CATEGORIES

for

SOFTWARE ENGINEERING

by

José Luiz Fiadeiro

from

THE UNIVERSITY OF LEICESTER

Springer

From a loving father

Contents

PART TWO – advanced topics

PART THREE – applications

Preface

Why another book on Category Theory?

In the past ten years, several books have been published on Category Theory either by Computer Scientists or having Computer Scientists as a target audience (e.g. [6,12,22,91,108], to which a precious collection of little gems [92] and the chapter cum book [93] should be added). Isn't the Working Computer Scientist spoilt with choice?

Although each of the above mentioned books presents an approach of its own, there is one aspect in common in their view of Computer Science: the analogy between arrows (morphisms) and (classes of) computations. This "type-theoretic" or "functional" approach corresponds to a view of Computer Science as a Science of Computation, i.e. a discipline concerned with the study of computational phenomena where the focus is on the nature and organisation of computations.

However, there is another view of Computer Science where the focus is, instead, on the development of computer programs or systems. This is the approach that supports, for instance, Software Engineering. From this point of view, arrows capture not computational phenomena, or abstractions thereof, but relationships between programs, or abstractions of programs, that arise in the development of computer systems: for instance, refinement of higher-level specifications into executable programs [102,107], and superposition of new features over existing systems [71].

Not surprisingly, this same difference in the points of view can be found when Logic (instead of Category Theory) is taken as a mathematical domain for formalising aspects of Computer Science. The "computations as proofs" paradigm is the one that corresponds to the "classical" application of Category Theory. Terms of the logic correspond to objects in a category of programs whose morphisms capture (partial) computations. From a logical point of view, the perspective that we take in this book is not centred on terms but on theories as system specifications. Morphisms then capture what in Logic is known as "interpretations between theories", the cornerstone for the formalisa-

tion of refinement in program development and other operations on specifications and system designs [17,81,82,83,84,105,106,107].

Category Theory can also be presented as the branch of Mathematics that, *par excellence*, addresses "structure". As the introduction will try to motivate, this is because Category Theory causes structure to emerge from relationships between objects as captured by arrows, and not extensionally as in Set Theory. Indeed, the term *morphism* often used for arrow in Category Theory, has in its etymology the notion of "preservation of form". What these structures are, or mean, is up to the "user". Hence, in the "classical" approach, we find applications of Category Theory that address the structure of computations. In the approach that is taken in this book, the structures that will be addressed are the ones that capture modularisation principles in software development, in particular, those that have been emerging in the guise of what has become known as Software Architectures [49].

The practical difference between the two approaches in what concerns Category Theory in general, and this book in particular, is that the reader will not find as many references to Algebraic Topology or related fields of Mathematics as applied, for instance, to Domain Theory. Although this book is still "mathematical", the Software Engineer will find the mathematics applied to objects of its day-to-day concerns: programs, object classes, specifications, designs, etc.

This approach can be also situated as belonging to the class of applications of Category Theory to General Systems Theory, namely in the tradition initiated in [51,52,63], an area of Science that, as the name indicates and the introduction will try to motivate, encompasses more than computational systems in the traditional sense. Through books aimed at wider audiences like [70], a unifying view of complex systems as they arise in disparate areas like Physics, Biology, Social Sciences, Economics and, yes, Informatics, has started to emerge (pun intended) which is a clear indication of new levels of maturity in Science in general and Informatics in particular. Hence, one of the purposes of this book is to help computing scientists and software engineers acquire formal tools that will enable them to follow and participate in this "new" culture.

A trait that is common to all these areas is a view of complex systems as communities of interacting, simpler, autonomous entities. Whereas, in areas like Biology or Social Sciences, the notion of "community" is intrinsic, its use in areas like Software Engineering is more artificial and is normally identified with methods and development techniques that, in the past few years, have attempted to tackle complexity by borrowing the organisational principles that can be recognised in such "natural" communities. Object-oriented modelling, agent-based programming, and component-based development all make use, in one way or an-

other, and with different emphasis, of this analogy. This brings us to the application area covered in the third part of this book.

CommUnity is the name of a language for parallel program design that is similar to Unity [19] but adopts instead an interaction model that places it in the realm of these more general and unifying approaches to systems. It addresses in particular the most recent trend – *service*-oriented software development, an (r)evolution of the popular object-object oriented modelling techniques for the "Internet-age" or what is becoming known as the "real-time" or "now" Economy. The distinctive feature of this new trend is in the emphasis that it puts in the externalisation and explicit modelling of interactions as first-class citizens so that systems can be more easily reconfigured, in run-time, and without interruption of vital services. These characteristics match, precisely, features that are intrinsic to Category Theory, namely those that distinguish it from Set Theory.

That is why, even if a substantial part of this book is illustrated with many examples borrowed from Software Engineering practice, we decided to devote three chapters to the application of Category Theory to CommUnity and its relationship to Software Architectures. This material will provide an opportunity for the reader to see the majority of the basic concepts and techniques of Category Theory applied in an integrated and systematic way. At the same time, the reader will be able to appreciate how far one can go in formalising software development methods and techniques in mathematical frameworks, which is essential for a mature Engineering discipline and, in my opinion, our responsibility as scientists.

This book and its many authors

Mentioning the connections between Category Theory and General Systems Theory is a good opportunity to give due credit to Joseph Goguen for the profound inspiration that his work has instilled, a sentiment that I know is shared by many other researchers in Computing Science. He has expressed his own views on the applications of Category Theory to Computing in several publications, most notably in [56], which include detailed summaries of technical results that we all have found very useful when categorical approaches were still regarded, at best, as "exotic" [59,60,65,104]. All of us regret that this material has never found its way to a textbook. Because it is not the aim of this book to fill this gap, the reader is strongly encouraged to consult this rich legacy at his or her own pace, bearing in mind that the list of references that is provided at the end is far from being complete.

Completeness is, in fact, a concern that has remained largely alien to my research agenda. (This observation is intended to make some faces smile but you can take it literally.) This book is more about a personal experience than the output of a rational process of identifying "the" or "a" complete categorical kernel that Software Engineers can use as a toolbox. The only justification for the inclusion of many concepts and constructions is that they were of help to me, either technically or aesthetically, making it likely that they will be directly useful for other people "like me". The exclusion of many other, even very basic ones[1], can be justified by the officious disclaimer "the line has to be drawn somewhere" but, most of the time, the reason is that I never really stumbled upon them in my daily routine or simply that I have not developed an understanding about them that is deep enough to add any value to what can be found in other books.

This personal experience has gone through well identifiable periods, each of which is associated with a different focus of interest in Computing and a group of people with which I worked directly and whose contributions I would like to acknowledge. My first contact with Category Theory was when I was studying Mathematics as an undergraduate at the University of Lisbon, and Prof. Furtado Coelho challenged the wrath of my fellow students, and his fellow staff, by including this most exotic, difficult and useless of subjects in the curriculum of Algebra II. The seed was there. Applications to Computing came a year later through the study of Goguen and Burstall's Theory of Institutions as a means of formalising Conceptual Modelling and Knowledge Representation Approaches, under the supervision and in collaboration with Amílcar and Cristina Sernadas. This is when things started to get serious.

In 1988, I started what has been a very rewarding collaboration with Tom Maibaum. During the three years I spent at Imperial College, we developed a categorical approach to object-oriented development based on temporal logic specifications, a marriage between my previous work with institutions and the ideas of Tom and Paulo Veloso on the nature of specifications in system development. Their contribution permeates the material that will be exposed in a way that cannot be referenced in the same way as a technical result. I have been very fortunate to be able to keep exchanging ideas and experiences with them: there are always hidden subtleties that only come to the surface when you are challenged by people like them and required to scratch the innermost levels of your understanding to satisfy their curiosity.

---

[1] Yes, I know that I will not be forgiven for having left out many "must-haves" such as the Yoneda lemma, cartesian-closed categories, topoï, monads, and so on.

During this same time, Félix Costa in Lisbon explored the categorical semantics of objects from the point of view of algebraic models of concurrency. When I returned to Lisbon in 1992, we brought it all together! These were very exciting and rewarding times. My collaboration with Félix provided much of the inspiration that led to my own understanding of the application of Category Theory to systems modelling. Although some specific contributions are acknowledged with references to his work, it would be unfair to reduce his contribution to this book to those occasions.

The next phase is devoted to the (then) emerging field of Software Architectures. It is centred on a language – CommUnity – that I developed together with Georg Reichwein and Tom Maibaum in an initial period, and later on with my students Antónia Lopes and Michel Wermelinger. It started as a proof of concept, showing that Goguen's categorical approach could be applied to Parallel Program Design in the style of Unity [19] and Interacting Processes [47]. Later on, it evolved into a prototype language for architectural modelling, a process that led me to understand many concepts that, until then, were blurred by the poor expressive power of the formalisms with which we had been working: non-determinism vs under-specification, refinement vs composition, and the role of signatures in separating computation and coordination. Some of this is revealed in part three of this book, but you will have to wait for another book to have the full story!

Although this "architectural" period is still very much alive (which does not mean that the other are already dead), another step in this evolution process has just taken place: the realization that Category Theory provides a perfect fit for what is required to support service-oriented software development, for instance in the sense of Web Services. But this step is so recent that, in fairness, I cannot acknowledge/blame anybody in particular for it… Nevertheless, it is unlikely that it would have happened so soon, or at all, if I had not accepted the challenge that Luís Andrade presented me with for working with ATX Software SA in putting these "theories" into "practice"… This has been a very rewarding process that has given me the opportunity to understand the implications of many of the structures and mechanisms that are intrinsic to Category Theory. I hope that I have managed to permeate this understanding in the way the material is exposed in the book.

This is probably why this book is being finalised now and only now: during each of the periods I mentioned, a book was planned and several parts were written… It is only now that the work of so many people has contributed to the contents that I can safely write it on my own without feeling guilty for excluding anybody in particular from co-authoring it.

It so happens that the last thrust in writing this book was made during my first year at the University of Leicester, a renowned address for research in Category Theory and its applications to Computer Science. Although I can honestly assure the reader that the decision to join Leicester was not for the advantages of promoting this book, it is certainly a privilege for the book to be bear this affiliation!

Finally, I should thank all the colleagues and students who have tread with me the paths that you can choose to follow in this book. The opportunity to discuss and lecture on many of the topics that are covered contributed decisively in helping me reach the level of maturity that made me decide that this book could be written. The feedback I received from tutorials presented at events such as ECOOP, ETAPS, FME, OOPSLA and TOOLS made me decide that this book should also be written.

I would like to thank in particular the University of Lisbon and the Technical University of Lisbon for the opportunities that they gave me to lecture much of the material covered in the first two parts, both at undergraduate and postgraduate level. These parts of the book were also covered in lectures given at the University of Coimbra, the Institute for Languages and Administration in Lisbon (ISLA), and the Federal University of Rio Grande do Sul (Brazil).

