathsf{iut}_2))$ *not belonging to the ones obtained after executing* $(tr, (i, i'))$ *on* $\otimes((\mathsf{spec}_1, \mathsf{spec}_2))$.

Now, we have $tr = \langle (i_1, i_1')|(o_1, o_1'), \ldots, (i_n, i_n')|(o_n, o_n')\rangle \in Trace(\otimes((\mathsf{iut}_1, \mathsf{iut}_2)))$. *According to the cartesian product definition, it is easy to show that the two traces:*

$$tr_1 = \langle i_1|o_1, \ldots, i_n|o_n\rangle \in Trace(\mathsf{iut}_1) \;\; and \;\; tr_2 = \langle i_1'|o_1', \ldots, i_n'|o_n'\rangle \in Trace(\mathsf{iut}_2)$$

are respectively the traces involved in iut_1 *and* iut_2 *to obtain tr. We also know by hypothesis that* $tr_1 \in Trace(\mathsf{spec}_1)$ *and* $tr_2 \in Trace(\mathsf{spec}_2)$.

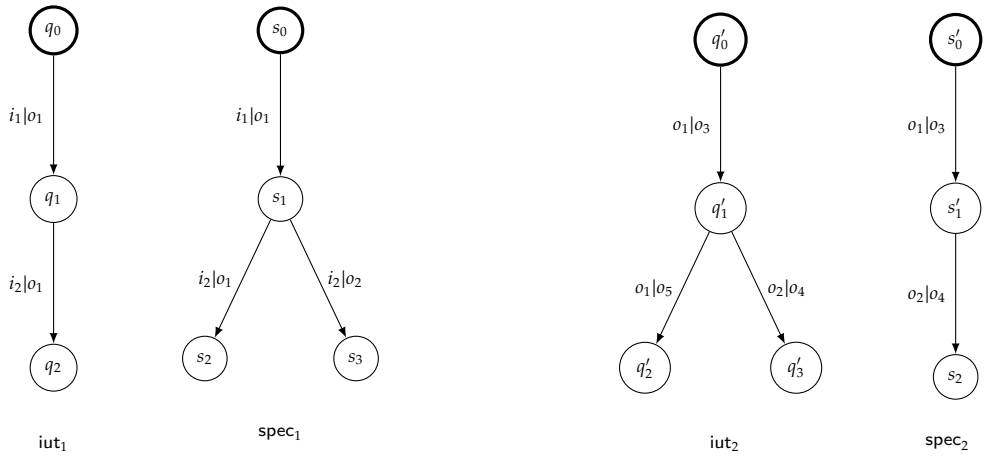
Since $(o, o') \in Out(\otimes((\mathsf{iut}_1, \mathsf{iut}_2)) \text{ after } (tr, (i, i')))$ *and* tr_1 *and* tr_2 *are used to obtain tr, then* $o \in Out(\mathsf{iut}_1 \text{ after } (tr_1, i))$ *and* $o' \in Out(\mathsf{iut}_2 \text{ after } (tr_2, i'))$. *Similarly,* $o \notin Out(\mathsf{spec}_1 \text{ after } (tr_1, i))$ *and* $o' \notin Out(\mathsf{spec}_2 \text{ after } (tr_2, i'))$ *because* $(o, o') \notin Out(\otimes((\mathsf{spec}_1, \mathsf{spec}_2)) \text{ after } (tr, (i, i')))$ *and* tr_1 *and* tr_2 *are used to obtain tr. Hence, there exists a trace* $tr_1 \in Trace(\mathsf{spec}_1)$, *an input i of* spec_1 *and an output* $o \in \mathsf{Out}_1$ *such that* $o \in Out(\mathsf{iut}_1 \text{ after } (tr_1, i))$ *and* $o \notin Out(\mathsf{spec}_1 \text{ after } (tr_1, i))$ *(respectively there exists a trace* $tr_2 \in Trace(\mathsf{spec}_2)$, *and input* i' *of* spec_2 *and an output* $o' \in \mathsf{Out}_2$ *such that* $o' \in Out(\mathsf{iut}_2 \text{ after } (tr_2, i'))$ *and* $o' \notin Out(\mathsf{spec}_2 \text{ after } (tr_2, i'))$*). Indeed, this means that* $\neg(\mathsf{iut}_1 \textit{ cioco } \mathsf{spec}_1)$ *and* $\neg(\mathsf{iut}_2 \textit{ cioco } \mathsf{spec}_2)$. *Hence, we have a contradiction with our hypothesis.*
End

1.3 Compositionality for feedback operators

We show here that the compositionality of *cioco* for both synchronous and relaxed feedback operators cannot be obtained without any assumptions made on both specifications and implementations.
We first give an example that illustrates the assumptions required to obtain the compositionality of *cioco* with respect to the feedback operators.

Example 1.2 *Figure VIII.3 shows two implementation models* iut_1 *and* iut_2 *that have been tested to be* ***cioco****-correct according to their respective specification models* spec_1 *and* spec_2*. It is easy to see that*

$$(\mathsf{iut}_1 \; \textit{cioco} \; \mathsf{spec}_1) \;\; \textit{and} \;\; (\mathsf{iut}_2 \; \textit{cioco} \; \mathsf{spec}_2)$$



(a) iut_1 *cioco* spec_1 (b) iut_2 *cioco* spec_2

Figure VIII.3 – Counterexample of compositionality

Let us now compose sequentially iut_1 *with* iut_2 *and* spec_1 *with* spec_2*, but first let us recall the feedback operator over the synchronous sequential interface* $\mathcal{I} = (f, \pi_i, \pi_o)$ *defined in Chapter V, Section 2.1. Then,* $\mathcal{I} = (f, \pi_i, \pi_o)$ *is the feedback interface defined for every* $(i, i') \in \mathsf{In}_1 \times \mathsf{In}_2$ *and* $(o, o') \in \mathsf{Out}_1 \times \mathsf{Out}_2$ *as follows:*

$$f((i,i'),(o,o')) = (i,o), \quad \pi_i((i,i')) = i \;\; \text{and} \;\; \pi_o((o,o')) = o'$$

Now, using the cartesian product and the feedback operator over the synchronous sequential interface $\mathcal{I} = (f, \pi_i, \pi_o)$ *defined above, the global implementation* $\mathsf{iut} = \circlearrowleft_{\mathcal{I}} (\otimes(\mathsf{iut}_1, \mathsf{iut}_2))$ *can do the trace* $\langle i_1|o_3, i_2|o_5\rangle$*. Thus,* $o_5 \in \mathit{Out}(\mathsf{iut} \text{ after } (\langle i_1|o_3\rangle, i_2))$ *whereas the global specification* $\mathsf{spec} = \circlearrowleft_{\mathcal{I}} (\otimes(\mathsf{spec}_1, \mathsf{spec}_2))$ *can do the trace* $\langle i_1|o_3\rangle$ *in such a way* $o_5 \notin \mathit{Out}(\mathsf{spec} \text{ after } (\langle i_1|o_3\rangle, i_2))$*. Hence, we can see that* iut *does not conform to* spec *according to cioco.*

This counterexample shows that the feedback operators may give rise to a global implementation that does not conform to its global specification, even if the local implementations conform to their local specifications. This is because the conformance relation *cioco* does not put any constraint on the traces that are not specified in the specification. It allows implementations to do what they want with the unspecified states. Observe that if the specification specifies for any input what the allowed outputs are, then we do not have this problem. Hence, to cope with this problem, we assume that specifications are input-enabled as in [30]. That is to say, all states of a specification spec accept all input actions of spec, and for each state s of spec and each input the function α is defined (α is a total function). Then, we have the following theorem for the compositionality for our feedback operators:

Theorm 1.2 *Let* $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$ *be a signature. Let* $\mathcal{I} = (f, \pi_i, \pi_o)$ *be a relaxed feedback interface. Let* $\mathcal{C}_j = (S_j, \alpha_j) \in \mathbf{Comp}(H)$ *such that each* $\mathcal{C}_j$ *are input-enabled for every* $j = 1, 2$*. Then, we have:*

$$\mathcal{C}_1 \; \textit{cioco} \; \mathcal{C}_2 \implies \hookleftarrow_{\mathcal{I}}(\mathcal{C}_1) \; \textit{cioco} \; \hookleftarrow_{\mathcal{I}}(\mathcal{C}_2) \tag{VIII.1}$$

$$\mathcal{C}_1 \text{ cioco } \mathcal{C}_2 \implies \circlearrowleft_{\mathcal{I}}(\mathcal{C}_1) \text{ cioco } \circlearrowleft_{\mathcal{I}}(\mathcal{C}_2) \tag{VIII.2}$$

Proof *We first need to prove the following lemma:*

Lemma 1.1 *Consider two components $\mathcal{C}_1$ and $\mathcal{C}_2$, then we have:*

1. *$Trace(\mathcal{C}_1) \subseteq Trace(\mathcal{C}_2)$ implies $(\mathcal{C}_1 \text{ cioco } \mathcal{C}_2)$*
2. *If $\mathcal{C}_2$ is input-enabled, then $(\mathcal{C}_1 \text{ cioco } \mathcal{C}_2)$ implies $Trace(\mathcal{C}_1) \subseteq Trace(\mathcal{C}_2)$.*

Proof

1. *Let $tr = \langle i_1|o_1, \ldots, i_n|o_n\rangle$ be a finite trace of $\mathcal{C}_2$, i an input of $\mathcal{C}_2$ and $o \in Out(\mathcal{C}_1 \text{ after } (tr, i))$ and let us prove that $o \in Out(\mathcal{C}_2 \text{ after } (tr, i))$.*
   $o \in Out(\mathcal{C}_1 \text{ after } (tr, i))$ implies $tr' = tr.\langle i|o\rangle = \langle i_1|o_1, i_2|o_2, \ldots, i_n|o_n, i|o\rangle \in Trace(\mathcal{C}_1)$. Since $Trace(\mathcal{C}_1) \subseteq Trace(\mathcal{C}_2)$, then $tr' \in Trace(\mathcal{C}_2)$. Thus, $o \in Out(\mathcal{C}_2 \text{ after } (tr, i))$, and consequently,

   $$Out(\mathcal{C}_1 \text{ after } (tr, i)) \subseteq Out(\mathcal{C}_2 \text{ after } (tr, i))$$

   The result then follows from the definition of cioco.