I am particularly grateful to Ugo Montanari for the invitation to lecture a 20 hours course on CommUnity and Software Architectures (part three of this book) as part of a postgraduate programme of the University of Pisa, and to Roland Backhouse and Jeremy Gibbons for the opportunity to lecture the same material at the Summer School on Generic Programming (Oxford 2002). Jeremy deserves a special thanks for sending me lots of comments and challenging questions!

Special acknowledgements

Although the previous paragraphs have given me the opportunity to acknowledge the contributions of a number of people and institutions, there are some specific colleagues to whom I would like to express my deepest gratitude for direct contributions to this book:

- *Félix Costa*: a significant part of the material covered in Part Two was developed jointly with him as reported in [32,33]. As already mentioned, much of my own understanding of Category Theory and its role in Computing Science grew up from discussions with him.
- *Antónia Lopes* and *Michel Wermelinger*: the fact that Part Three of this book was essentially extracted from [37,78] is a good indication

of how important and extensive their contribution has been. CommUnity as we know it today is as much theirs as it is mine.

- *Tom Maibaum*: his encouragement and support in the earlier phases of the production of the book were decisive.
- *Uwe Wolter*: he had the courage to use an early draft to support part of a course that he gave in 2002/2003 in Bergen. As a result, I received loads of feedback, which was invaluable for the final tuning of the material and the way it is presented.

Finally, I would like to thank the EPSRC for an eight-month visiting fellowship at King's College London in 1999 which gave me the opportunity to make a significant advance in the writing of the book, and to Janet Maibaum for her help in setting Microsoft Word up to the job[2]!

Leicester, September 2003

[2] Yes, this book is a proof that Category Theory is not reserved to users of a well-known typesetting system that I will not name… And this remark is not meant as a recommendation for the products developed by a company that I have already named…

1 Introduction

1.1 The social life of objects

One of the questions that we are most frequently asked is "What is Category Theory good for?" or "Why should I use Category Theory?" The question usually indicates a genuine and healthy reaction to the proposal of a new piece of mathematics that one is invited to learn, similar to the reaction we have each time we are asked to change our eating habits and, say, replace butter with olive oil when cooking... How is it going to make us happier, or healthier (which is not always the same)? When is the change justified?

The question also indicates that the nature and role of Category Theory, namely in Computing, is not completely clear to most people. The way we like to present Category Theory is as a toolbox similar to Set Theory: a kind of mathematical *lingua franca* in the sense that it can be used for formalising concepts that arise in our day-to-day activity. It constitutes, however, a richer toolbox in the sense that the instruments that it provides are more sophisticated, and thus makes it easier to model situations that are more complex and involve structured objects. On the other hand, because these instruments are more sophisticated than those of Set Theory, they require a dedicated learning effort. Briefly, in Category Theory one can do as much as in Set Theory, in an easier way when it comes to formalising and relating different notions of "structure", but at the cost of learning a few more concepts and techniques.

So, I would like to reformulate the original question as: "Having been brought up to think about the world in terms of sets, why should I now change and use another frame of reference?" The purpose of this book is to convince you that this change of reference is worth making, i.e. that this book will have been worth reading and that the concepts and techniques that it introduces should belong to the mathematical toolbox of the software engineer!

For many of "us", the key factor that justifies this change is related to the fact that, whereas concepts in Set Theory are typically formalised

extensionally, in the sense that a set is defined by its elements, Category Theory provides a more implicit way of characterising objects. It does so in terms of the relationships that each object exhibits to the other objects in the universe of discourse. The best summary of the essence of Category Theory that I know is from the logician Jean-Yves Girard[3]. For him, Category Theory characterises objects in terms of their "social lives".

In my opinion, this focus on "social" aspects of object lives is exactly the reason for the applicability of Category Theory to Computing in general, and Software Engineering in particular. To realise why this is so, one just needs to think that current software development methods, namely object-oriented ones, typically model the universe as a *society* of interacting objects. Agent-oriented methods are based on the same societal metaphor. This focus on interaction is not accidental; it is an attempt at tackling the increasing complexity of modern software systems. In this context, complexity does not necessarily arise from the computational or algorithmic nature of systems, but results from the fact that their behaviour can only be explained as emerging from the interconnections that are established between their components. By promoting interactions as a focal point in the definition of the structure of a system, one also brings software to the realm of natural, physical and social systems, something that seems to be essential for the development of well-integrated systems. Category Theory is advocated as a good mathematical structure for this integration precisely because it focuses on relationships and interactions! The work of Goguen on General Systems Theory [51,52,63], and recent books like [96], show exactly that.

This is why many of the examples that will be used throughout the book address what has become known in Software Engineering as "Software Architectures" [49], i.e. precisely the study of the gross modularisation principles that should allow us to design systems as, possibly standard, structures of smaller components and interconnections between them. The focus that Category Theory puts on morphisms, as structure-preserving mappings, is paramount for Software Architectures because it is the morphisms that determine the nature of the interconnections that can be established between objects (system components). Hence, the choice of a particular category can be seen to reflect, in some sense, the choice of a particular "architectural style". Moreover, Category Theory provides techniques for manipulating and reasoning about system configurations represented as diagrams. As a consequence, it becomes possible to establish hierarchies of system complex-

[3] What stronger evidence would one need of the close relationship between Logic and Category Theory...

ity, allowing systems to be used as components of even more complex systems (i.e. to use diagrams as objects), and for inferring properties of systems from their configurations.

The ultimate conclusion that we would like the reader to draw is that high-school education could well evolve in a way that equips our future generations with tools that are more adequate for the kind of systems that they are likely to have to develop and interact with. We believe that the teaching of mathematics could progressively shift from the set-theoretical approach that made it "modern" some decades ago (not to say in what is now "the last century") to one that is centred on interactions. This is, of course, a big challenge, mainly because it is not enough for the mathematical theory to be there: the right way needs to be found for it to be transmitted. We believe that recent books such as [74] are putting us in a path towards meeting this challenge, and we would hope this book to make a further contribution. However, we will be satisfied if the rationale for such a shift can, somehow, emerge from the way we will motivate and present the basics of Category Theory.

1.2 Categories versus sets

Let us consider an example in order to make clear what the difference of approach is between Set Theory and Category Theory. For this purpose, there is nothing better than showing how certain mundane set-theoretic constructions are modelled in Category Theory.

Consider, for instance, the characterisation of the empty set. In Set Theory, the empty set is characterised precisely by the property of not having any elements. There are several ways in which we can say this using a formal notation. Here is one:

$$(\forall x) x \notin \emptyset$$

The important point here is that any characterisation requires the use of the membership relation $\in$, i.e. the characterisation is made with respect to the elements that belong to the set.

Consider now the characterisation of the empty set in Category Theory. As discussed above, such a characterisation involves the definition of a "social life" of sets, i.e. it has to be made relative to the way sets interact with one another. Hence, the obvious question to ask first is "what is the social life of sets"?

The first important thing to understand is that Category Theory does not provide an answer to questions like this one; it is up to whoever is formalising a particular domain of discourse to come up with a definition of "social life" that is convenient. Convenience here has to be

measured against the formalisation activity that is being undertaken, possibly as an abstraction of real-world phenomena. Hence, it is not subject to mathematical proof. What Category Theory does is require some basic properties of such a "social life" so that the whole mathematical machinery that we are about to describe can be applied successfully. Such basic properties are defined in section 2.2. In a nutshell, they prescribe ways in which one is related to oneself and the way one's relations' relations are our own relations.

Each time, during both classes and industry-oriented tutorials, our audiences were first confronted with the need for defining a social life for sets, the immediate answer was:

"*Set A is related to set B iff* $A \subseteq B$"

Discussing the reason why this answer comes up spontaneously is well beyond the scope of this book[4]. The characterisation of the empty set based on this social life of sets is quite easy: the empty set is the only set that is related to every other set by this particular relationship.

Prompted for another definition of social life, the following answer was given several times:

"*Set A is related to set B iff* $A \cap B \neq \emptyset$"

According to this definition, the empty set is the only set that is not related to any set; all non-empty sets relate at least to themselves. Incidentally, we shall see in section 2.2 that this definition of social life does not define a category, one of the reasons being that, in a category, every object is at least related to itself in a canonical way. Nevertheless, the example is useful for showing that changing the definition of social life may lead to quite different characterisations of the same objects.

We typically have to force the audience to come up with the "standard" definition of social life between sets:

"*A social relationship between a set A and a set B is given as a total function* $f: A \to B$"

The characterisation of the empty set in this case is very similar to the first one: the empty set is such that, given any other set *A*, there is one, and only one, function to *A* – the empty function. Notice that, although the concepts of "contains", "intersection" and "function" are defined via the membership relation, the characterisation of the empty set given in the three cases does not involve it directly.

For further evidence that the nature of objects changes according to the "social life" that is of interest, consider the characterisation of singletons. According to the first definition, a singleton is such that only the empty set and the singleton itself relate to it. In the second case, a

[4] Nevertheless, the author is willing to collaborate in any research project that aims at understanding the psychology of modern formalism.

singleton is such that it relates to the sets that contain the element: *{a}* relates to *B* iff $a \in B$. This is a good evidence for the fact that the second definition is not very "categorical": it reduces to set membership and makes the identity of the elements visible. This is also the case with the first definition: in both cases, two singletons are socially equivalent, in the sense that they relate to any other set in the same way, iff they are equal. Incidentally, a possible set-theoretic characterisation of a singleton set *{a}* is:

$$x \in \{a\} \text{ iff } x=a$$

The use of equality between elements makes clear the need to look inside the set.

The third definition characterises singletons as sets into which there is one, and only one, total function from any other set. Indeed, the function being total, a singleton offers no choice for the mapping to be established: everything is mapped to this single element. According to this definition, two singletons cannot be distinguished because they relate in exactly the same way with the other sets. This is quite intuitive because, not being able to use set membership, we cannot look inside the singletons and notice that they have different elements. Hence, this social life is a better abstraction from set membership than the other two.

The "social" way of characterising objects is, in fact, similar to the way, in object-oriented programming, the view of objects that matters for building systems is that of the methods that objects make available to the other objects to interact with. However, the view that matters for their implementation may be different, reflecting the fact that different uses reflect different structural views of the same concept (which, in Category Theory, means different categories).

Other examples could be given from branches of engineering or our day-to-day praxis. For instance, another example is the way we understand devices as mundane as hi-fi systems. When assembling a hi-fi system from separate components, what is important is to know how each component can be connected to the others. The way they are implemented in terms of microcircuits probably explains the restrictions on the connections that can be established, but it is something that a user would like to see abstracted away. The characterisation of human behaviour can also be used as an example. There is probably a neurophysiological justification for what we call "shy" or "expansive" characters, but the meaning that we normally attach to such features of human nature is derived from the way people interact with us.