2. *By induction on the structure of a trace tr of $\mathcal{C}_1$. Let $tr = \langle i_1|o_1, \ldots, i_n|o_n\rangle \in Trace(\mathcal{C}_1)$.*

   - ***Basic Step:*** *$tr = \langle\rangle$ is empty trace.*
     $tr = \langle\rangle \in Trace(\mathcal{C}_2)$ trivially holds.
   - ***Induction Step:*** *Let us write tr as concatenation of two finite traces as follows:*

     $$tr = \langle i_1|o_1, i_2|o_2, \ldots, i_{n-1}|o_{n-1}\rangle \cdot \langle i_n|o_n\rangle$$

     $tr \in Trace(\mathcal{C}_1)$ implies $o_n \in Out(\mathcal{C}_1 \text{ after } (\langle i_1|o_1, \ldots, i_{n-1}|o_{n-1}\rangle, i_n))$. Since $\mathcal{C}_2$ is input-enabled, i_n is inevitably an input of $\mathcal{C}_2$ at any state s. By induction hypothesis, we have

     $$\langle i_1|o_1, \ldots, i_{n-1}|o_{n-1}\rangle \in Trace(\mathcal{C}_2) \text{ and } o_n \in Out(\mathcal{C}_1 \text{ after } (\langle i_1|o_1, \ldots, i_{n-1}|o_{n-1}\rangle, i_n))$$

     then $o_n \in Out(\mathcal{C}_2 \text{ after } (\langle i_1|o_1, \ldots, i_{n-1}|o_{n-1}\rangle, i_n))$ because $\mathcal{C}_1$ cioco $\mathcal{C}_2$.
     Thus $\langle i_1|o_1, \ldots, i_{n-1}|o_{n-1}, i_n|o_n\rangle \in Trace(\mathcal{C}_2)$. Consequently, $Trace(\mathcal{C}_1) \subseteq Trace(\mathcal{C}_2)$.

End

*Let us now **prove the first point of Theorem 1.2**. According to Lemma 1.1, we have to prove:*

$$Trace(\mathcal{C}_1) \subseteq Trace(\mathcal{C}_2) \implies Trace(\hookleftarrow_{\mathcal{I}}(\mathcal{C}_1)) \subseteq Trace(\hookleftarrow_{\mathcal{I}}(\mathcal{C}_2))$$

For this, let us use the proof by induction on the length of a finite trace tr of $Trace(\hookleftarrow_{\mathcal{I}}(\mathcal{C}_1))$. Let $tr = \langle i_0|o_0, \ldots, i_n|o_n\rangle$ be a finite trace of $\hookleftarrow_{\mathcal{I}}(\mathcal{C}_1)$.

- ***Basic Step:*** *$tr = \langle\rangle$ is empty trace.*
  $tr = \langle\rangle \in Trace(\hookleftarrow_{\mathcal{I}}(\mathcal{C}_2))$ trivially holds.

– ***Induction Step:*** *Let us write tr as concatenation of two finite traces as follows:*

$$tr = \langle i_0|o_0, i_1|o_1, \ldots, i_{n-1}|o_{n-1}\rangle \cdot \langle i_n|o_n\rangle$$

$tr \in Trace(\hookleftarrow_{\mathcal{I}}(\mathcal{C}_1))$ implies, according to the relaxed feedback definition (see Chapter V, Definition 1.3), that there exists an input sequence x and a couple $(\bar{x}, y_{\bar{x}})$ inductively defined from a finite sequence of states $(s_0, s_1, \ldots, s_n)$ of S_1 as follows:
– $\bar{x}(0) = x(0)$ and $y_{\bar{x}}(0) \in \eta'_{\mathsf{Out}\times S_1}(\alpha_1(s_0)(x(0)))_{|_1}$

– $\forall j, 0 < j \leq n,\ \bar{x}(j) = f(x(j), y_{\bar{x}}(j-1)),\ y_{\bar{x}}(j) \in \eta'_{\mathsf{Out}\times S_1}(\alpha_1(s_j)(\bar{x}(j)))_{|_1}$ and $s_j \in \eta'_{\mathsf{Out}_1\times S_1}(\alpha_1(s_{j-1})(\bar{x}(j-1)))_{|_2}$
and $\forall j, 0 \leq j \leq n,\ \pi_i(\bar{x}(j)) = i_j$ and $\pi_o(y_{\bar{x}}(j)) = o_j$.

By induction hypothesis,

$$\langle i_0|o_0, \ldots, i_{n-1}|o_{n-1}\rangle \in Trace(\hookleftarrow_{\mathcal{I}}(\mathcal{C}_2)) \text{ because } \langle i_0|o_0, \ldots, i_{n-1}|o_{n-1}\rangle \in Trace(\hookleftarrow_{\mathcal{I}}(\mathcal{C}_1))$$

Then, similarly to the above, there exists an input sequence x' and a couple $(\bar{x'}, y_{\bar{x'}})$ inductively defined from a finite sequence of states $(s'_0, s'_1, \ldots, s'_n)$ of S_2 as follows:
– $\bar{x'}(0) = x'(0)$ and $y_{\bar{x'}}(0) \in \eta'_{\mathsf{Out}\times S_2}(\alpha_2(s'_0)(x'(0)))_{|_1}$

– $\forall j, 0 < j \leq n-1,\ \bar{x'}(j) = f(x'(j), y_{\bar{x'}}(j-1)),\ y_{\bar{x'}}(j) \in \eta'_{\mathsf{Out}\times S_2}(\alpha_2(s'_j)(\bar{x'}(j)))_{|_1}$ and $s'_j \in \eta'_{\mathsf{Out}\times S_2}(\alpha_2(s'_{j-1})(\bar{x'}(j-1)))_{|_2}$
and $\forall j, 0 \leq j \leq n-1,\ \pi_i(\bar{x'}(j)) = i_j$ and $\pi_o(y_{\bar{x'}}(j)) = o_j$.

Since $Trace(\mathcal{C}_1) \subseteq Trace(\mathcal{C}_2)$, $\langle i_0, \ldots, i_n\rangle$ is inevitably an input sequence of $\mathcal{C}_2$.
Then, $\eta'_{\mathsf{Out}\times S_2}(\alpha_2(s'_n)(f(i_n, y_{\bar{x'}}(n-1))))_{|_1}$ is well defined.
Now, we know that

$$\eta'_{\mathsf{Out}\times S_1}(\alpha_1(s_n)(f(x(n), y_{\bar{x}}(n-1))))_{|_1} \subseteq \eta'_{\mathsf{Out}\times S_2}(\alpha_2(s'_n)(f(i_n, y_{\bar{x'}}(n-1)))_{|_1}$$

This is because $Trace(\mathcal{C}_1) \subseteq Trace(\mathcal{C}_2)$. This implies that

$$y_{\bar{x}}(n) \in \eta'_{\mathsf{Out}\times S_2}(\alpha_2(s'_n)(f(i_n, y_{\bar{x'}}(n-1))))_{|_1}$$

Hence according to the relaxed feedback definition, $\langle i_1|o_1, \ldots, i_{n-1}|o_{n-1}, i_n|o_n\rangle \in Trace(\hookleftarrow_{\mathcal{I}}(\mathcal{C}_2))$. Consequently, $Trace(\hookleftarrow_{\mathcal{I}}(\mathcal{C}_1)) \subseteq Trace(\hookleftarrow_{\mathcal{I}}(\mathcal{C}_2))$.

Let us now ***prove the second point of Theorem 1.2****. According to Lemma 1.1, we have to prove:*

$$Trace(\mathcal{C}_1) \subseteq Trace(\mathcal{C}_2) \Longrightarrow Trace(\circlearrowleft_{\mathcal{I}}(\mathcal{C}_1)) \subseteq Trace(\circlearrowleft_{\mathcal{I}}(\mathcal{C}_2))$$

For this, let us use the proof by induction on the length of a finite trace tr of $Trace(\circlearrowleft_{\mathcal{I}}(\mathcal{C}_1))$.
Let $tr = \langle i_0|o_0, \ldots, i_n|o_n\rangle$ be a finite trace of $\circlearrowleft_{\mathcal{I}}(\mathcal{C}_1)$.

– ***Basic Step:*** *$tr = \langle\rangle$ is empty trace.*
$tr = \langle\rangle \in Trace(\circlearrowleft_{\mathcal{I}}(\mathcal{C}_2))$ trivially holds.

– ***Induction Step:*** *Let us write tr as concatenation of two finite traces as follows:*

$$tr = \langle i_0|o_0, i_1|o_1, \ldots, i_{n-1}|o_{n-1}\rangle \cdot \langle i_n|o_n\rangle$$

$tr \in Trace(\circlearrowleft_{\mathcal{I}}(\mathcal{C}_1))$ implies, according to the synchronous feedback definition (see Chapter V, Definition 1.5), that there exists an input sequence x, an output sequence y and a finite sequence of states $(s_0, s_1, \ldots, s_n)$ of S_1 such that:

$$\forall j, 0 \leq j \leq n,\ (y(j), s_{j+1}) \in \eta'_{\mathsf{Out}_1\times S_1}(\alpha_1(s_j)(f(x(j), y(j))))$$

and $\forall j, 0 \leq j \leq n,\ \pi_i(x(j)) = i_j$ *and* $\pi_o(y(j)) = o_j$.

By induction hypothesis,

$$\langle i_0|o_0, \ldots, i_{n-1}|o_{n-1}\rangle \in Trace(\circlearrowleft_{\mathcal{I}}(\mathcal{C}_2)) \text{ because } \langle i_0|o_0, \ldots, i_{n-1}|o_{n-1}\rangle \in Trace(\circlearrowleft_{\mathcal{I}}(\mathcal{C}_1))$$

Then, similarly to the above, there exists an input sequence x'*, an output sequence* y' *and a finite sequence of states* $(s'_0, s'_1, \ldots, s'_n)$ *of* S_2 *such that:*

$$\forall j, 0 \leq j \leq n,\ (y'(j), s'_{j+1}) \in \eta'_{\mathsf{Out}_2 \times S_2}(\alpha_2(s'_j)(f(x(j)', y(j)')))$$

and $\forall j, 0 \leq j \leq n-1,\ \pi_i(x(j)') = i_j$ *and* $\pi_o(y(j)') = o_j$.