In brief, like all mathematics, Category Theory is about providing us with abstraction mechanisms. In our opinion, the fact that these mechanisms relate to interaction make Category Theory particularly

suited for certain aspects of Computing Science, and Software Engineering in particular.

1.3 Overview of typical application areas

In this section, we summarise some of the main applications areas of categorical techniques we know, with references to the literature, bearing in mind that our focus is Software Engineering and not Computer Science as a whole. We strongly suggest the reader to consult [26,56] for excellent introductions to (and overviews of) the field.

1.3.1 General systems theory

When one thinks of it, this is really early work. Goguen started exploring Category Theory as a mathematical toolbox in the early 70s, applying it to General Systems Theory [51,52,63]. The famous *motto* [56] "*given a category of widgets, the operation of putting a system of widgets together to form a super-widget corresponds to taking a colimit of the diagram of widgets that shows how to interconnect them*" was first applied to mathematical models of system behaviour. We also find in this area the origins of latter applications to object-oriented modelling [57]. In fact, one of his early works is even called, very appropriately, "objects" [53].

But Goguen's early work is just one example of the many applications of Category Theory to the area of General Systems. We would like to encourage the non-specialist to get acquainted with the field through one of the recent books that have appeared in the wider market of "Science" like [70,96].

1.3.2 Algebraic development techniques

This is one of the most "typical" areas of the applications of Category Theory to Computer Science. One of the earlier and most influential mouvements started in the second half of the 70s with the work of a group of researchers at IBM (Joseph Goguen, Jim Thatcher, Eric Wagner and Jesse Wright), very aptly named ADJ (for *adj*unction), around the initial semantics of abstract data type specification [64]. The pioneering work of Rod Burstall and Joseph Goguen around the language CLEAR [17,18] showed how simple universal constructions (e.g. colimits) could be used to give semantics to operations for structuring specifications (e.g. computing the sum of two specifications, parameterising a specification, etc). The area produced textbooks [e.g. 28,29,75,94] as well as surveys [8,26] that provide a good showcase for

"categories at work", namely in what concerns elegance and economy of means.

One of the finest hours of this research programme was the development of the theory of Institutions, a categorical formalisation of the notion of Logic developed by Goguen and Burstall in the early 80s [61]. The aim of this effort was to provide a mathematical framework in which the specification building operators defined for the language CLEAR could be made independent of the underlying logic and thus available for other specification formalisms. The theory of Institutions is, in our opinion, one of the best examples of the usefulness of Category Theory as a "unifier" of concepts and techniques developed by different research teams in response to different or similar needs. As a "science", Computing is still very young and, hence, fragmented in that it is often difficult to find the "universal laws" that we recognise as the objects of study in other sciences. Through the theory of Institutions, Category Theory showed to be a powerful tool for abstracting from individual and uncoordinated efforts some "universal laws" (or "dogmas" as they are called in [56]) that apply to specifications in general [e.g. 97]. This categorical framework has been subsequently extended and put to use, for example, in applying algebraic specification techniques to other computational paradigms (e.g. object-oriented), and defining ways for specifications to be mapped from one formalism to another.

1.3.3 Concurrent and object-oriented systems

Concurrency theory is another area to which categorical techniques have been applied in the "engineering" view that interests us in this book. This domain of application is dominated by the work of Glynn Winskel [110,111,112] who showed how models for concurrent system behaviour like transition systems, synchronisation trees and event structures can be formalised in Category Theory. The idea is that each process model is endowed with a notion of morphism that defines a category in which typical operations of process calculi are given as universal constructions. This work was continued in the 90s in collaboration with Mogens Nielsen and Vladimiro Sassone [99], exploring the use of adjunctions as a means of translating between different models. The chapter [113] provides an excellent overview of this application area.

This stream of work was extended for giving semantics to object-oriented systems by J.Félix Costa, Hans-Dieter Ehrich, and Amílcar and Cristina Sernadas [e.g. 20,21,25], also with contributions from Goguen [24]. In a nutshell, these authors showed how universal constructions can be used to express object-oriented features like encapsulation, inheritance and composition in concurrency models endowed with richer notions of state. The work of Goguen on sheaf-theoretic models [57]

in particular establishes an interesting connection to the earlier work on General Systems Theory.

Applications of Category Theory to the (logical) specification of concurrent and object-oriented systems can also be found. These include our work in the late 80s and early 90s [38,39,40,43]. Basically, we showed how the techniques developed for the modularisation of equational and first-order logic specifications of abstract data types could be applied to concurrent and object-oriented systems by using the modal and temporal logics that had been developed for the specification of reactive systems (e.g. [13]).

Briefly, the idea was to fit the specification of such systems in the "institutional" picture as set up by Goguen and Burstall [32,33]. We later showed that our work is directly related to the General Systems tradition [31], in the sense that the module structure defined by our categorical formalisation is directly compositional over the run-time structure of the system. This lead to applications of the categorical techniques to parallel program design [44] in which the notion of morphism captures what in the literature is known as superposition or superimposition [19,47,71].

There is also a substantial amount of work in the application to concurrent and object-oriented systems of the algebraic approach to abstract data type specification [7], most notably by Egidio Astesiano and his colleagues in Genova. Basically, these approaches present different alternative ways of bringing states and transitions to the universe of discourse of abstract data types.

1.3.4 Software architectures

Applications of categorical techniques to the semantics of interface description and module interconnection languages were developed by Goguen in the early 80s [54] and more recently recast in the context of the emerging interest in Software Architectures [58]. These applications are in the tradition of the algebraic approach to abstract data types, more specifically in what has become known as "parameterised programming" [55], in the sense that they capture functional dependencies between the modules that need to be linked to constitute a given program.

The view of architectures that is captured in this way is somewhat different from the one followed in the work of Dewayne Perry, David Garlan and other researchers who have launched Software Architectures as we know them today [14,49,101]. This more recent trend focuses instead on the organisation of the *behaviour* of systems as compositions of components ruled by protocols for communication and synchronisation. As explained in [2], this kind of organisation is founded on *interaction* in the behavioural sense, which explains why formalisms

like the CHAM [16] are preferred to the functional flavour of equational logic for the specification of architectural components.

This why, in the early 90s, we proposed a categorical toolset for architectural description based on our work on the formalisation of parallel program design techniques [34,42]. The idea is that, contrarily to most other formalisations of architectural concepts that can be found in the literature, Category Theory is not another semantic domain in which to formalise the description of components and connectors. Instead, through its universal constructions, it provides the very semantics of "interconnection", "configuration", "instantiation" and "composition", i.e. that which is related to the gross modularisation of complex systems.

1.3.5 Service-oriented software development

But there is more to Category Theory than the ability to support such interaction-based views of system behaviour. By relying on "local naming" (as shown by the impossibility of distinguishing between singleton sets), Category Theory requires all such interactions to be modelled explicitly and outside the participating objects. This is exactly one of the aspects that makes the distinction between service and object-oriented system development.

Clientship, i.e. the ability to establish client/supplier relations between objects, leads to systems of components that are too tightly coupled and rigid to support the levels of agility that are required to operate in environments that are "business time critical", namely those that make use of Web Services, B2B, P2P, or otherwise operate in what is known as "internet-time". Because interactions in object-oriented approaches are based on *identities*, in the sense that, through clientship, objects interact by invoking specific methods of specific objects (instances) to get something specific done, the resulting systems are too rigid to support such levels of agility. Any change on the collaborations that an object maintains with other objects needs to be performed at the level of the code that implements that object and, possibly, of the objects with which the new collaborations are established. On the contrary, interactions in a service-oriented approach should be based only on an abstract description of what is required, thus decoupling the "what one wants to be done" from the "who does it and how". This is, precisely, the discipline of interconnection that Category Theory enforces.

Hence, our last claim, in support of the last paragraph of section 1.1, is that Category Theory is, definitely, the mathematics of the Internet-age (and beyond)!

1.4 What can be found in this book

This book emerged from tutorials and courses given in the past few years both in academia and more industry-oriented fora like OOPSLA, ObjectWorld, TOOLS, ECOOP, and FME. This experience showed that there was an audience for this particular way of exposing Category Theory.

The book is structured in three parts, leaving room for different reading/teaching paths to be followed. With respect to most other books in the market, this one will use examples of a different nature, focus on, and give more emphasis to aspects that are less common in other fields of Computing. It also adopts a different pace altogether.

- Part One, i.e. chapters 2-5, covers some of the "basics" of Category Theory, the “bare essentials” that are addressed in any book, from graphs to universal constructions and functors. However, a different emphasis and tone are used that are meant to be more appealing and accessible to software engineers. Mathematically mature readers may, hopefully, appreciate a different way of exposing and illustrating these familiar concepts and constructions.
- The material included in Part Two, chapters 6 and 7, is of a more advanced nature, not only because it is more challenging from a mathematical point of view, but because it makes appeal to an additional level of maturity in so far as Computing Science is concerned, namely the use of multiple formalisms as supporting complementary viewpoints. There, the reader will find material that only a few other books cover in comparable depth, with a strong emphasis on functor-based constructions like fibrations, and ending with a covering of adjunctions that deviates somewhat from the standard coverage. Again, examples are drawn from areas that have normally been confined to papers such as institutions and models of concurrency. The sections in this part should be accessible to anyone with basic knowledge of Category Theory, but a quick travel through Part One will help the reader become familiar with the notation and the examples that are used in Part Two.
- Part Three offers the chance of seeing Category Theory “at work” in a more ambitious project – giving semantics to CommUnity, a prototype language for architectural modelling. It can be ignored by readers who are not particularly interested in the applications to Software Engineering. On the other hand, it can be followed, to a large extent, without the material exposed in Part Two. This means that a novice to Category Theory but interested in Software Engi-

neering, or anyone whose goal is to understand CommUnity or setup a course that aims at teaching (any subset of) the material in Part Three, can safely skip Part Two. It will be necessary to go back to Part Two in order to understand in full the mathematical structures that relate to Software Architecture, but this may be done later on, once the reader feels more at-ease with the mathematics.

Parts one and two use examples taken directly from Meyer's book on Eiffel [88] to illustrate definitions and constructions. Readers who are not familiar with the particular notation of Eiffel, but are used to object-oriented modelling, will hopefully be able to understand the examples without much effort. The choice for Eiffel instead of a more "modern" language has to do with two facts. On the one hand, it *was* modern when the book started to be written... On the other hand, it is one of the object-oriented languages that has solid foundations, which allows it to be used to illustrate directly mathematical constructions such as the ones you are about to read.

The use of Eiffel allows us to illustrate applications of Category Theory to the more "static" aspects of system modelling, namely to what is related with classification (inheritance). Other examples are brought to bear from "Specification Theory" that involve Logics and Algebraic Models for system behaviour in order to show how the more "dynamic" and "evolutionary" aspects can also be handled. These different aspects come together in Part Three.