Since $Trace(\mathcal{C}_1) \subseteq Trace(\mathcal{C}_2)$, $\langle i_0, \ldots, i_n\rangle$ *is inevitably an input sequence of* $\mathcal{C}_2$*. Then,*

$$\eta'_{\mathsf{Out}_2 \times S_2}(\alpha_2(s'_n)(f(i_n, y(n)))_{|_1} \text{ is well defined}$$

Now, since $Trace(\mathcal{C}_1) \subseteq Trace(\mathcal{C}_2)$*, we have that:*

$$\eta'_{\mathsf{Out}_1 \times S_1}(\alpha_1(s_n)(f(x(n), y(n))))_{|_1} \subseteq \eta'_{\mathsf{Out}_2 \times S_2}(\alpha_2(s'_n)(f(i_n, y(n)'))_{|_1}$$

Thus, $y(n) \in \eta'_{\mathsf{Out}_2 \times S_2}(\alpha_2(s'_n)(f(i_n, y(n))))_{|_1}$*. Hence according to synchronous feedback definition,* $\langle i_1|o_1, \ldots, i_n|o_n\rangle \in Trace(\circlearrowleft_{\mathcal{I}}(\mathcal{C}_2))$*. Consequently,* $Trace(\circlearrowleft_{\mathcal{I}}(\mathcal{C}_1)) \subseteq Trace(\circlearrowleft_{\mathcal{I}}(\mathcal{C}_2))$*.*

End

1.4 Compositionality for complex operator

Theorem 1.1 and Theorem 1.2 obviously lead to the following theorem:

Theorm 1.3 *Let* op *be a complex operator of arity* n*. Let* $\mathcal{C}_1, \ldots, \mathcal{C}_n, \mathcal{C}'_1, \ldots, \mathcal{C}'_n$ *be components such that:*

$$\forall i, 1 \leq i \leq n,\ \mathcal{C}_i \text{ cioco } \mathcal{C}'_i, \text{ then one has } op(\mathcal{C}_1, \ldots, \mathcal{C}_n) \text{ cioco } op(\mathcal{C}'_1, \ldots, \mathcal{C}'_n).$$

Proof *By induction on the the structure of the complex operator* op *d'arity* n*.*

- ***Basic Step:***
  op *is of the form* _*. The property mentioned in Theorem 1.3 trivially holds.*
- ***Induction Step:*** *we distinguish the following cases:*
  1. $op = \otimes(op_1, op_2)$ *with arity of* op_1 *is* n_1*, arity of* op_2 *is* n_2 *and* $n_1 + n_2 = n$

     by induction hypothesis and the definition of both op_1 *and* op_2*, we have:*

     (1) $op_1(\mathcal{C}_1, \ldots, \mathcal{C}_{n_1})$ *cioco* $op_1(\mathcal{C}'_1, \ldots, \mathcal{C}'_{n_1})$
     and both $op_1(\mathcal{C}_1, \ldots, \mathcal{C}_{n_1})$ *and* $op_1(\mathcal{C}'_1, \ldots, \mathcal{C}'_{n_1})$ *are components;*

     (2) $op_2(\mathcal{C}_{n_1+1}, \ldots, \mathcal{C}_n)$ *cioco* $op_2(\mathcal{C}'_{n_1+1}, \ldots, \mathcal{C}'_n)$
     and both $op_2(\mathcal{C}_{n_1+1}, \ldots, \mathcal{C}_n)$ *and* $op_2(\mathcal{C}'_{n_1+1}, \ldots, \mathcal{C}'_n)$ *are components.*

     Then, (1) + (2) + *Theorem 1.1 implies that*
     $op = \otimes(op_1(\mathcal{C}_1, \ldots, \mathcal{C}_{n_1}), op_2(\mathcal{C}_{n_1+1}, \ldots, \mathcal{C}_n))$
     cioco
     $op = \otimes(op_1(\mathcal{C}'_1, \ldots, \mathcal{C}'_{n_1}), op_2(\mathcal{C}'_{n_1+1}, \ldots, \mathcal{C}'_n))$

2. *$op = \circlearrowleft_{\mathcal{I}} (op')$ by induction hypothesis and the definition of op', we have:*

   $(*)$ $op'(\mathcal{C}_1, \ldots, \mathcal{C}_n)$ cioco $op'(\mathcal{C}'_1, \ldots, \mathcal{C}'_n)$ and both $op'(\mathcal{C}_1, \ldots, \mathcal{C}_n)$ and $op'(\mathcal{C}'_1, \ldots, \mathcal{C}'_n)$ are components;

   Then, $(*)$ + Theorem 1.2 implies that

$$op = \circlearrowleft_{\mathcal{I}}(op'(\mathcal{C}_1, \ldots, \mathcal{C}_n)) \text{ cioco } op = \circlearrowleft_{\mathcal{I}}(op'(\mathcal{C}'_1, \ldots, \mathcal{C}'_n))$$

3. *$op = \leftrightarrow_{\mathcal{I}} (op')$ by induction hypothesis and the definition of op', we have:*

   $(*)$ $op'(\mathcal{C}_1, \ldots, \mathcal{C}_n)$ cioco $op'(\mathcal{C}'_1, \ldots, \mathcal{C}'_n)$ and both $op'(\mathcal{C}_1, \ldots, \mathcal{C}_n)$ and $op'(\mathcal{C}'_1, \ldots, \mathcal{C}'_n)$ are components;

   Then, $(*)$ + Theorem 1.2 implies that

$$op = \leftrightarrow_{\mathcal{I}}(op'(\mathcal{C}_1, \ldots, \mathcal{C}_n)) \text{ cioco } op = \leftrightarrow_{\mathcal{I}}(op'(\mathcal{C}'_1, \ldots, \mathcal{C}'_n))$$

End

By Theorem 1.3, we directly have that sequential, double sequential, synchronous parallel and concurrent compositions as well as synchronous product are compositional for *cioco*.

Example 1.3 *(**Continue Example 1.1**) As an example of compositional testing, we have considered in Example 1.1 the money $\mathcal{M}$ and drink $\mathcal{D}$ components, where we have also shown that* $\mathsf{iut}_{\mathcal{M}}$ *cioco* $\mathsf{spec}_{\mathcal{M}}$ *and* $\mathsf{iut}_{\mathcal{D}}$ *cioco* $\mathsf{spec}_{\mathcal{D}}$*. Here, the question is*

$$\text{if } \odot(\mathsf{iut}_{\mathcal{M}}, \mathsf{iut}_{\mathcal{D}}) \text{ cioco } \odot(\mathsf{spec}_{\mathcal{M}}, \mathsf{spec}_{\mathcal{D}})?$$

Our first attempt to answer this question is to check if the assumptions imposed in Theorem 1.3 are satisfied. Observe that neither $\mathsf{spec}_{\mathcal{M}}$ *nor* $\mathsf{spec}_{\mathcal{D}}$ *are input-enabled. Hence, Theorem 1.3 fails to hold the compositinality of cioco for the components $\mathcal{M}$ and $\mathcal{D}$. However, it is easy to see that the global implementation* $\mathsf{iut} = \odot(\mathsf{iut}_{\mathcal{M}}, \mathsf{iut}_{\mathcal{D}})$ *can do the trace*

$$tr = \langle \text{coinC}|\text{preparing}, \mathsf{abs}|\text{coffee}, \text{coinC}|\text{preparing}, \mathsf{abs}|\text{refund} \rangle$$

Thus,

$$\text{refund} \in Out(\mathsf{iut} \text{ after } (\langle \text{coinC}|\text{preparing}, \mathsf{abs}|\text{coffee}, \text{coinC}|\text{preparing} \rangle, \mathsf{abs}))$$

whereas the global specification $\mathsf{spec} = \odot(\mathsf{spec}_{\mathcal{M}}, \mathsf{spec}_{\mathcal{D}})$ *can also do the trace*

$$\langle \text{coinC}|\text{preparing}, \mathsf{abs}|\text{coffee}, \text{coinC}|\text{preparing} \rangle$$

in such a way

$$\text{refund} \notin Out(\mathsf{spec} \text{ after } (\langle \text{coinC}|\text{preparing}, \mathsf{abs}|\text{coffee}, \text{coinC}|\text{preparing} \rangle, \mathsf{abs}))$$

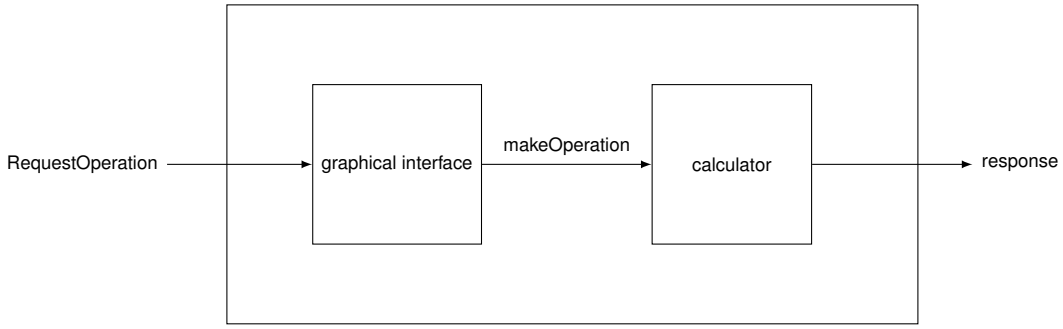
Hence, we can see that iut *does not conform to* spec *according to cioco.*

2 Test purposes for sub-systems

In this section, we propose an approach to testing components that are typically involved in the whole system by defining dedicated test purposes for them, from the global behaviour of the whole system. Such test purposes are given in an accurate way by defining a projection

mechanism taking a global behaviour p of the whole system and keeping only the part of p being activated in the sub-system that we want to test. Thus, our method for generating test purposes from the global system specification helps to generate good relevant unit test cases to test individual components.

The objective of the approach we propose here is to make component testing more efficient by focusing on the way components are used in global systems. Indeed, as the number of test case combinations is often huge, testing components in isolation would cause test cases important for the global system to be overlooked. As an illustration, let us consider an over simplified system that computes grade averages.



A typical design view of this system consists of two sub-systems:

- an "user interface" $\mathcal{I}$ that helps the user to make various operations on grades;
- a "calculator" $\mathcal{C}$ that receives operation commands from the user, performs the requested operation, and reports back to the user.

Now, testing the component $\mathcal{C}$ separately may lead to the consideration of test cases involving arithmetic operations which are irrelevant to computing student grade averages as subtraction, multiplication, square root, etc. This may cause test cases of interest to the system to be missed, i.e. test cases only bringing into play addition and division for grades ranging from 0 to 20. Then, by making a projection of this behaviour on calculator component $\mathcal{C}$, we intend to generate a test purpose that guides the test derivation process of $\mathcal{C}$ by only testing operations needed to compute grade averages.

We show here how a trace of a system can be projected on its components. Such projected traces will be the cornerstone to define test purposes dedicated to test components separately. Hence, those test purposes will capture behaviours of sub-systems that typically occur in the whole system. This will be done by combining projection mechanisms and execution mechanisms to generate system computation trees.

2.1 Sub-systems and projection

We introduce the definition of a sub-system involved in a given system. This intuitively allows us to characterize the set of all basic sub-systems from which the global system can be built.

Definition 2.1 *(**Sub-systems**) Let $\mathcal{S} = op(\mathcal{C}_1, \ldots, \mathcal{C}_n)$ be a system over a signature H. The **set of sub-systems of** $\mathcal{S}$, noted $\mathbb{S}ub(\mathcal{S})$, is inductively defined on the structure of op as follows:*

- *if $op = \_$, then $\mathbb{S}ub(\mathcal{S}) = \{\mathcal{S}\}$;*
- *if $op = op_1 \otimes op_2$ with op_1 and op_2 of arity n_1 and n_2 respectively (i.e. $n = n_1 + n_2$), then $\mathbb{S}ub(\mathcal{S}) = \{\mathcal{S}\} \cup \mathbb{S}ub(op_1(\mathcal{C}_1, \ldots, \mathcal{C}_{n_1})) \cup \mathbb{S}ub(op_2(\mathcal{C}_{n_1+1}, \ldots, \mathcal{C}_n))$;*

- *if* $op = \circlearrowleft_{\mathcal{I}}(op')$, *then* $\mathbb{S}ub(\mathcal{S}) = \{\mathcal{S}\} \cup \mathbb{S}ub(op'(\mathcal{C}_1, \ldots, \mathcal{C}_n))$;
- *if* $op = \hookleftarrow_{\mathcal{I}}(op')$, *then* $\mathbb{S}ub(\mathcal{S}) = \{\mathcal{S}\} \cup \mathbb{S}ub(op'(\mathcal{C}_1, \ldots, \mathcal{C}_n))$.

For any finite trace tr of a finite computation tree of $\mathcal{S}$ and a sub-system sys of $\mathcal{S}$, we characterize the set of finite traces $tr_{\downarrow_{sys}}$ of sys involved in tr.

Definition 2.2 ***(Projection of a finite trace)*** *Let* $\mathcal{S} = op(\mathcal{C}_1, \ldots, \mathcal{C}_n)$ *be a system over a signature* $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$. *Let* $sub \in \mathbb{S}ub(\mathcal{S})$ *be a sub-system of* $\mathcal{S}$ *over* $H' = T(\mathsf{Out}' \times \_)^{\mathsf{In}'}$. *Let* $tr = \langle i_1|o_1, i_2|o_2, \ldots, i_m|o_m\rangle \in Trace(\mathcal{S})$. *The* ***projection of*** tr ***on*** sub, *denoted by* $tr_{\downarrow_{sub}}$, *is the subset of* $Trace(sub)$ *inductively defined as follows:*

- *if* $op = \_$, *then* $tr_{\downarrow_{sub}} = \{tr\}$;
- *if* $op = op_1 \otimes op_2$ *with* op_1 *and* op_2 *of arity* n_1 *and* n_2 *respectively (i.e.* $n = n_1 + n_2$*), then*[5]*:*

$$tr_{\downarrow_{sub}} = \begin{cases} \text{is the projection of } \langle i_{1_{|_1}}|o_{1_{|_1}}, i_{2_{|_1}}|o_{2_{|_1}}, \ldots, i_{m_{|_1}}|o_{m_{|_1}}\rangle \\ \text{on } sub \text{ if } sub \in \mathbb{S}ub(op(\mathcal{C}_1, \ldots, \mathcal{C}_{n_1})) \\ \\ \text{is the projection of } \langle i_{1_{|_2}}|o_{1_{|_2}}, i_{2_{|_2}}|o_{2_{|_2}}, \ldots, i_{m_{|_2}}|o_{m_{|_2}}\rangle \\ \text{on } sub \text{ otherwise} \end{cases}$$

- *if* $op = \circlearrowleft_{\mathcal{I}}(op')$ *with* $\mathcal{I} = (f, \pi_i, \pi_o)$, *then* $tr_{\downarrow_{sub}} = \bigcup_{tr' \in tr_{\downarrow_{\mathcal{S}'}}} tr'_{\downarrow_{sub}}$ *where* $\mathcal{S}' = op'(\mathcal{C}_1, \ldots, \mathcal{C}_n)$ *and*

$$tr_{\downarrow_{\mathcal{S}'}} = \Big\{ \langle i'_1|o'_1, \ldots, i'_m|o'_m\rangle \quad | \quad \forall j, 1 \leq j \leq m, \\ \exists s_j \in S', o'_j \in \eta'_{\mathsf{Out}' \times S'}(\alpha_{\mathcal{S}'}(s_j)(f(i'_j, o'_j)))_{|_1} \\ i_j = \pi_i(i'_j) \text{ and } o_j = \pi_o(o'_j) \Big\}$$

2.2 System-based test purposes

In this subsection, we adapt the notion of test purpose presented in Chapter VII, Section 3 to test, from a global behaviour of a system, the behaviour of its involved sub-systems and then we guide the component testing intelligently by taking into account the way components are used in systems. Thus, taking a behaviour p of a system $\mathcal{S}$, we intend to define test purposes that are able to test the behaviour p_i of each sub-system $\mathcal{S}_i \in \mathbb{S}ub(\mathcal{S})$. We identify therefore for each sub-system all its finite paths that are involved in constructing the whole behaviour of $\mathcal{S}$.

We first define the *finite computation tree* of a subsystem sub of a global system $\mathcal{S}$ which captures all its finite traces:

Definition 2.3 ***(Finite computation tree)*** *Let* $\mathcal{S}$ *be a system over* $T(\mathsf{Out} \times \_)^{\mathsf{In}}$. *Let* $sub \in \mathbb{S}ub(\mathcal{S})$ *be a subsystem of* $\mathcal{S}$ *over* $T(\mathsf{Out}' \times \_)^{\mathsf{In}'}$. *The* ***finite computation tree of*** sub ***generated by*** $\mathcal{S}$ ***of depth less than*** n, *noted* $FCT(sub, n)$ *is the coalgebra* $(S_{FCT}, s^0_{FCT}, \alpha_{FCT})$ *defined by:*

- $s^0_{FCT} = \langle\rangle$
- S_{FCT} *is the whole set of finite traces defined as follows:*
  - $s^0 = \{\langle\rangle\}$
  - $\forall j, 1 \leq j \leq n, s^j = \{tr'.\langle i|o\rangle \mid \exists tr' \in s^{j-1}, \exists i \in \mathsf{In}', \exists o \in \mathsf{Out}', \exists tr \in Trace(\mathcal{S})$ *such that* $tr'.\langle i|o\rangle \in tr_{\downarrow_{sub}}\}$

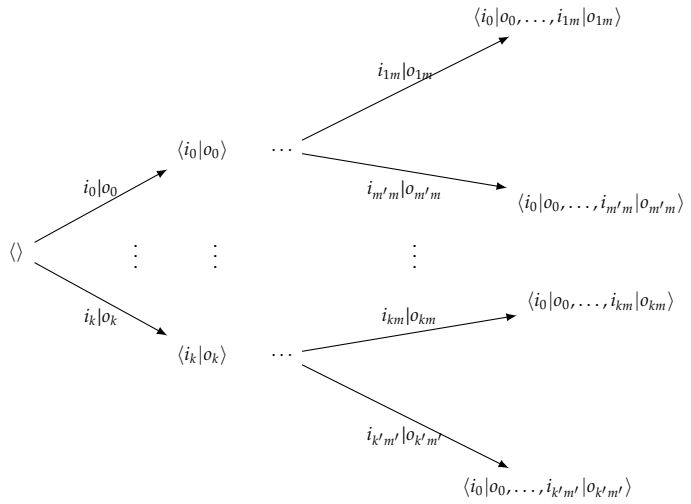
5. $a_{|_i}$ is the projection of the n-tuple a on i^{th} argument.

Hence, $S_{FCT} = \bigcup_{0 \leq j \leq n} s^i$

– $\alpha_{FCT} : S_{FCT} \times \mathsf{In}' \longrightarrow T(\mathsf{Out}' \times S_{FCT})$ *is the mapping which for every* $\langle i_0|o_0, \ldots, i_m|o_m\rangle \in S_{FCT}$ *and every input* $i \in \mathsf{In}'$ *associates* $\eta'^{-1}_{\mathsf{Out}' \times S_{FCT}}(\Pi)$ *where* Π *is the set:*

$$\Pi = \{(o, \langle i_0|o_0, \ldots, i_m|o_m, i|o\rangle) \mid \exists o \in \mathsf{Out}', \exists tr \in Trace(\mathcal{S}) \text{ such that } \langle i_0|o_0, \ldots, i_m|o_m, i|o\rangle \in tr_{\downarrow_{sub}}\}$$

In this definition, S_{FCT} is the set of the nodes of the tree. s^0_{FCT} is the root of the tree. Each node is represented by the unique finite trace $\langle i_0|o_0, \ldots, i_m|o_m\rangle$ ($m \leq n$). α_{FCT} gives, for each node p and for each input i, the set of nodes Π that can be reached from p when the input i is submitted to the component.



Definition 2.4 ***(Test Purpose)*** *Let* $\mathcal{S}$ *be a system over a signature* $H = T(\mathsf{Out} \times \_)^{\mathsf{In}}$. *Let* $sub \in Sub(\mathcal{S})$ *be a sub-system of* $\mathcal{S}$ *over* $H' = T(\mathsf{Out}' \times \_)^{\mathsf{In}'}$ *and* $FCT(sub, n) = (S, s_0, \alpha)$ *its finite computation tree generated by* $\mathcal{S}$. *Let* tr *be a finite trace of* $\mathcal{S}$ *such that its length is less than* n. *Let* $tr_{\downarrow_{sub}}$ *be the projection of* tr *on* sub. *A* test purpose TP *for* tr *and* sub *is a mapping* $\mathsf{TP} : S_{FCT} \longrightarrow \{\mathsf{accept}, \mathsf{skip}, \odot\}$ *such that:*

– *for every node* $p = \langle i_0|o_0, \ldots, i_m|o_m\rangle \in tr_{\downarrow_{sub}}$, $\mathsf{TP}(p) = \mathsf{accept}$;

– *if* $\mathsf{TP}(\langle i_0|o_0, \ldots, i_m|o_m\rangle) = \mathsf{accept}$, *then:*
$\forall j, 0 \leq j \leq m, \mathsf{TP}(\langle i_0|o_0, \ldots, i_{j-1}|o_{j-1}\rangle) = \mathsf{skip}$

– $\mathsf{TP}(\langle\rangle) = \mathsf{skip}$

– *if* $\mathsf{TP}(\langle i_0|o_0, \ldots, i_k|o_k\rangle) = \odot$, *then:*
$\mathsf{TP}(i_0|o_0, \ldots, i_k|o_k, i'_{k+1}|o'_{k+1}, \ldots, i'_{k'}|o'_{k'}\rangle) = \odot$
for all $k < k' \leq n$ *and for all* $(i'_l)_{k \leq l < n} \in \mathsf{In}'$ *and* $(o'_l)_{k \leq l < n} \in \mathsf{Out}'$.

In order to build a test purpose for a finite behaviour projection $tr_{\downarrow_{sub}}$ on a sub-system sub, we identify all finite paths of its finite computation tree FCT whose traces embody $tr_{\downarrow_{sub}}$ and we tag them with accept. We then tag every node which represents a prefix of an accepted behaviour with skip. The other nodes, which lead to behaviours that we do not want to test, are tagged with $\odot$.

Example 2.1 *Let us consider a finite trace*

$$tr = \langle \text{coinC}|\text{preparing}, \text{abs}|\text{coffee}, \text{coinC}|\text{preparing}, \text{abs}|\text{coffee}, \text{coinC}|\text{preparing}, \text{abs}|\text{coffee}\rangle$$

of the coffee machine obtained by a synchronous parallel composition of the money component $\mathcal{M}$ and the drink component $\mathcal{D}$ whose specifications are illustrated in Figure VIII.2a and Figure VIII.2b respectively, and then, from it, build a test purpose dedicated to test the behaviour of the drink component $\mathcal{D}$.