This means that different reading/teaching paths can be established that are based on the examples: one can follow the "Eiffel path" for a lighter approach, perhaps more suitable for "practitioners"; or one can follow the "Specification Theory path" that will enable the "scientists" to visit more challenging places and reach the end of Part Two. But who are the true Software Engineers if not the people who can combine both? Therefore, the best is to read the book from the beginning to the very end!

PART ONE – basics

2 Introducing categories

2.1 Graphs

A distinctive attribute of Category Theory as a mathematical formalism is that it is essentially graphical. This means that most concepts and properties can be defined, proved and/or reasoned about using diagrams of a formal nature. This diagrammatical nature of Category Theory is one aspect that makes it so applicable to Software Engineering.

Therefore, it is not surprising that the most basic and, hence, the first definition in this book is that of graphs.

2.1.1 Definition – graph

A graph is a tuple $<G_0,G_1,src,trg>$ where:

- G_0 is a collection[5] (of nodes)
- G_1 is a collection (of arrows)
- *src* maps each arrow to a node (the source of the node
- *trg* maps each arrow to a node (the target of the node).

We usually write $f:x \to y$ to indicate that $src(f)=x$ and $trg(f)=y$.

Between two nodes there may exist no arrows, just one in either direction, or several arrows, possibly in both directions.

The attentive reader will have noticed that our very first definition still uses Set Theory, even if only informally. This may appear confusing, especially after our lengthy discussion of the merits of Category Theory with respect to Set Theory. However, we need a meta language for talking about graphs (and categories, and ...) which cannot, of course, be the object language itself (i.e. that of Category Theory). Hence, we will use the "informal" language that is typical of Mathemat-

[5] Questions of "size" arise here because we shall soon be talking about the graph of graphs and constructions of a similar nature. This is why we use the term "collection" instead of "set". See, for instance, [79] for a full treatment of such questions.

ics which, as also acknowledged in the introduction, is full of set-theoretic concepts.

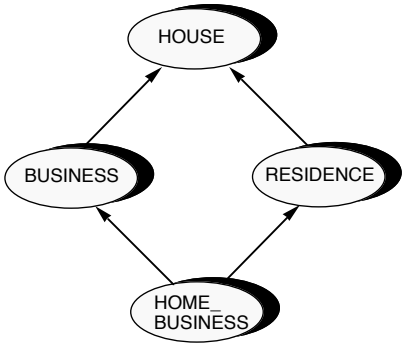
2.1.2 Example – sets and functions

1. The most "popular" graph in this book (and in any other book we know) is the graph whose nodes are the sets and whose arrows are the total functions.
2. To illustrate the fact that different graphs may share the same nodes, we introduce what is, perhaps, the second most popular graph in the book – the graph whose nodes are, again, the sets but whose arrows are the partial functions, i.e. functions that may be undefined on given elements of the source set.

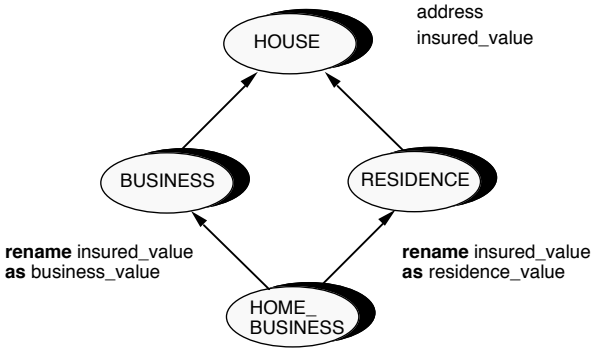
2.1.3 Example – class inheritance hierarchies

A typical example of the use of graphs in Computing is class inheritance hierarchies. These are graphs whose nodes are object classes and for which the existence of an arrow between two nodes (classes) means that the source class inherits from the target class.

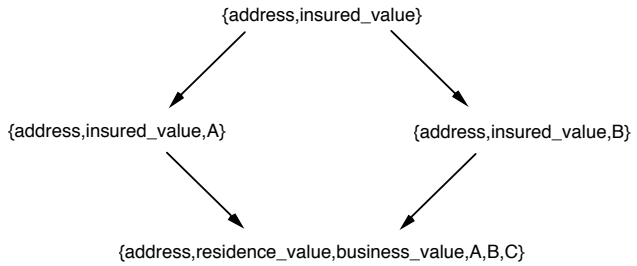
The following inheritance diagram was taken from [88]. It tells that class *home_business* inherits from both *business* and *residence*, each of which inherits from *house*.



In class inheritance hierarchies, there exists at most one arrow between two nodes: either a class inherits from another or it does not. However, arrows can carry more information. For instance, when one class inherits from another one, some renaming of the features of the original class may be required. Such renamings may be associated with the arrows of a class inheritance diagram.



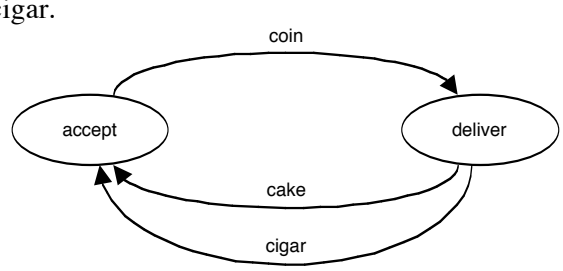
Each such enriched class inheritance diagram defines a subgraph of the graph of sets and (total) functions – classes are represented through their sets of features and the renamings through the functions that they induce. Notice that the arrows of the class inheritance hierarchy and the functions that operate the renamings point in opposite directions.



2.1.4 Example – transition systems

Another very common example of graphs is transition systems. Every transition system constitutes a graph whose nodes are the states and whose arrows are the transitions.

Below we show the transition system that models the behaviour of a vending machine that accept coins and, for each accepted coin, delivers a cake or a cigar.



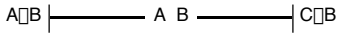
As hinted in the introduction, there are many (and deep) relationships between Logic and Category Theory. The next two examples

convey some of these relationships. See [73] for a textbook on this subject.

2.1.5 Example – logical entailment

One of the possible views that one can have of a "logic" is through the notion of a sentence being a consequence of, or derivable from, another sentence. This notion of consequence can be represented by a graph whose nodes are sentences and whose arrows correspond to "logical entailment".

The following are examples of three nodes and two arrows of the graph that captures logic entailment in propositional logic:

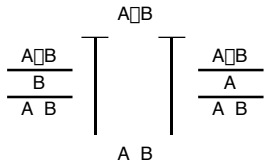


We can add detail to entailment and distinguish between different possible ways in which a sentence can be a consequence of another, i.e. by taking arrows to be proofs:

2.1.6 Example – proof systems

Every proof system constitutes a graph whose nodes are sentences and whose arrows are proofs.

The following are examples of two nodes and two arrows of the graph that corresponds to natural deduction in propositional logic:



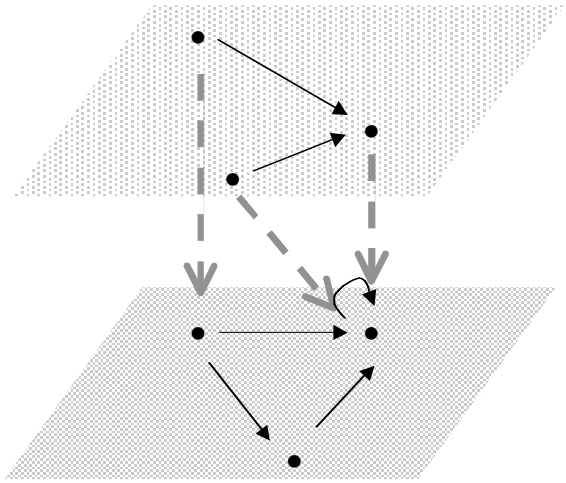
One should be aware of the difference between a graph and the graphical representation that is chosen for its nodes and arrows. The latter is normally chosen according to what is the traditional notation of the domain of application. This is why we deliberately used "turnstiles" for "arrows" above. In the case of the proof system, we could even have omitted the turnstile because the proof itself is the arrow. However, from a graphical point of view, using both the proof and the turnstile seems to make the representation more clear.

Graphs have a "social life" of their own that is useful to know about. For instance, the graphs introduced in examples 2.1.5 and 2.1.6 are intuitively related through an operation that adds detail (or forgets, depending on the point of view), namey proofs. Such relationships between graphs are called graph homomorphisms:

2.1.7 Definition – graph homomorphism

A *homomorphism* of graphs $\phi:G \to H$ is a pair of maps $\phi_0:G_0 \to H_0$ and $\phi_1:G_1 \to H_1$ such that for each arrow $f:x \to y$ of G we have $\phi_1(f):\phi_0(x) \to \phi_0(y)$ in H. That is, nodes are mapped to nodes and arrows to arrows but preserving sources and targets.

The figure below illustrates a situation in which the node-component of a homomorphism(represented by the dashed arrows) is neither injective (two nodes are collapsed) nor surjective (one of the nodes of the lower graph has no counterpart above). When the node component is not injective, arrows may get mapped to endo-arrows, i.e. arrows with the same source and target, as illustrated. In the example, the arrow-component (left implicit) is injective but not surjective.



2.1.8 Example – abstracting entailment from proofs

There is a "canonical" homomorphism between proof systems and consequence systems that consists of the identity on nodes and collapses any non-empty set of proof arrows between two nodes into just one consequence arrow. That is, this homomorphism "forgets" the details of the proofs, retaining just the fact that one exists to justify the consequence relation. For instance, when applied to the example in 2.1.6 it would deliver just one arrow from $A \wedge B$ to $A \ B$ as in 2.1.5.

Notice that the relationship between the inheritance graph used in 2.1.3 and the corresponding graph of feature sets and renamings cannot be captured by a homomorphism because the source and target of arrows is reversed. Absence of a homomorphism in these circumstances seems to be accidental because the direction of the arrows is somewhat arbitrary: we could well have chosen a graph with the same collections of nodes but with the arrows reversed.

2.1.9 Definition – duality

The graph obtained from another one by reversing the direction of arrows is called its *dual*: i.e. the dual of $G=<G_0,G_1,src,trg>$ is $G^{op}=<G_0,G_1,trg,src>$. A homomorphism from a graph G to the dual H^{op} of a graph H is said to be *contravariant* between G and H.

An essential step towards defining the notion of category concerns paths in graphs, i.e. what supports the move from direct to more "global" social relationships.

2.1.10 Definition – path in a graph

Let G be a graph and x, y nodes of G. A *path* from x to y of length $k>0$ is a sequence $f_1...f_k$ of arrows of G (not necessarily distinct) such that:

- $src(f_1)=x$
- $trg(f_i)=src(f_{i+1})$ for $1\leq i\leq k-1$
- $trg(f_k)=y$

For every x, the path of length 0 at x (the empty path at x) from x to x is, by convention, the empty sequence.