First, we build the finite computation $FCT(\mathcal{D}, 6)$ tree of $\mathcal{D}$ generated by $\odot(\text{spec}_{\mathcal{M}}, \text{spec}_{\mathcal{D}})$ that we represent in Figure VIII.4. Second, we compute the projection of tr on $\mathcal{D}$

$$tr_{\downarrow_{\mathcal{D}}} = \{\langle \text{makeC}|\text{preparing}, \text{abs}|\text{coffee}, \text{makeC}|\text{preparing}, \text{abs}|\text{coffee}, \text{makeC}|\text{preparing}, \text{abs}|\text{coffee}\rangle\}$$

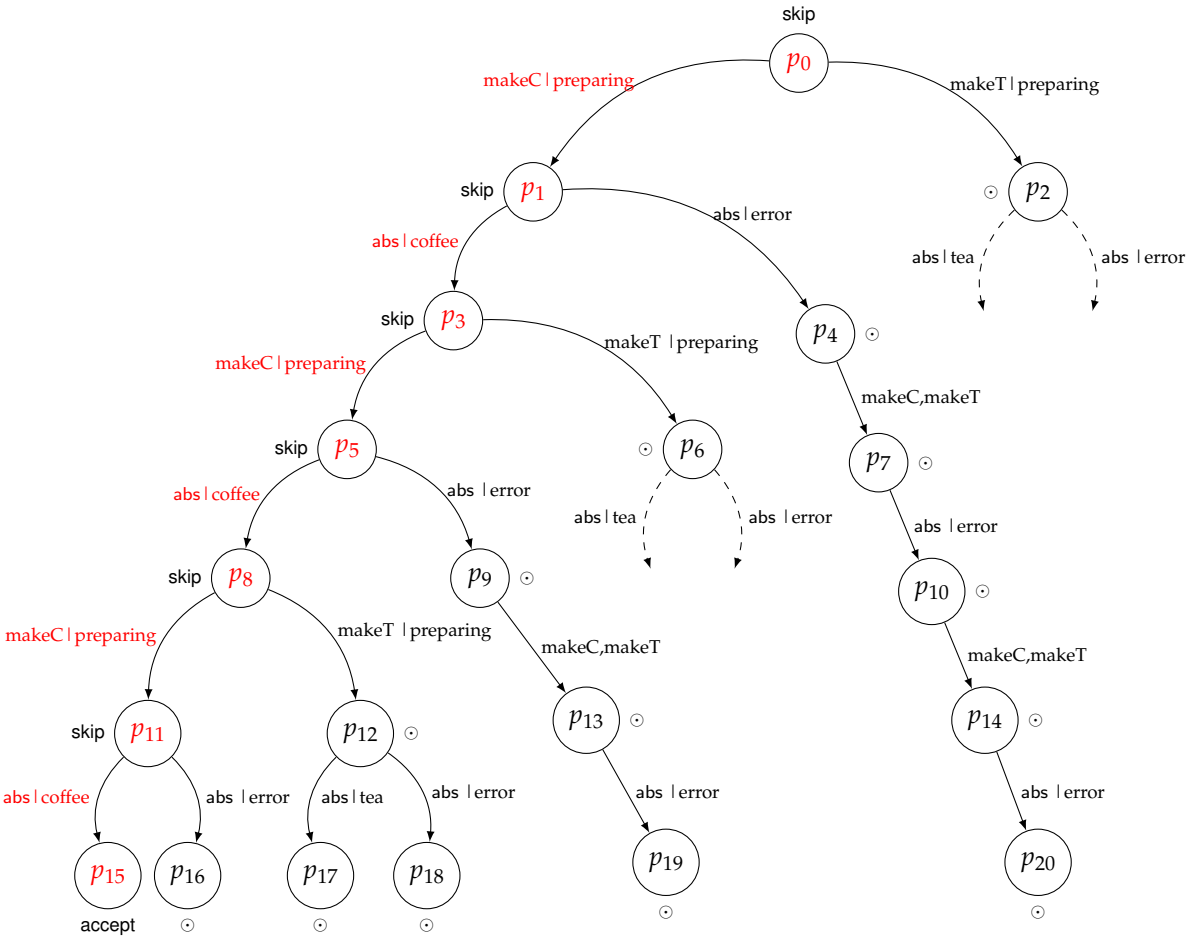
This corresponds intuitively to three requests for making coffee to the component drink $\mathcal{D}$ without any error occurring. A test purpose TP *for this behaviour would then concentrate on the delivering of coffee and ignore the behaviours of $\mathcal{D}$ related both to errors when $\mathcal{D}$ fails and to tea delivering. Hence, each state of $FCT(\mathcal{D}, 6)$ reachable after $tr_{\downarrow_{\mathcal{D}}}$ is tagged with* **accept**. *Then, p_{15} is only tagged with* **accept** *because it is the unique leaf which corresponds to an expected behaviour. All nodes leading from the root p_0 to this node p_{15} are tagged with* **skip** *(i.e p_0, p_1, p_3, p_5, p_8 and p_{11}). Finally, all other states are tagged with $\odot$.*
Thus, testing of $\mathcal{D}$ is re-enforced as far as the coffee delivering is concerned: only behaviours related to correct coffee delivering process are chosen and then sub-system behaviours that are not activated in the global system are not tested. This allows us to restrict the test domain to the one under consideration.

3 Related works

In this section, we present a brief overview of contributions which are technically close to our proposal, but which focus on various aspects of compositional testing, as well as on component-based ones. We discuss the differences between problematics addressed by those contributions and those addressed by our approach.

The component-based systems testing framework proposed in [30] is closer technically to our compositional testing approach. In this paper, the authors address compositionality for *ioco* conformance relation. Both specification and implementation component models are considered as *LTSs*. Parallel composition and hiding operators are used to combine *LTS* models. The parallel composition of two *LTSs* S_1 and S_2 consists in synchronizing their actions: when both S_1 and S_2 are ready to engage in the same action a, there is a transition in the composed *LTS* which carries the action a; when one of them is ready to engage in an action not shared with the other one, it may evolve independently and the reaction of the composed *LTS* consists only of the reaction of the *LTS* that reacts. The hiding operator consists in hiding the common or synchronized actions by replacing them by an internal action τ, and then restricting observability of internal actions. It has been proved that the conformance testing *ioco* is only compositional with respect to parallel composition and hiding when specifications and implementations are assumed input-enabled.

Among the works concerning compositional testing as it was defined in [30] and which were adopted in our framework, we can mention the work proposed by *Sampaio* in [136]. In this paper the authors extend the testing theory defined in the setting of *CSP* process algebra whose conformance relation *cspio* is an adapted version of *ioco* to *CSP* formalism [132], to be able to address compositional testing proposed by *Tretmans* in [30]. Indeed, it has been shown that *cspio* is compositional not only for parallel composition $\|$ and hiding operator $/\_$ but also for other *CSP*'s composition operators such that deterministic and nondeterministic choices, by assuming that input completeness of the specification is in the same alphabet of the implementation.



$p_0 = \langle\rangle$
$p_1 = \langle \text{makeC}|\text{preparing}\rangle$
$p_2 = \langle \text{makeT}|\text{preparing}\rangle$
$p_3 = \langle \text{makeC}|\text{preparing}, \text{abs}|\text{coffee}\rangle$
$p_4 = \langle \text{makeC}|\text{preparing}, \text{abs}|\text{error}\rangle$
$p_5 = \langle \text{makeC}|\text{preparing}, \text{abs}|\text{coffee}, \text{makeC}|\text{preparing}\rangle$
$p_6 = \langle \text{makeC}|\text{preparing}, \text{abs}|\text{coffee}, \text{makeT}|\text{preparing}\rangle$
$p_7 = \langle \text{makeC}|\text{preparing}, \text{abs}|\text{error}, \{\text{makeC}, \text{makeT}\}\rangle$
$p_8 = \langle \text{makeC}|\text{preparing}, \text{abs}|\text{coffee}, \text{makeC}|\text{preparing}, \text{abs}|\text{coffee}\rangle$
$p_9 = \langle \text{makeC}|\text{preparing}, \text{abs}|\text{error}, \text{makeC}|\text{preparing}, \text{abs}|\text{error}\rangle$
$p_{10} = \langle \text{makeC}|\text{preparing}, \text{abs}|\text{error}, \{\text{makeC}, \text{makeT}\}, \text{abs}|\text{error}\rangle$
$p_{11} = \langle \text{makeC}|\text{preparing}, \text{abs}|\text{coffee}, \text{makeC}|\text{preparing}, \text{abs}|\text{coffee}, \text{makeC}|\text{preparing}\rangle$
$p_{12} = \langle \text{makeC}|\text{preparing}, \text{abs}|\text{coffee}, \text{makeC}|\text{preparing}, \text{abs}|\text{coffee}, \text{makeT}|\text{preparing}\rangle$
$p_{13} = \langle \text{makeC}|\text{preparing}, \text{abs}|\text{coffee}, \text{makeC}|\text{preparing}, \text{abs}|\text{error}, \{\text{makeC}, \text{makeT}\}\rangle$
$p_{14} = \langle \text{makeC}|\text{preparing}, \text{abs}|\text{error}, \{\text{makeC}, \text{makeT}\}, \text{abs}|\text{error}, \{\text{makeC}, \text{makeT}\}\rangle$
$p_{15} = \langle \text{makeC}|\text{preparing}, \text{abs}|\text{coffee}, \text{makeC}|\text{preparing}, \text{abs}|\text{coffee}, \text{makeC}|\text{preparing}, \text{abs}|\text{coffee}\rangle$
$p_{16} = \langle \text{makeC}|\text{preparing}, \text{abs}|\text{coffee}, \text{makeC}|\text{preparing}, \text{abs}|\text{coffee}, \text{makeC}|\text{preparing}, \text{abs}|\text{error}\rangle$
$p_{17} = \langle \text{makeC}|\text{preparing}, \text{abs}|\text{coffee}, \text{makeC}|\text{preparing}, \text{abs}|\text{error}, \text{makeT}|\text{preparing}, \text{abs}|\text{tea}\rangle$
$p_{18} = \langle \text{makeC}|\text{preparing}, \text{abs}|\text{coffee}, \text{makeC}|\text{preparing}, \text{abs}|\text{coffee}, \text{makeT}|\text{preparing}, \text{abs}|\text{error}\rangle$
$p_{19} = \langle \text{makeC}|\text{preparing}, \text{abs}|\text{coffee}, \text{makeC}|\text{preparing}, \text{abs}|\text{error}, \{\text{makeC}, \text{makeT}\}, \text{abs}|\text{error}\rangle$
$p_{20} = \langle \text{makeC}|\text{preparing}, \text{abs}|\text{error}, \{\text{makeC}, \text{makeT}\}, \text{abs}|\text{error}, \{\text{makeC}, \text{makeT}\}, \text{abs}|\text{error}\rangle$

Figure VIII.4 – Test purpose of the drink component

In [48], the authors propose to test each component of a system in isolation by generating accurate test purposes for them from the global specification of the system and assuming that the specification of every component in the system is available. They use the input-output symbolic transition systems $(IOSTS)$ as the behavioural model of components and both synchronized product and hiding operator in order to compose components. Then, the authors propose to derive test purposes for a given component $\mathcal{C}$ of the system $\mathcal{S}$ from a global behaviour of $\mathcal{S}$. This is done by defining an adequate projection mechanism that allows them to project symbolic behaviour of $\mathcal{S}$ on its components. Those projected behaviours are then considered good behaviours to be tested on sub-systems. Thus, they are used to build test purposes.

In [86, 87], *Petrenko* and *al.* see the compositional testing problem differently from our approach and those presented in [30, 136]. They address the following question: "*how to design a component that when combined with a known part of the system, called the context, has to satisfy a given overall specification?*" To answer this question, the authors of [86, 87] specify the behaviour of components as finite state machines and the interactions between components by means of two operators: synchronous composition and parallel composition (or asynchronous composition). Then, they associate a class of languages to finite state machines that allow them to define equations over languages. Such a behaviour modeling is made to be the cornerstone of testing complex systems in context. In this setting, the above question is expressed formally by the following equation over *FSM* languages

$$(C \ \mathbf{op} \ X) \ \mathbf{rel} \ \mathsf{spec}$$

where C models the context, spec models the global specification, X is unknown, **op** stands for a composition operator and **rel** for a conformance relation. It has been proved that the largest solution of this language equation is given by the language $S = \overline{C} \ \mathbf{op} \ \overline{\mathsf{spec}}$ when **op** stands for both parallel composition or synchronous composition and **rel** stands for languages inclusion $\subseteq$. As previously mentioned, finite state machines and operators used to compose them can be defined in our framework. Then, to extend their results to our framework, it still need to define a class of languages accepted by *Barbosa*'s components. To do that, we can take advantage of the work done in [137] that generalize the classical notion of regular expression to coalgebras over polynomial functors.

In [138], the authors extend the so-called *assume-guarantee reasoning* [28] used in model checking areas as a means to cope with the state explosion problem of compositional testing. They then proposed to test each component of a system separately, while taking into account assumptions about the context of the component. They use the input-output labeled transition systems as behavioural models of components and the parallel composition $\|$ to compose components. The conformance relation used in this approach is the *ioco* relation. The underlying idea behind this approach is to check that, given a assumption A about the environment in which the components are supposed to operate, such that iut_2 *ioco* A and $(\mathsf{iut}_1 \ \| \ A)$ *ioco* spec then $(\mathsf{iut}_1 \ \| \ \mathsf{iut}_2)$ *ioco* spec. The authors showed that this property holds if the assumption A is input-enabled. This approach then requires the specification spec to be given as a single model rather than a set of components unlike our approach. They do not impose input-completeness of specifications which gives them an advantage with respect to our result.

Chapter IX

Conclusion

1 **Summary** 181
2 **Future research** 181

In this chapter, we conclude this thesis by describing the main objectives of the work, the goals we have achieved and the direction of future work.

1 Summary

Building correct systems has been the most difficult challenge for engineers and still continues to be so nowadays, due to the fact of growing system complexity and size. In this thesis, we explained the importance of component-based models to meet this challenge, and proposed an unified framework for both modeling and reasoning about the correctness of component-based systems formally. Hence, this thesis has been placed in the area of both modeling and testing of component-based systems.

We then defined a formalism based on *Barbosa*'s component definition [9, 31]. For this formalism, a trace semantics from causal functions was proposed as is usually done in control theory and dynamic systems design. The resulting formalism is then generic enough to subsume a large family of state-based formalisms. A number of theoretical results were also obtained. First, in order to deal with large systems, we defined the notion of an integration operator as the composition of two basic operators: the product and feedback. We then showed the generic results of compositionality independently of a given integration operator. We also obtained results related to the construction of a final object in the category of components. Taking advantage of the genericity of the formalism, we then defined both conformance and compositional testing theories, which by definition can be applied to any formalism instance of our framework.

2 Future research

The main direction of our work can be categorized according to the following:

– The proposed formalism is just an initial proposal of formalism to model complex systems. For its application in concrete cases, experience is needed in the case of real size systems. Another goal is to give a mathematical framework for a discipline, called *systems*

engineering, that has been fully tried and tested in the modeling of modern industrial systems, but has not been well-formalized. This will first require that we extend the formalism to take into account components heterogeneity (software, hardware, human) which is mainly characterized by how inputs are handled to provide observable outputs (i.e. discretely or continuously). In the context of *B. Golden*'s thesis [42], he defines a formalism which is abstract enough to unify, by using non-standard analysis techniques, different time treatments of components. On the contrary, systems considered in [42] are deterministic. The idea is then to try to combine our approach with that of [42].

– In systems engineering, mainly two kinds of operators play a crucial role in defining systems:

  1. Integration operators
  2. Abstraction/simulation operators.

  The first kind of operators has been widely discussed in this thesis, but not the second. Both abstraction and simulation operators aim to structure systems at many levels of description, from the most abstract to the most concrete until realization. These operators are classically brought together into only one which is similar to the operator of refinement classically used in software engineering [139, 140].

– It would be interesting to take data and not just values In and Out, into account. In order to do that, we first have to extend the signature over which components are defined by data. This would be done by replacing both inputs In and outputs Out sets with data structure specified using equational and algebraic specifications. This would lead us to extend our algorithms for test case generation. This extension will naturally be based on symbolic evaluation techniques as it has been done in [44, 53]. This will also require us to first extend our conformance relation *cioco* to *sicoco* [43].

– It would be interesting to address compositional verification in our framework. In order to do that, we would first have to define a logic (temporal) under our formalism, and then establish a certain number of properties on this logic such as defining a calculus and proving that it is correct and complete, showing that the logic is stable with respect to bisimulation, studying preservation of properties along integration operators, etc.

List of Figures

1 Représentation du système par une boîte noire . . . 2
2 vue compositionnelle d'un système complexe . . . 3
3 Classification des techniques de test . . . 5
4 Illustration du test de conformité . . . 7

I.1 Black box view of a system . . . 12
I.2 Compositional view of complex system . . . 13
I.3 Classification of testing techniques . . . 17

II.1 Examples of categories . . . 28

III.1 Graphical representation of LTS and LTS′ . . . 53
III.2 Example of a synchronization tree . . . 62

IV.1 Coffee machine . . . 75
IV.2 *ATM* component . . . 76
IV.3 Pedestrian crossing . . . 76
IV.4 Pedestrian crossing modeling . . . 77
IV.5 Transformation of an *IOLTS* into a component over $\mathcal{P}(\mathsf{Out} \times \_)^{\mathsf{In}}$ . . . 79
IV.6 Binary Mealy automaton . . . 87

V.1 Cartesian product . . . 90
V.2 Illustration of a system with feedback . . . 91
V.3 Relaxed feedback composite: $\hookleftarrow_{\mathcal{I}}(\mathcal{C})$ . . . 92
V.4 Syracuse's sequence component . . . 95
V.5 Examples of feedback composition . . . 96
V.6 Sequential composition . . . 101
V.7 Double sequential composition . . . 102
V.8 Extended cartesian product $\otimes_e$ . . . 103
V.9 Example: illustration of $\otimes_{\mathrm{e}}$ . . . 104
V.10 Synchronous product . . . 104
V.11 Synchronous product: $\circledast((\mathcal{C}_1, \mathcal{C}_2)) = \rhd_s(\mathcal{C}_0, (\mathcal{C}_1 \otimes \mathcal{C}_2))$ . . . 104
V.12 Concurrent composition . . . 105
V.13 Concurrent composition: $\oplus((\mathcal{C}_1, \mathcal{C}_2)) = \rhd_s(\rhd_s(\mathcal{C}_0, (\mathcal{C}_1 \otimes \mathcal{C}_2)), \mathcal{C}'_0)$ . . . 106
V.14 Synchronous parallel composition . . . 106
V.15 Extended concurrent composition $\oplus_e$ . . . 107
V.16 Example: illustration of $\oplus_e$ . . . 108
V.17 Encoder (on the left) and Decoder (on the right) . . . 110
V.18 Controller system $\mathcal{C}$ . . . 111
V.19 Gate system $\mathcal{G}$ . . . 112

V.20 Synchronous product $\mathcal{B} = \circledast(\mathcal{G}_1, \mathcal{G}_2)$ of $\mathcal{G}_1$ and $\mathcal{G}_2$ . . . 113
V.21 Sequential composition $\mathcal{K} = \triangleright_s(\mathcal{B}, \mathcal{O})$ of $\mathcal{B}$ and $\mathcal{O}$ . . . 113
V.22 Sequential composition $\mathcal{S} = \triangleright_s(\mathcal{C}, \mathcal{K})$ of $\mathcal{C}$ and $\mathcal{K}$ . . . 113
V.23 Model of a crosswalk, to be composed in a synchronous parallel composition with the traffic light model of Figure IV.4 . . . 114
V.24 Pedestrian crossing modeling . . . 115
V.25 $\odot(\mathcal{M}', \mathcal{M})$ . . . 115
V.26 Level crossing . . . 116
V.27 Controller system $\mathcal{C}$ . . . 117
V.28 Barrier system $\mathcal{B}$ . . . 117
V.29 Crossing level global model $\mathcal{S}$ . . . 118

VI.1 Conformance testing process . . . 132
VI.2 Relations between IMPS, MODS and SPECS . . . 134

VII.1 Illustration of *cioco* . . . 142
VII.2 Counter example for *quasi-reduction* . . . 144
VII.3 Finite computation tree for the coffee machine . . . 145
VII.4 Example of finite computation tree . . . 149
VII.5 Test purposes of the coffee machine . . . 151
VII.6 General view of the algorithm . . . 153
VII.7 Communication between the iut and the algorithm . . . 154
VII.8 Instantiating of the algorithm . . . 162

VIII.1 Architecture of a coffee machine in components . . . 165
VIII.2 Illustration of *cioco*'s compositionality . . . 167
VIII.3 Counterexample of compositionality . . . 169
VIII.4 Test purpose of the drink component . . . 178

List of Tables

II.1 Examples of categories 29

III.1 LTS and LTS′ 53

IV.1 The deterministic computational features 78
IV.2 The partial computational features 78
IV.3 The non-deterministic computational features 78

VI.1 Conformance testing elements 136

VII.1 Examples of conformance relations 139

Bibliography

[1] S. A. Slaughter, D.E. Harter, and M. S. Krishnan. Evaluating the cost of software quality. *Commun. ACM*, 41:67–73, August 1998.

[2] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

[3] D.F. D'Souza and Al.C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis(SM) Approach*. Addison-Wesley Professional, October 1998.

[4] H. Jifeng, L. Xiaoshan, and L. Zhiming. Component-based software engineering. In Dang Van Hung and Martin Wirsing, editors, *ICTAC*, volume 3722 of *Lecture Notes in Computer Science*, pages 70–95. Springer, 2005.

[5] J. J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theor. Comput. Sci.*, 249(1):3–80, October 2000.

[6] B. Jacobs and J. Rutten. A tutorial on coalgebras and coinduction. *EATCS Bulletin*, 62:222-259, 1997.

[7] H. Reichel. An approach to object semantics based on terminal co-algebras. *Mathematical Structures in Computer Science*, 5(2):129–152, 1995.

[8] B. Jacobs. Objects and classes, co-algebraically. In *Object Orientation with Parallelism and Persistence*, pages 83–103, 1995.