The collection of paths of G of length k is denoted by G_k. Hence:

- G_0 corresponds to the collection of nodes
- G_1 corresponds to the collection of arrows
- G_2 corresponds to the collection of pairs of composable arrows.

2.2 Categories

Categories provide an abstraction over graphs by making paths the basic working elements – what are called *morphisms*; paths provide richer information about "social life" than just one-to-one relationships. For that purpose, categories add to graphs an identity map that "converts" nodes to morphisms (null paths), and a composition law on morphisms that internalises path construction. Morphism composition is required to be associative as for path concatenation. This means that morphisms have no internal structure that can be derived from the order of composition.

2.2.1 Definition – category

A category $\boldsymbol{C}$ is a triple $<G,;,id>$ where:

- G is a graph, often denoted by $\boldsymbol{graph(C)}$
- ; is a map from G_2 into G_1 (called the *composition* law); for every fg in G_2, we denote by $f;g$ the arrow that results from the composition
- id is a map from G_0 into G_1 (called the *identity* map); for every node x, we denote by id_x its identity arrow

such that, for every f in G_1, fg in G_2, and fgh in G_3:

- $src(f;g)=src(f)$ and $trg(f;g)=trg(g)$
- $src(id_x)=trg(id_x)=x$
- $(f;g);h=f;(g;h)$
- if $f:x \to y$, $id_x;f=f;id_y=f$

The nodes (resp. arrows) of G are also called the *objects* (resp. *morphisms*) of $\mathbf{C}$. The collection of objects of $\mathbf{C}$ is denoted by $|\mathbf{C}|$. We will often use the notation $c:\mathbf{C}$ to indicate that c is a object of $\mathbf{C}$, what we sometimes also call a $\mathbf{C}$-object. Given objects x and y, $hom_{\mathbf{C}}(x,y)$ denotes the collection of morphisms from x to y.

The properties required are straightforward: the first two establish the types of the composite and identity arrows, respectively; the other two establish associativity of composition and the identity laws.

2.2.2 Example – "the" category of sets

The category ***SET***: objects are sets, morphisms are (total) functions between them, composition is functional composition – i.e. $(f;g)(x)=g(f(x))$ – and the identity map assigns to every set the identity function on that set. Because function composition is associative and the identity map is both a left and right identity for function composition, all the conditions are met.

Function composition is usually denoted by the symbol $\circ$ and the order of the arguments reversed, i.e. given $f:x \to y$ and $g:y \to z$, the composed function $f;g$ is also denoted by $g \circ f$. This is the "application order": $(g \circ f)(a)=g(f(a))$ for every $a \to x$. Most textbooks on Category Theory adopt this alternative notation for the composition law. Ultimately, the choice between one notation and the other is a matter of taste or convenience[6]. Our choice was motivated by the fact that it is closer to the diagrammatic notation (arrow sequencing) and, hence, supports the diagrammatic forms of reasoning that normally appeal to software engineers. Besides, as evidenced by the definition, the application order derives too much from set membership, i.e. reasoning with it is normally too close to the traditional set-theoretic approach from which we are trying to get away.

2.2.3 Example – graphs

The category ***GRAPH*** has graphs as objects and its morphisms are the graph homomorphisms. The composition law is defined as follows: for every pair ϕ and ψ of graph homomorphisms such that $src(\psi)=trg(\phi)$,

[6] In fact, this was one of the major decisions that had to be made before starting writing! Many people are put off reading a book because they are used to the "other" notation. Hence, ultimately, the decision was made taking into account the intended audience (and for "pedagogical" reasons as explained).

$(\square;\square)_0=(\square_0;\square_0)$ and $(\square;\square)_1=(\square_1;\square_1)$. The identity map is defined as follows: for every graph G, $(id_G)_0=id_{G_0}$ and $(id_G)_1=id_{G_1}$.

proof

By taking graphs as nodes and graph homomorphisms as arrows, we do obtain a graph. The identity and associativity properties are inherited for each component (node and arrow) from the corresponding properties of functions between sets.

2.2.4 Example – logical entailment

The category ***LOGI*** has as objects sentences, and morphisms between sentences correspond to the existence of a logical entailment.

proof

In this category there is at most one morphism between two objects. So, the identity and composition equations are trivially satisfied. We just have to prove the existence of endomorphisms (reflexivity) and a composition law (transitivity). But (A ⊢ A) is a tautology, and, from (A ⊢ B) and (B ⊢ C), we can conclude (A ⊢ C) (cut rule)!

In fact, every pre-order defines a category:

2.2.5 Proposition – pre-orders

Every pre-order $<S,\leq>$, i.e., every set equipped with a reflexive and transitive relation, defines a category $\boldsymbol{S}_{\leq}$ as follows:

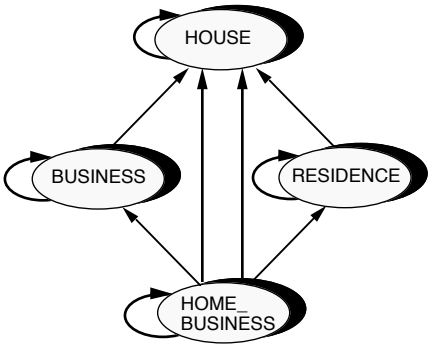
- $|\boldsymbol{S}_{\leq}|=S$ and there is a morphism between x and y in S iff $x\leq y$
- composition is defined by applying the transitivity law
- identity morphisms are defined by applying the reflexivity laws.

2.2.6 Example – proofs

The category ***PROOF*** has sentences as objects, and morphisms are proofs, i.e., a morphism $f:A\square B$ is a specific proof of B from A. The identity morphisms are the trivial proofs of sentences from themselves (empty proofs). Proof composition corresponds to sequence concatenation – the cut rule.

2.2.7 Example – inheritance hierarchies

In Eiffel, the relationship "ancestor" is defined as the "reflexive and transitive closure" of the inheritance hierarchy: class A is an ancestor of class B iff A is B itself or A is an ancestor of a parent of B (i.e. of a class from which B inherits) [88]. Given an inheritance graph G between classes, e.g. example 2.1.3, the category ***ancestor(G)*** is generated from the graph by completing it with the arrows that result from reflexivity (identities) and transitivity (compositions).



2.2.8 Proposition – category generated from a graph

Every graph G generates a category ***cat***(G)[7] whose objects are the nodes and whose morphisms are the paths of the graph. Identities are empty paths. Composition is concatenation of paths.

2.2.9 Example – runs

Every state transition system generates the category of its runs defined to be the paths of the underlying graph.

Category Theory supports and encourages forms of diagrammatic reason, also called "diagram chasing", a practice that is consistent with the modern culture in Computing. Contrarily to what often happens in Software Engineering, the notion of diagram is formal and the reasoning that can be done with diagrams is mathematical.

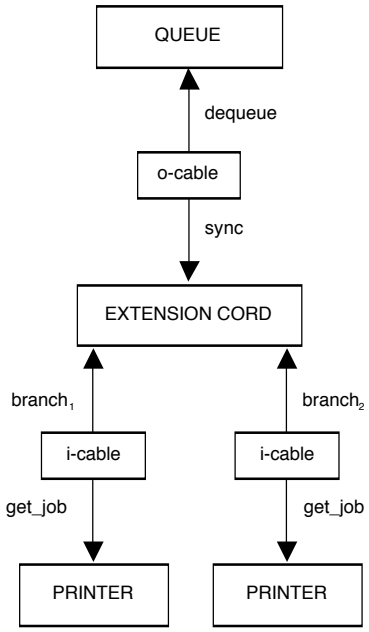
2.2.10 Definition – diagram

Let $\mathbf{C}$ be a category and I a graph. A *diagram* in $\mathbf{C}$ with *shape* I is a graph homomorphism $D: I \to \mathbf{graph}(\mathbf{C})$. Given a node x (resp. arrow u) of I, we normally denote its image $D_0(x)$ (resp. $D_1(u)$) by D_x, (resp. D_u).

The fact that a diagram is being defined as a graph homomorphismand not a graph may surprise some software engineers. This is because there is sometimes a confusion between the "form" and the "contents" of a diagram. The homomorphism defines a labelling of the graph I, which defines the "shape" or "form" of the diagram. That is, a diagram in a category can be seen as a graph whose nodes are labelled with objects and the arrows are labelled with morphisms that respect sources and targets. In particular, the labelling does not need to

[7] The observant reader may have noted that we have already departed from our convention of using only upper case characters for the names of categories! This is not a random aberration, but reflects the fact that generating a category from a graph is a map from one category to another. Such maps, called functors, will be introduced in Chapter 5 and denoted using bold, lower case characters.

be injective, meaning that different nodes may be labelled with the same object. An example can be given in terms of "configuration diagrams" such as those that will be used in chapter 8. In such diagrams, the graph identifies system components and interconnections between them, and the homomorphism their types. For instance, a configuration of a queue serving two independent printers can be given by the diagram:



The fact that we are using two printers of the same type, and not only one, results from having two nodes labelled *PRINTER*. The same applies to the cables: each printer is connected to the extension cord by its own cable, but they are of the same type; so we have two nodes labelled with *i-cable*; these cables connect to the same "printer port", so the two arrows to *PRINTER* are labelled with the same morphism *get_job*; however, the cables connect to different ports of *EXTENSION CORD*, hence the corresponding morphisms are different.

2.2.11 Definition –commutative diagram

Let $\boldsymbol{C}$ be a category and I a graph. A diagram $\Box : I \Box \boldsymbol{graph(C)}$ is said to commute iff, for every pair x,y of nodes and every pair of paths $w=u_1...u_m$, $w'=v_1...v_n$ from x to y in graph I,

$$\Box_{u_1};...;\Box_{u_m}=\Box_{v_1};...;\Box_{v_n}$$

holds in $\boldsymbol{C}$.

The property that a diagram commutes establishes a set of equalities between arrows. Hence, diagrams and commutativity provide us with the ability of doing equational reasoning in a "visual" form, an advantage that has not been fully exploited yet in Software Engineering. See [48] for a more developed use of these possibilities for mathematical reasoning. To indicate that a diagram commutes, we will decorate it with the symbol ◎.

We close this section with an example that provides a good illustration of how morphisms "preserve structure" and how diagram-chasing facilitates reasoning about categories.

2.2.12 Example – automata

A (deterministic) automaton consists of an input set X, a state set S, an output set Y, a transition function $f: X \times S \to S$, an initial state $s_0 \in S$, and an output function $g: S \to Y$. A notion of morphism of automata must be able to capture this structure by indicating how a given automaton can be simulated by another. Namely, it must translate the input, state and output sets from one automaton to the other in such a way that the transition functions, the initial states and the output functions of the two automata "agree".

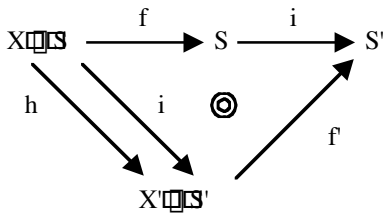
More concretely, we should require of a morphism $A \to A'$ that:

1. the initial state of A be mapped to the initial state of A'
2. if we perform a transition in A and map the resulting state into A', we get the state that is obtained by first translating the input and initial state into A' and then applying the transition function of A'
3. the translation into A' of the output of a state of A consists of the output in A' of the translation into A' of the original state of A.

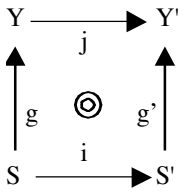
Hence, a morphism from $A=\langle X,S,Y,s_0,f,g\rangle$ to $A'=\langle X',S',Y',s'_0,f',g'\rangle$ is defined to consist of three functions $\langle h:X \to X', i:S \to S', j:Y \to Y'\rangle$ such that

1. $i(s_0)=s'_0$
2. $f;i=h \times i;f'$
3. $g;j=i;g'$

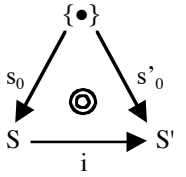
Notice that equations 2 and 3 can be expressed as requests for certain diagrams to commute. For equation 2,



For equation 3,



What about the first equation? Can it be expressed as a commutative diagram as well? The answer seems to be less obvious or immediate because the equation is very "set-theoretical": it is an equality between elements (initial states), not functions… However, we have already mentioned in the introduction that an element $s_0 \in S$ can be identified with a morphism $s_0:\{\bullet\} \to S$ from an arbitrary singleton set – an observation that we shall formalise in section 4.1, but which can remain at an intuitive level for the purpose of this discussion. Given this, the first equation can be rewritten as $s_0;i=s'_0$ and expressed via:



We should make clear that the choice between using diagrams or text should be made according to the intended readership or usage. The point that we want to make here is that it is *possible* to use diagrams to express and reason about properties in a formal way, and that there is some *unified* or *universal* side to this pratice; however, we shall abstain from giving recommendations as to when and for whom diagrammatic notations should be used.

The composition law of automata is defined to be that of functions between sets, i.e. composition applies internally to each of the three components of the morphisms. Likewise, the identity for an automaton *A* is defined to be the triple that consists of the three identity functions over the sets of input, states and outputs.

The proof that a category – ***AUTOM*** – is indeed obtained in this way is very revealing of a procedure that will be systematised in section 3.2. Because the associativity of the composition law and the properties of the identity are automatically inherited from the corresponding properties of functions, all that needs to be proved is that (1) the composition of two morphisms is, indeed, a morphism (i.e. it satisfies the three equations above) and (2) the identity is, indeed, a morphism (satisfies the same three equations)! We prove just the case of the composition

$$<h,i,j>;<h',i',j'>=<h;h',i;i',j;j'>$$

of morphisms

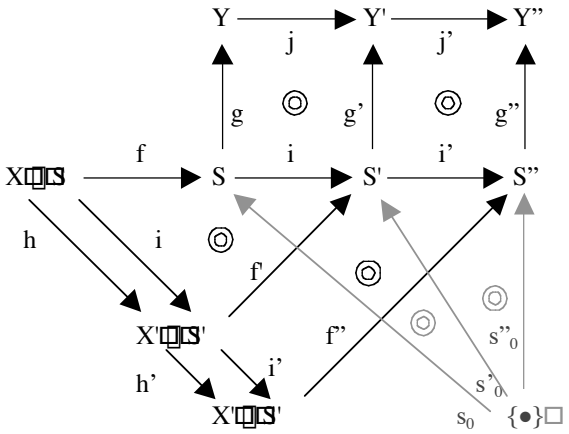
$$<X,S,Y,s_0,f,g> \to <X',S',Y',s'_0,f',g'>$$

and

$$<X',S',Y',s'_0,f',g'> \to <X'',S'',Y'',s''_0,f'',g''>$$

1. $(i;i')(s_0)=i'(i(s_0))=i'(s'_0)=s''_0$
2. $f;(i;i')=(f;i);i'$
   $=(h \Box i;f');i'$
   $=h \Box i;(f';i')$
   $=h \Box i;(h' \Box i';f'')$
   $=(h \Box i;h' \Box i');f''$
   $=(h;h') \Box (i;i');f''$
3. $g;(j;j')=(g;j);j'=(i;g');j'=i;(g';j')=i;(i';g'')=(i;i');g''$

This proof is best understood in terms of the following commutative diagram(s):



2.3 Distinguished kinds of morphisms

There are several classes of morphisms that have special properties worth knowing about because they allow us to recognise situations in which standard results or constructions apply.

2.3.1 Definition/Proposition – isomorphism

Let $\boldsymbol{C}$ be a category and x,y objects of $\boldsymbol{C}$. A morphism $f:x \to y$ of $\boldsymbol{C}$ is said to be an *isomorphism* iff there is a morphism $g:y \to x$ of $\boldsymbol{C}$ such that: $f;g=id_x$ and $g;f=id_y$. In these conditions, x and y are said to be isomorphic.

2.3.2 Example

1. In ***SET***, a morphism is an isomorphism iff it is bijective.
2. In ***LOGI***, two formulae are isomorphic iff they are logically equivalent.

2.3.3 Exercise

What about isomorphic objects in ***PROOF***?

2.3.4 Definition/Proposition – inverse

Given an isomorphism $f:x \to y$ there is a unique morphism $g:y \to x$ such that: $f;g=id_x$ and $g;f=id_y$. This morphism is called the *inverse* of f.

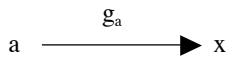
proof

The morphism g mentioned in the definition above is unique: given $h:y \to x$ in the same circumstances, we have

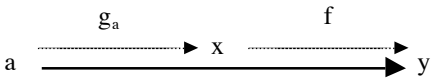
h

$= h;id_x$ □	property of the identity
$= h;(f;g)$ □	because $f;g=id_x$
$= (h;f);g$ □	associativity
$= id_y;g$ □	because $h;f=id_y$
$= g$ □	property of the identity

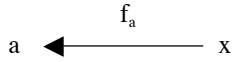
Any two objects x and y related by an isomorphism $f:x \to y$ have a very important property: the class of morphisms from x (resp. into x) is in one-to-one correspondence with the class of morphisms from y (resp. into y). This one-to-one correspondence is established by composing any morphism $f_a:x \to a$ (resp.$g_a:a \to x$) with the inverse of f (resp. f itself). Because morphisms characterise the interactions that objects can hold with other objects (their "social lives"), what this property says is that isomorphic objects interact in essentially the same way. That is to say, isomorphic objects cannot be distinguished by interacting with them. In any context where an object is used, it can be replaced by an isomorphic one by using the isomorphism and its inverse to re-establish the interconnections: any incoming arrow



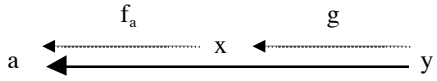
is replaced by



and any outgoing arrow



is replaced by



Hence, it is usual to treat any two isomorphic objects as being "essentially" the same.

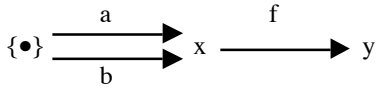
Again, we should point out that this property holds in so far the structure ("social life") that is captured by the category is concerned. For a different category over the same objects, revealing other structural aspects, the isomorphism may not carry through. For instance, in object-oriented software development, two objects may be isomorphic in so far as having the same interfaces and exhibiting the same behaviour at their interfaces, but may have completely different implementations.

There are other classes of morphisms that are important to mention, namely those that generalise well-known properties in set-theory like injective and surjective functions. The way this generalisation is made reveals a lot of the way Category Theory operates and how it relates to and differs from Set Theory. For instance, the typical characterisation of an injective function $f:x \to y$ in Set Theory is:

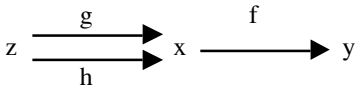
for every $a,b \in x$, $f(a)=f(b)$ implies $a=b$.

As expected, this characterisation uses set-membership. As motivated already, in Category Theory we have to replace it by interactions (morphisms). We have already mentioned that elements $a \in x$ of sets can be identified with morphisms $a:\{\bullet\} \to x$ from singleton sets. Hence, the set-theoretic definition amounts to saying that,

for any (total) functions $a,b:\{\bullet\} \to x$, $a;f=b;f$ implies $a=b$.

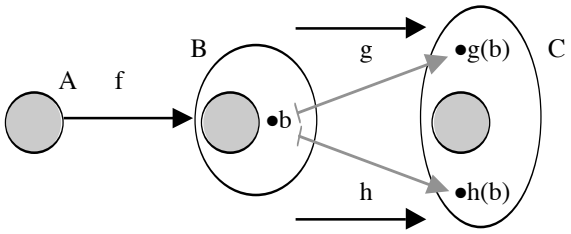


When considered in this way, an injective function can be characterised as not interfering with independent observations that are made on the source. The categorical characterisation consists precisely in generalising this definition to arbitrary morphisms instead of just elements: for every pair of morphisms $g,h:z \to x$, $g;f=h;f$ implies $g=h$.



What is even more interesting is the fact that the generalisation carries through, in a dual way that we will formalise in the next chapter, to the characterisation of surjective functions. Surjectivity is all about inter-

ference with the social life of the target of the arrow. If we take a non-surjective function $f:A \to B$ and $b \notin f(A)$, then we can have two functions $g,h:B \to C$ that coincide in $f(A)$, i.e. such that $f;g=f;h$, and yet are different because they disagree on the way they map b.



2.3.5 Definition – mono and epimorphisms

Consider an arbitrary category $\mathbf{C}$ and morphism $f:x \to y$ in $\mathbf{C}$.

1. f is said to be a *monomorphism*, or a mono, or monic, iff, for every pair of morphisms $g,h:z \to x$, $g;f=h;f$ implies $g=h$.
2. f is said to be an *epimorphism*, or an epi, or epic, iff, for every pair of morphisms $g,h:y \to z$, $f;g=f;h$ implies $g=h$.

Monos and epis satisfy many of the properties that we know from injective and surjective functions. They are left here as exercises.

2.3.6 Exercises

Consider an arbitrary category $\mathbf{C}$.

1. Prove that isomorphisms are both epic and monic.
2. Prove that the composition of monos (resp. epis) is also a mono (resp. epi).
3. Prove that if $f;g$ is monic (resp. epic), then so is f (resp. g).

A final word of caution though. The analogy with Set-Theory cannot be carried too far: the converse of the property referred to in the first exercise does not hold for arbitrary categories! That is, not all morphisms that are both epic and monic are isomorphisms. This is because, to be an isomorphism, an arrow needs to have a left and a right inverse which, contrarily to what happens in Set Theory, is not guaranteed simply by being monic and epic.

2.3.7 Definition – split mono and epimorphisms

Consider an arbitrary category $\mathbf{C}$ and morphism $f:x \to y$ in $\mathbf{C}$.