[9] L.S. Barbosa. Towards a calculus of state-based software components. *Journal of Universal Computer Science*, 9(8):891–909, August 2003.

[10] L.S. Barbosa and J.N. Oliveira. State-based components made generic. *Electronic Notes in Theoretical Computer Science*, 82(1):39 – 56, 2003. CMCS'03, Coalgebraic Methods in Computer Science (Satellite Event for ETAPS 2003).

[11] L.S Barbosa and S. Meng. Generic components. *Proceedings of First APPSEM-II Workshp*, March 2003.

[12] L.S Barbosa. Components as processes: An exercise in coalgebraic modeling. *FMOODS2000 - Formal Methods for Open Object-Oriented Distributed Systems*, pages 397–417, Sptembre 2000.

[13] L.S. Barbosa. *Components as coalgebras*. PhD thesis, Departamento de Informática Escola de Engenharia Universidade do Minho, 2001.

[14] P. Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi*, NATO ASI Series. Springer Verlag, 1993.

[15] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 14–23, Piscataway, NJ, USA, 1989. IEEE Press.

[16] E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[17] R. Hower. Software quality assurance and testing resource center. 1996-2011. Available at http://www.softwareqatest.com/.

[18] CNET. 10 great bugs of history. 2000. Available at http://www.bus.tu.ac.th/usr/angsana/IS301-1-42/Outline/greatbug.htm.

[19] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[20] E. M. Clarke and E. A Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[21] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[22] G. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.

[23] B. Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[24] E.W. Dijkstra. Notes on structured programming. pages 1–82, 1972.

[25] Institute O. Electrical and Electronics E. (ieee). *IEEE 90: IEEE Standard Glossary of Software Engineering Terminology*. 1990.

[26] B. Beizer. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[27] J. Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[28] E. Clarke, D. Long, and K. McMillan. Compositional model checking. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 353–362, Piscataway, NJ, USA, 1989. IEEE Press.

[29] E. Chang, Z. Manna, and A. Pnueli. Compositional verification of real-time systems. In *Proc. 9'th IEEE Symp. On Logic In Computer Science*, pages 458–465. IEEE Computer Society Press, 1994.

[30] H.M. van der Bijl, A. Rensink, and G.J. Tretmans. Compositional testing with ioco. In A. Petrenko and A. Ulrich, editors, *Formal Approaches to Software Testing (FATES)*, volume 2931 of *Lecture Notes in Computer Science*, pages 86–100, Berlin, 2004. Springer Verlag.

[31] S. Meng and L.S. Barbosa. Components as coalgebras: the refinement dimension. *Theor. Comput. Sci.(TCS)*, 351(2):276–294, 2006.

[32] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, pages 461–493, 1992.

[33] S. Eilenberg. *Automata, Languages and Machines*, volume C. Academic Press, New York, 1978.

[34] G. H. Mealy. A method for synthesizing sequentiel circuits. *Bell Systems Techn. Jour.*, 1955.

[35] R. Milner. A calculus of communicating systems. *Springer-Verlag New York, Inc, secaucus, NG, USA*, 1982.

[36] S. Brookes and A. W. Roscoe. An Improved Failures Model for Communicating Processes. *NSF-SERC Seminar on Concurrency, Pittsburgh, July 1984. Springer Lecture Notes in Computer Science(LNCS) 197.*, pages 281–305, 1985.

[37] N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

[38] M. Phalippou. *Relations d'implementation et hypothèses de test sur des automates à entrées et à sorties*. Thesis, Université de Bordeaux I, Septembre 1994.

[39] T. Jéron C. Jard. TGV: theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer*, 7(4):297–315, August 2005.

[40] J. Tretmans. Conformance testing with labelled transition systems : Implementation relations and test generation. *Computer networkss and ISDN systems*, 29(1):49–79, 1996.

[41] J. Rutten. Algebraic specification and coalgebraic synthesis of mealy machines. *Technical Report SEN-R0514, Centrum voor Wiskunde en Informatica (CWI)*, 2005.

[42] M. Aiguier, B. Golden, and D. Krob. Modeling of complex systems: A minimalist and unified semantics for heterogeneous integrated systems. *Technical report, 2011. Submitted to the journal "Applied Mathematics and Computation" - Available at http://www.lix.polytechnique/fr/golden/.*, 2011.

[43] L. Frantzen, J. Tretmans, and T.A.C. Willemse. A Symbolic Framework for Model-Based Testing. In K. Havelund, M. Núñez, G. Rosu, and B. Wolff, editors, *Formal Approaches to Software Testing and Runtime Verification – FATES/RV 2006*, number 4262 in Lecture Notes in Computer Science, pages 40–54. Springer, 2006.

[44] C. Gaston, P. Le Gall, N. Rapin, and A. Touil. Symbolic execution techniques for test purpose definition. In M. Uyar, A.Y. Duale, and M..A. Fecko, editors, *TestCom*, volume 3964 of *LNCS*, pages 1–18. Springer, 2006.

[45] J. Tretmans. Testing labeled transition systems with inputs and outputs. In *The 8th International Workshop on Protocol Test Systems*, pages 461–476, Ervy, France., 1995. In Cavalli, A. and Budkowski, S., editors,.

[46] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.

[47] V. Rusu, L.d. Bousquet, and T. Jéron. An approach to symbolic test generation. In *IFM '00: Proceedings of the Second International Conference on Integrated Formal Methods*, pages 338–357, London, UK, 2000. Springer-Verlag.

[48] A. Faivre, C. Gaston, and P. Le Gall. Symbolic model based testing for component oriented systems. In A. Petrenko, M. Veanes, J. Tretmans, and W. Grieskamp, editors, *TestCom/FATES*, volume 4581 of *Lecture Notes in Computer Science*, pages 90–106. Springer, 2007.

[49] P. Wolper. *Verification: Dreams and Reality*. Inaugural lecture of the course "The algorithmic verification of reactive systems", online available at http://www.montefiore.ulg.ac.be/ pw/cours/francqui.html.

[50] ISO. *Information technology, Open Systems Interconnection*. International standard IS 9646, ISO, Geneve, 1991.

[51] ISO/IEC JTC1/SC21 N6201. *Information Retrieval, Transfer and Management for OSI, Formal Methods in Conformance Testing, working draft*. Project 1.21.54 (Arles ouput). ISO, June 1991.

[52] ISO/IEC. *ISO/IEC JTC1/SC21 WG7, ITU-T SG 10/Q.8.Proposed ITU-T Z.500 and Committee Draft on Formal Methods in Conformance Testing*. CD 13245-1. ISO -ITU-T, Geneve, 1996.

[53] A. Touil. Exécution symbolique pour le test de conformité et le test de raffinement. *Doctorat de l'université EVRY-VAL-d'ESSONNE*, 6 décembre 2006.

[54] B. Jacobs. *Introduction to Coalgebra: Towards Mathematics of States and Observations*. Book draft, 2005.

[55] D. Pattinson. An introduction to the theory of coalgebras, 2003. Lecture Notes, Second North American Summer School on Logic, Language and Information.

[56] H.P Gumm. Elements of the general theory of coalgebras. *Notes of lectures given at LUATCS'99:Logic, Universal Algebra, Theoretical Computer Science, Johannesburg*, 1999.

[57] S. Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer Verlag, New York, Heidelberg, Berlin, 1971.

[58] M. Barr and C. Wells, editors. *Category theory for computing science, 2nd ed.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1995.

[59] E.G. Manes. Algebraic theories. *26 of Graduate Texts in Mathematics*, 1976.

[60] M. Arbib and E. Manes. Machines in a category. *Journal of Pure and Applied Algebra*, 19:9–20, 1980.

[61] M. Arbib E. Manes. *Algebraic approaches to program semantics*. Springer-Verlag New York, Inc., New York, NY, USA, 1986.

[62] P. Aczel. *Non-Well-Founded Sets*. CSLI, Stanford, CA, 1988.

[63] P. Aczel. Final universes of processes. *Mathematical Foundations of Programming Semantics*, 802:1–28, 1994.

[64] G. D. Plotkin. A structural approach to operational semantics. *Report DAIMI FN-19, Aarhus University, Aarhus*, 1981.

[65] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines—a survey. *Proceedings of the IEEE*, 84(8), August 1996.

[66] A. Arnold and M. Nivat. Comportements de processus. *In Colloque AFCET, Les mathématiques de l'Informatique*, 1982.

[67] T. Jéron. *Contribution à la génération automatique de tests pour les systèmes réactifs*. Habilitation à diriger les recherches, Université de Rennes 1, March 2004.

[68] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[69] J. Rutten and D. Turi. On the foundations of final semantics: Non-standard sets, metric spaces, partial orders. In *Proceedings of the rex workshop on semantics: foundations and applications, volume 666 of lecture notes in Computer Science*, pages 477–530. Springer-Verlag, 1998.

[70] Michael B. Smyth and Gordon D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM J. Comput.*, 11(4):761–783, 1982.

[71] J. Worrell. On the final sequence of a finitary set functor. *Theor. Comput. Sci.*, 338(1-3):184–199, 2005.

[72] P. Aczel and N.P. Mendler. A final coalgebra theorem. In *Category Theory and Computer Science*, pages 357–365, London, UK, 1989. Springer-Verlag.

[73] M. Barr. Terminal coalgebras in well-founded set theory. *Theor. Comput. Sci.*, 114(2):299–315, 1993.

[74] H. H. Hansen, D. Costa, and J. J. M. M. Rutten. Synthesis of mealy machines using derivatives. *Electr. Notes Theor. Comput. Sci. (ENTCS)*, 164(1):27–45, 2006.

[75] P.J Cameron. *Sets, Logic and categories*. Undergraduate Mathematics. Springer, 1999.

[76] D van Dalen, H.C. Doets, and H. de Swart. *Sets: Naive, Axiomatic and Applied*. Number 106. Pure and applied Math. Pergamum Press, 1978.

[77] G.N Raney. Sequential functions. *Journal of the (ACM)*, 5(2):177–180, April 1958.

[78] H. Wolff. Monads and monoids on symmetric monoidal closed categories. *Archiv der Mathematik*, 24:113–120, 1973. 10.1007/BF01228184.

[79] B. Kanso, M. Aiguier, F. Boulanger, and A. Touil. Testing of abstract components. In A. Cavalcanti, D. Déharbe, M. Gaudel, and J. Woodcock, editors, *ICTAC*, volume 6255 of *Lecture Notes in Computer Science*, pages 184–198. Springer, 2010.

[80] E.D. Sontag. *Mathematical control theory: deterministic finite dimensional systems (2nd ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1998.

[81] B. Golden M. Aiguier and D. Krob. Modeling of complex systems: A minimalist and unified semantics for heterogeneous integrated systems. 2011. Technical report, Ecole Polytechnique, Available at http://www.lix.polytechnique.fr/ golden/.

[82] C. A. R. Hoare and C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, 1985.

[83] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. In *Proceedings of the IEEE*, pages 1270–1282, 1991.

[84] E.A. Lee and P. Varaiya. *Structure and interpretation of signals and systems*. Addison-Wesley, 2003.