1. f is said to be a *split* monomorphism, or a split mono, or split monic, iff it admits a right inverse, i.e. iff there is $g:y \to x$ such that $f;g=id_x$.
2. f is said to be a split epimorphism, or a split epi, or split epic, iff it admits a left inverse, i.e. iff there is $g:y \to x$ such that $g;f=id_y$.

2.3.8 Exercises

Consider an arbitrary category $\boldsymbol{C}$.

1. Prove that every split mono (resp. split epi) is monic (resp. epic).
2. Prove that every morphism that is a split mono and a split epi is an isomorphism. In fact, prove that only one, either the mono or the epi, needs to be "split".

3 Building categories

There are many ways in which new categories can be built from existing ones. One of the advantages of building new categories by "inheriting" from old ones is that it is easier to prove that the construction yields, indeed, a category. In this chapter, we will present some elementary constructions that illustrate the point, together with examples of categories that relate to software development. In later chapters, will study some more elaborate constructions that we have found useful for our day-to-day.

3.1 Some elementary operations

There are a number of elementary operations for constructing categories. The first example reflects the fact that the choice for the direction of the morphisms has no essential significance in the sense that, if we reverse the direction of all the morphisms, we obtain a category that has the same structural properties of the original one.

This fact does not mean, however, that the direction of the morphisms should not be carefully chosen when defining a category. It is often the case that there is a "natural" direction for morphisms, either given by the mathematical nature of the morphisms or their use in practice. If reversed, the direction of the morphisms may make it more difficult or less immediate to understand the intended meaning and the particular constructions one may want to define on objects and morphisms. Hence, if you are going to become a “practising category theorist”, be prepared to answer questions like “but aren’t the arrows going the wrong way?” which, many times, just means “you do not seem to belong to my club!”…

3.1.1 Definition – dual of a category

For any category $\mathbf{C}$ we can construct its *dual* or *opposite* $\mathbf{C}^{op}$:

- $\mathbf{C}^{op}$ has the same objects and arrows as $\mathbf{C}$
- arrows of $\mathbf{C}^{op}$ go in the reverse direction: if $f:A \to B$ in $\mathbf{C}$ then $f:B \to A$ in $\mathbf{C}^{op}$
- and compose in the reverse direction: $f;g$ in $\mathbf{C}^{op}$ is $g;f$ in $\mathbf{C}$.

3.1.2 Example

Following [88], the dual of the category ***ancestor****(G)* can be named ***descendant****(G)*.

The fact that every category and its opposite have the same structural properties reflects a general *duality principle* that applies to all definitions and results. Every concept in Category Theory has a dual, which is the concept that is obtained by reversing the direction of the arrows, i.e. the concept that holds in the dual category. Every result in Category Theory has a dual which holds for the dual concepts. We shall have plenty of occasions to illustrate how the duality principle is applied in practice.

Another elementary construction consists in taking the product of two categories:

3.1.3 Definition/Proposition – product category

Given categories $\mathbf{C}$ and $\mathbf{D}$, their *product* $\mathbf{C}\times\mathbf{D}$ is such that its objects are the pairs $\langle c,d\rangle$ where $c:\mathbf{C}$ and $d:\mathbf{D}$, and the morphisms $\langle c,d\rangle \to \langle c',d'\rangle$ are all the pairs $\langle f,g\rangle$ where $f:c \to c'$ in $\mathbf{C}$ and $g:d \to d'$ in $\mathbf{D}$. Composition of morphisms is defined componentwise, and the identity for an object $\langle c,d\rangle$ is the pair $\langle id_C, id_D\rangle$.

proof

The properties of composition and the identities are trivially inherited from the corresponding properties of the components.

There are also ways of extracting categories from other mathematical structures. For instance, we mentioned in 2.2.5 that every pre-order defines a category. Yet a simpler construction is the one that views sets as categories:

3.1.4 Definition/Proposition – discrete categories

Every set S determines a category whose objects are the elements of S and whose morphisms are just the identities, i.e. for every $s \in S$ $hom(s,s)$ is a singleton (consisting of the identity for s), and $hom(s,s')$ is empty for every $s,s' \in S$ such that $s \neq s'$. Such categories are said to be *discrete*.

proof

Immediate. Note that, if we take S to be the set of nodes of a graph with no arrows, then this construction is a special case of 2.2.8.

The discrete category defined by a set may have many objects, as many as the elements of the set, but has only the minimum number of morphisms – the identities. The next example illustrates the other extreme: as many morphisms as we want, but only one object... These categories correspond to monoids:

3.1.5 Definition/Proposition – monoid as a category

A monoid is a triple $<M,*,1>$ where M is a set, * is an associative binary operation on M, and 1 is an identity for *, i.e. $m*1=1*m=m$ for every $m \in M$. Any monoid defines a category that consists of only one object, which we can denote by ., and whose morphisms $. \to .$ are the elements of M. Composition of morphisms is given by the operation * and the identity morphism is 1.

proof

The properties of composition and the identity are trivially inherited from the corresponding properties of the monoid.

3.1.6 Exercise

Characterise the duals of the categories defined by monoids. Are these categories their own duals?

3.2 "Adding structure"

The most typical way of building a new category is, perhaps, by adding "structure" to the objects of a given category (or a subset thereof). The expression "adding structure" has, of course, a broad meaning that the reader will only fully apprehend after building a few categories... The morphisms of the new category are then the morphisms of the old category that "preserve" the additional structure.

The following example is as typical as any other and will be used throughout the book for applications related to systems modelling.

3.2.1 Example – pointed sets

The category $\boldsymbol{SET}_{\bullet}$ (of pointed sets) is defined as follows. Its objects are the pairs $<A,\bullet_A>$ where A is a set and $\bullet_A$ is an element of A called the designated element. The morphisms between two pointed sets $<A,\bullet_A>$ and $<B,\bullet_B>$ are the total functions $f:A \to B$ such that $f(\bullet_A)=\bullet_B$.

The category over which $\boldsymbol{SET}_{\bullet}$ is being built is, obviously, ***SET***, the category of sets and total functions. The additional structure that is being added to the objects of the base category, sets, is the designation of an element (not the element itself because it is already in the set). The morphisms of the new category are all the morphisms of the base category that preserve the additional structure, which in this case means mapping the designated element of the source to the designated element of the target.

This way of building new categories makes it easier to conduct proofs as illustrated next.

proof

The attentive reader will have noticed that the definition of $\mathbf{SET}_{\bullet}$ omitted two fundamental ingredients: the definition of the composition law and of the identity map. This is because they are, by default, inherited from the base category. Because the laws of composition and identity are satisfied in the base category, they do not need to be checked again for the new category. Hence, all that needs to be checked is that (1) the composition law is closed for $\mathbf{SET}_{\bullet}$, i.e. that the composition of two functions that are morphisms in $\mathbf{SET}_{\bullet}$ is also a morphism of $\mathbf{SET}_{\bullet}$, and (2) that the identities are, indeed, morphisms in $\mathbf{SET}_{\bullet}$:

1. Given morphisms $f:\langle A,\bullet_A\rangle \to \langle B,\bullet_B\rangle$ and $g:\langle B,\bullet_B\rangle \to \langle C,\bullet_C\rangle$, we have to check that $(f;g)(\bullet_A)=\bullet_C$. But $f(\bullet_A)=\bullet_B$ because f is a morphism of pointed sets. Hence, $(f;g)(\bullet_A)=g(f(\bullet_A))=g(\bullet_B)$. On the other hand, because g is also a morphism of pointed sets, $g(\bullet_B)=\bullet_C$.
2. Given a pointed set $\langle A,\bullet_A\rangle$, we now have to prove that the identity map id_A for A as a set is a morphism of pointed sets, i.e. is such that $id_A(\bullet_A)=\bullet_A$. But this is true because id_A is precisely the identity function on A.

Notice the similarities between the structure of this proof and the proof outline given for ***AUTOM*** (2.2.12). This style of proof is typical of categories built from other categories by "adding structure".

The particular use of pointed sets that we have in mind is for modelling the behaviour of objects at their interfaces (methods). A very abstract and general model of object behaviour is one in which the events that can potentially occur during the lifetime of an object consist of all possible subsets of the set of its methods. This means that we work in a model in which every object can potentially handle concurrent method calls. We shall see later on how we can model restrictions to the degree of concurrency that an object can impose on its methods. The empty set represents an event in which the object is not involved. This explicit, but abstract, representation of environment events has been shown to be important for modelling the behaviour of concurrent and distributed systems [13].

The fact that the events that we are using for modelling object behaviour consist of sets of method calls can be abstracted away to recognise a more general notion of process alphabet as used in Concurrency. The application of categorical techniques in Concurrency Theory is particularly rich and revealing of the power of Category Theory for systematising constructions and establishing relationships between different models. The application to object behaviour that we will use for illustration purposes throughout the book touches some of these aspects but the reader interested in a more complete picture of the breadth of the field should consult [21,24,99,110]. Specific applications of these categorical techniques to object-oriented modelling can be found in [21,24].

In this more general model of process behaviour, a process alphabet is a pointed set. Each element of the set represents an event whose oc-

currence may be witnessed during the lifetime of the process. The designated element of the set represents an event of the environment, i.e. an event in which the process is not involved. The notion of morphism in this model captures the relationship that exists between systems and components of systems. More precisely, every morphism identifies the way in which the target is embedded, as a component, in the source. The fact that the morphism is a function between the alphabets is pretty intuitive: for every event a occurring in the lifetime of the system, we have to know how the component participates in that event; if the component is not involved in a, then the morphism should map a to the designated element of the alphabet of the component, i.e. to the element that has been designated to represent the events in which the component does not participate. On the other hand, any event occurring in the environment without the participation of the system cannot involve the component either; hence, every morphism needs to preserve the designated element.

For instance, consider a producer-consumer system. We can assign the events *produce* and *store* to the producer and *consume* and *retrieve* to the consumer. An event of the system during which both *produce* and *consume* take place concurrently is mapped to *produce* by the morphism *prod: system→ producer* that identifies the producer as a component of the system. On the other hand, an event of the system during which the consumer executes *retrieve* and the producer remains idle is mapped by *prod* to the designated element of the alphabet of the producer.

Our last example of constructing new categories from base ones is also very typical. It provides so-called *comma-categories*. Several different notations can be found in the literature for comma-categories. We shall use the same notation as [79]. The particular form of comma-categories that we are about to define are also called over/under-cone categories in [22] and (co)slice-categories in [48].