[85] E.A. Lee and S.A. Seshia. *Introduction to Embedded Systems - A Cyber-Physical Systems Approach*. Lee and Seshia, 1 edition, 2010.

[86] N. Yevtushenko, T. Villa, R.K. Brayton, A. Petrenko, and A.L. Sangiovanni-Vincentelli. Sequential synthesis by language equation solving. In *International Workshop on Logic and Synthesis*.

[87] A. Petrenko and N. Yevtushenko. Solving asynchronous equations. In *Proceedings of the FIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XI) and Protocol Specification, Testing and Verification (PSTV XVIII)*, FORTE XI / PSTV XVIII '98, pages 231–247, Deventer, The Netherlands, The Netherlands, 1998. Kluwer, B.V.

[88] C.G. Cassandras and S. Lafortune. *Introduction to discrete event systems*. SpringerLink Engineering. Springer Science+Business Media, 2008.

[89] S. Meng and B.K. Aichernig. A coalgebraic calculus for component based systems. *In Proceedings of FACS'03, Workshop on Formal Aspects of Component Software, Satellite Workshop of the FM*, September 2003.

[90] I. Hasuo, B. Jacobs, and A. Sokolova. The microcosm principle and concurrency in coalgebras, 2007. preprint, available from http://www.cs.ru.nl/ ichiro/papers. *I. HASUO, B. JACOBS, AND A. SOKOLOVA*, 2008.

[91] I. Hasuo, C. Heunen, B. Jacobs, and A. Sokolova. Coalgebraic components in a many-sorted microcosm. In *Conference on Algebra and Coalgebra in Computer Science*, pages 64–80, 2009.

[92] J. R. Burch, R.Passerone, and A.L. Sangiovanni-vincentelli. Overcoming heterophobia: Modeling concurrency in heterogeneous systems. In *Int. Conf. on Application of Concurrency to System Design*, 2001.

[93] E.A Lee Jie Liu Xiaojun Liu J. Ludvig S. Neuendorffer S. Sachs J. Eker, J.W. Janneck and Yuhong Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE,*

Special Issue on Modeling and Design of Embedded Software, Volume 91(1), Page(s): 127 - 144, Jan 2003.

[94] C. Brooks and E.A. Lee. Ptolemy ii - heterogeneous concurrent modeling and design in java. February 2010. Poster presented at the 2010, href="http://www.eecs.berkeley.edu/BEARS" Berkeley EECS Annual Research Symposium (BEARS).

[95] C. Hardebolle and F. Boulanger. Modhel'x: A component-oriented approach to multi-formalism modeling. *Models in Software Engineering - Workshops and Symposia at MoDELS 2007, Nashville, TN, USA, September 30 - October 5, 2007, Reports and Revised Selected Papers*, 5002/2008:247–258, June 2008.

[96] C. Hardebolle and F. Boulanger. Multi-formalism modelling and model execution. *International Journal of Computers and their Applications (IJCA)*, 2009.

[97] A. Jantsch. Models of embedded computation. In *Embedded systems handbook*. CRC Press, 2005.

[98] F. Arbab. Reo: a channel-based coordination model for component composition. *Mathematical. Structures in Comp. Sci.*, 14:329–366, June 2004.

[99] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time systems in BIP. In *4th IEEE International Conference on Software Engineering and Formal Methods (SEFM06), Pune*, pages 3–12, september 2006.

[100] G. Gößler and J. Sifakis. Composition for component-based modeling. *Sci. Comput. Program.*, 55(1-3):161–183, 2005.

[101] F. Arbab, C. Baier, J. J. M. M. Rutten, and M. Sirjani. Modeling Component Connectors In Reo By Constraint Automata. *Science of Computer Programming*, 61:75 – 113, 2006.

[102] F. Arbab and J. Rutten. A coinductive calculus of component connectors. In Martin Wirsing, Dirk Pattinson, and Rolf Hennicker, editors, *Recent Trends in Algebraic Development Techniques*, volume 2755 of *Lecture Notes in Computer Science*, pages 34–55. Springer Berlin / Heidelberg, 2003.

[103] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. Automated test and oracle generation for smart-card applications. In *International Conference on Research in Smart Cards (e-Smart'01), Volume 2140 of LNCS*, pages 58–70, 2001.

[104] H. Kahlouche, C. Viho, and M. Zendri. Hardware testing using a communication protocol conformance testing tool. In *In The International Work-shop on Tools and Algorithms for Construction and Analysis of Systems. (TACAS '99)*, March 1999.

[105] G. Bernot. Testing against formal specifications: A theoretical view. In S. Abramsky and T. Maibaum, editors, *TAPSOFT '91*, volume 494 of *Lecture Notes in Computer Science*, pages 99–119. Springer Berlin / Heidelberg, 1991.

[106] ISO/IEC. LOTOS-a formal description technique based on the temporal ordering of observational behaviour. In *Technical Report 8807, International Organization for Standards - Information Processing Systems - Open Sys- tems Interconnection*, 1988.

[107] IUT-T. Recommendation Z-100. specification and description language (SDL). In *Technical report*, 1994.

[108] L. Frantzen, J. Tretmans, and T.A.C. Willemse. Test generation based on symbolic specifications. (3395):1–15, 2005.

[109] G. Bernot. Testing against formal specifications: A theoretical view. In *TAPSOFT'91: Proc. of the Intl. Joint Conference on Theory and Practice of Software Development, Vol. 2*, pages 99–119, London, UK, 1991. Springer-Verlag.

[110] J. Tretmans. A formal approach to conformance testing. In *Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems VI*, pages 257–276, Amsterdam, The Netherlands, 1994. North-Holland Publishing Co.

[111] D. Lee and M. Yannakakis. Testing finite-state machines: State identification and verification. *IEEE Transactions on Computers*, 43:306–320, 1994.

[112] A. Petrenko and N. Yevtushenko. Conformance tests as checking experiments for partial nondeterministic fsm. In W. Grieskamp and C. Weise, editors, *FATES*, volume 3997 of *Lecture Notes in Computer Science*, pages 118–133. Springer, 2005.

[113] A. Petrenko and N. Yevtushenko. Testing from partial deterministic fsm specifications. *IEEE Trans. Comput.*, 54:1154–1165, September 2005.

[114] A. Petrenko, R. Petrenko, R. Groz, and S. Boroday. Confirming configurations in efsm testing. *IEEE Transactions on Software Engineering*, 30:2004, 2004.

[115] C. Bourhfir, R. Dssouli, and E.M. Aboulhamid. Automatic test generation for efsm-based systems. Technical report, 1043.

[116] F. C. Hennie. Fault detecting experiments for sequential circuits. In *FOCS'64*, pages 95–110, 1964.

[117] M. Yannakakis and D. Lee. Testing finite state machines. In *STOC*, pages 476–485. ACM, 1991.

[118] G. Gönenç. Conformance testing methodologies and architectures for osi protocols. chapter A method for the design of fault detection experiments, pages 368–375. IEEE Computer Society Press, Los Alamitos, CA, USA, 1995.

[119] A. Gill. *Introduction to the theory of finite-state machines*. McGraw-Hill, New York, 1962.

[120] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Eng.*, 4:178–187, May 1978.

[121] W. Chung and P. Amer. Improved on UIO sequence generation and partial UIO sequences. *Testing, and Verification, XII, Lake Buena Vista*, June 1992.

[122] R. De Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science (TCS)*, 34(1–2):83–133, nov 1984.

[123] R De Nicola. Extensional equivalence for transition systems. *Acta Inf.*, 24:211–237, April 1987.

[124] E. Brinksma. A theory for the derivation of tests. *Proc. 8th Int. Conf. Protocol Specification, Testing, and Verification (PSTV VIII)*, pages 63–74, 1988.

[125] I. Phillips. Refusal testing. *Theor. Comput. Sci.*, 50(3):241–284, 1987.

[126] E. Zinovieva. *Symbolic Test Generation for Reactive Systems with Data*. PhD thesis, IRISA/INRIA Rennes, France, November 2004.

[127] A. W. Heerink. *Ins and Outs in Refusal Testing*. PhD thesis, University of Twente, Enschede, May 1998.

[128] E. Brinksma and J. Tretmans. Testing transition systems: An annotated bibliography. In F. Cassez, C. Jard, B. Rozoy, and M.D. Ryan, editors, *MOVEP*, volume 2067 of *Lecture Notes in Computer Science*, pages 187–195. Springer, 2000.

[129] J. Tretmans. Testing techniques. 2002.

[130] R. Langerak. A testing theory for LOTOS using deadlock detection. In E. Brinksma, G. Scollo, and C.A. Vissers, editors, *Protocol Specification, Testing and Verification (PSTV)*, pages 87–98. North-Holland, 1989.

[131] L. Briones and E. Brinksma. A test generation framework for quiescent real-time systems. In *IN FATES 04*, pages 64–78. Springer-Verlag GmbH, 2004.

[132] S. Nogueira, A. Sampaio, and A. Mota. Guided Test Generation from CSP Models. In *Proceedings of the 5th international colloquium on Theoretical Aspects of Computing*, pages 258–273, Berlin, Heidelberg, 2008. Springer-Verlag.

[133] A. W. Heerink and G. J. Tretmans. Refusal testing for classes of transition systems with inputs and outputs. In T. Mizuno, N. Shiratori, T. Higashino, and A. Togashi, editors, *Proceedings of the IFIP TC6 WG6.1 Joint Intl. Conf. on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE X) and Protocol Specification, Testing and Verification (PSTV XVII)*, volume 107 of *IFIP Conference Proceedings*, pages 23–38, London, 1997. Chapman & Hall.

[134] M. van Osch. Hybrid input-output conformance and test generation. In Klaus Havelund, Manuel Núñez, Grigore Rosu, and Burkhart Wolff, editors, *Formal Approaches to Software Testing and Runtime Verification*, volume 4262 of *Lecture Notes in Computer Science*, pages 70–84. Springer Berlin / Heidelberg, 2006.

[135] J.C Fernandez, C. Jard, T. Jéron, L. Nedelka, and C. Viho. Using on-the-fly Verification Techniques for the Generation of Test Suites. Research Report RR-2987, INRIA, 1996.

[136] A. Sampaio, S. Nogueira, and A. Mota. Compositional verification of input-output conformance via csp refinement checking. In *ICFEM '09: Proceedings of the 11th International Conference on Formal Engineering Methods*, pages 20–48, Berlin, Heidelberg, 2009. Springer-Verlag.

[137] M. Bonsangue, J. Rutten, and R. Silva. A kleene theorem for polynomial coalgebras. In *In Foundations of Software Science and Computational Structures, 12th International Conference, FOSSACS 2009, volume 5504 of LNCS*, pages 122–136, 2009.

[138] L. Briones, C. Pasareanu, and D. Giannakopoulou. Assume-guarantee reasoning with ioco testing relation. In *ICTSS*, November 2010.

[139] J.A. Goguen and R.M. Burstall. Institutions: abstract model theory for specification and programming. *J. ACM*, 39:95–146, January 1992.

[140] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972. 10.1007/BF00289507.