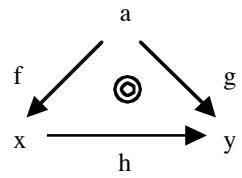
3.2.2 Example – comma-categories

Given a category $\mathbf{C}$ and an object $a:\mathbf{C}$, we define the category of objects under a – $a\downarrow\mathbf{C}$ – as follows. Its objects are all the pairs $\langle f,x\rangle$ where f is a morphism of the form $f:a\rightarrow x$ in $\mathbf{C}$. The morphisms between $f:a\rightarrow x$ and $g:a\rightarrow y$ are all the $\mathbf{C}$-morphisms $h:x\rightarrow y$ such that $f;h=g$.

proof:

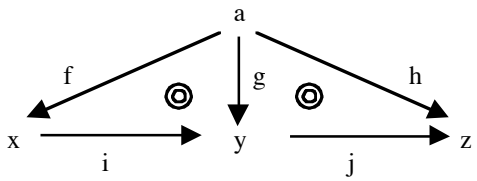
In this case, the additional structure that is being added to every object $x:\mathbf{C}$ is a $\mathbf{C}$-morphism $f:a\rightarrow x$. Hence, again, we assume that the composition law and the identity map are inherited from the base category and prove only the closure properties. This is also a good example to illustrate how reasoning in Category Theory is often done at a graphical level. Because our objects now are morphisms of

the base category, it is more convenient to represent a morphism h between $f:a \to x$ and $g:a \to y$ by the diagram:



As already mentioned, the symbol ◎ indicates that the diagram commutes, i.e. all compositions of morphisms in the diagram along paths that have the same source and target are equal. In the case above, commutativity of the diagram expresses the equation $f;h=g$.

Given morphisms i and j



we have to prove that the composition $(i;j)$ in $\mathbf{C}$ is a morphism in $a \downarrow \mathbf{C}$, i.e. $f;(i;j)=h$. There is another way of asserting the property that needs to be proved which makes appeal to the kind of "diagrammatic reasoning" that we have claimed to be typical of Category Theory: we have to prove that, for any diagram of the form above, the outer triangle commutes if the inner triangles also commute. This can be done as follows:

1. $f;i=g$ because i is a morphism in $a \downarrow \mathbf{C}$
2. $(f;i);j=g;j$ from (1)
3. $g;j=h$ because j is a morphism in $a \downarrow \mathbf{C}$
4. $(f;i);j=h$ from (2) and (3)
5. $f;(i;j)=(f;i);j$ from the associativity of the composition law in $\mathbf{C}$
6. $f;(i;j)=h$ from (4) and (5)

The fact that the morphism id_x is an identity for $f:a \to x$ results from the property $f;id_x=id_x$.

For instance, when applied to a category of the form ***ancestor***(G), this construction returns, for every class c, the classes that inherit from c, themselves organised as a hierarchy. When applied to the category ***PROOF*** it returns, for every sentence, the inferences that can be made from that sentence.

3.2.3 Exercise

Characterise the dual category of $a \downarrow \mathbf{C}$ – the category of objects over a, usually denoted by $\mathbf{C} \downarrow a$

Generalisations of the notion of comma category will be presented in section 6.1.

Together with example 2.2.12, we hope that we have provided enough insight into the way categories are often defined: by "adding structure" to the objects of another category. Because there is no abstract characterisation of this technique, we have relied on the examples to make apparent that there is a systematic procedure for checking the definition of a category built by "adding structure".

3.3 Subcategories

There is another intuitive, and often useful, way of building new categories from old: by forgetting some of the objects and some of the morphisms to create a *subcategory* of the original one. The only proof burden associated with this method is in making sure that we do not through away too much (or too little...).

3.3.1 Definition – subcategory

Given categories $\boldsymbol{C}=<G_{\mathbf{C}}, ;_{\mathbf{C}}, id_{\mathbf{C}}>$ and $\boldsymbol{D}=<G_{\mathbf{D}}, ;_{\mathbf{D}}, id_{\mathbf{D}}>$, we say that $\boldsymbol{D}$ is a *subcategory* of $\boldsymbol{C}$ iff

- $|\boldsymbol{D}| \subseteq |\boldsymbol{C}|$, i.e. every object of $\boldsymbol{D}$ is an object of $\boldsymbol{C}$
- For any objects x and y of $\boldsymbol{D}$, $hom_{\boldsymbol{D}}(x,y) \subseteq hom_{\boldsymbol{C}}(x,y)$, i.e. the morphisms in $\boldsymbol{D}$ are also morphisms in $\boldsymbol{C}$
- The composition laws and the identity maps in the two categories agree, i.e. given composable morphisms f and g of $\boldsymbol{D}$, their composition $f;_{\mathbf{D}}g$ in $\boldsymbol{D}$ is the same as their composition $f;_{\mathbf{C}}g$ in $\boldsymbol{C}$, and for every object x of $\boldsymbol{D}$, its identity $id_{\mathbf{D}}(x)$ in $\boldsymbol{D}$ is the same as $id_{\mathbf{C}}(x)$ in $\boldsymbol{C}$.

Notice that, for $\boldsymbol{D}$ to be a subcategory of $\boldsymbol{C}$, it is not enough to have inclusions between the sets of objects and of morphisms: the additional structure given by the identities and the composition law has to be preserved.

Examples of subcategories will be given as we go along, most of the time as exercises.

3.3.2 Examples

1. By keeping just the sets that are finite and all the total functions between them, we define a subcategory ***fSET*** of ***SET***.
2. By keeping all sets but just the functions that are injective, we define a subcategory ***INJ*** of ***SET***. This is because identities are injective and the composition of injective functions is still injective.
3. Given any category, we can forget all its morphisms except for the identities, and obtain a (discrete) subcategory.

4. We can extend the morphisms of ***SET*** to include all partial functions between sets. Because the identity function is trivially partial and partial functions compose in the same way as total functions, we obtain a category of which ***SET*** is a subcategory. The category of partial functions will be called ***PAR***.
5. When discussing pointed sets, we have motivated the notion of alphabet of object behaviour by identifying events with sets of methods. That is, we have worked with pointed sets that consist of powersets, the empty set being the designated element of each powerset. We can show that we define a subcategory of $\mathbf{SET}_{\square}$ by choosing as morphisms $2^B \to 2^A$ maps that compute inverse images for functions $A \to B$, i.e. a morphism $g:2^B \to 2^A$ is such that, for some $f:A \to B$, $g(B')=f^{-1}(B')$ for every $B' \subseteq B$. We call this category ***POWER***.

The cases in which we throw away some of the objects but keep all the morphisms between those that remain, like we did for finite sets, deserve a special designation:

3.3.3 Definition – full subcategory

Given categories ***C*** and ***D*** we say that ***D*** is a *full* subcategory of ***C*** iff ***D*** is a subcategory of ***C*** and, for any objects x and y of ***D***, $hom_D(x,y)=hom_C(x,y)$.

3.3.4 Exercise

Check which of the subcategories defined in 3.3.2 are full.

We have mentioned several times that, intensionally, a category captures a certain notion of structure, what we have called a "social life" for its objects. It is only natural to expect that, when selecting a subcategory, we obtain a different but, nevertheless related, notion of structure. For instance, if ***D*** is a subcategory of ***C***, two objects that are isomorphic in ***C*** are not necessarily isomorphic in ***D***. A trivial illustration of this fact can be obtained by realising that the discrete subcategory of ***C*** that is obtained by forgetting all the morphisms except the identities (see 3.3.2) is such that any object is only isomorphic to itself (it has no social life...). Indeed, the more structure a category provides through its morphisms, the more observational power we have over its objects and, hence, the more isomorphisms we are able to establish. The reverse property, however, holds and is left as an exercise.

3.3.5 Exercise

Let ***D*** be a subcategory of ***C***. Show that any two objects that are isomorphic in ***D*** are necessarily isomorphic in ***C***.

The relationship between isomorphisms and subcategories is, in fact, very revealing of the way categories can be used to capture notions of

structure. Recall that we classified a subcategory as being full iff it is obtained by selecting a subclass of the objects but leaving the morphisms between the selected objects unchanged. Typically, this happens when one wants to select just the objects that satisfy a certain property, e.g. sets that are finite as in example 3.3.2. In this case, the converse of the property proved in the exercise above also holds because, by not interfering with the morphisms, we are keeping the structure of the objects:

3.3.6 Exercise

Let ***D*** be a full subcategory of ***C***. Show that any two ***D***-objects that are isomorphic in ***C*** are also isomorphic in ***D***.

In the cases where, like finiteness of sets, the discriminating property does not distinguish between isomorphic objects, we obtain a subcategory that is not only full but contains all objects that are isomorphic, i.e. share the same substructure.

3.3.7 Definition – isomorphism-closed full subcategory

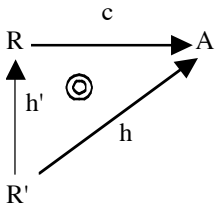
A full subcategory ***D*** of a category ***C*** is said to be *isomorphism-closed* iff every ***C***-object that is isomorphic to a ***D***-object is also a ***D***-object.

So far, we have looked at the formation of a subcategory as the result of a selection of special objects and morphisms from a given category. In certain circumstances, this selection is, actually, the result of a process of "abstraction", i.e. certain objects are selected because they are "canonical" with respect to a set of objects that share a given property.

As an example, consider automata as defined in 2.2.12. Intuitively, only the states that are reachable, in the sense that they can be reached through the transition function from the initial state and some input sequence, are important for determining the "social life" of automata in terms of their ability to simulate other automata. Consider, then, the full subcategory ***REACH*** of ***AUTOM*** that consists of all reachable automata. The idea that, for the purposes of simulating other automata, every automaton can be represented by the automaton that is obtained by removing all non-reachable states can be formalised as follows:

- every automaton A is related to a "canonical" reachable automaton R through a morphism $c:R \to A$. More concretely, if $A=\langle X,S,Y,s_0,f,g\rangle$, then $R=\langle X,S_R,Y,s_0,f_R,g_R\rangle$ where S_R is the subset of S consisting of the states that are reachable, and f_R (resp. g_R) is the restriction of f (resp. g) to S_R. Notice that R is well-defined because f returns reachable states when applied to reachable states. The morphism c consists of the inclusion of S_R into S and the identities on X and Y. Notice that the inclusion satisfies the three properties of simulations in a trivial way.

- every inclusion $c:R \to A$ as defined above satisfies the following property: given any reachable automata R' and simulation $h:R' \to A$, there is a unique morphism of reachable automata $h':R' \to R$ such that $h=h';c$.



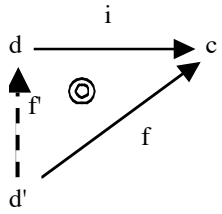
- In other words, R is the reachable automaton that is "closest" to A in the sense that any other reachable automata that A can simulate can also be simulated by R. Indeed, if $R'=<X',S',Y',s'_0,f',g'>$ and $h=<h_X,h_S,h_Y>$, then the equation $h=h';c$ can only be satisfied if $h'_S:S' \to S_R$ coincides with h_S on the whole domain S', i.e. if h_S only returns reachable states. But this is the case because R' is itself reachable. On the other hand, the equation fully determines the morphism, thus establishing the uniqueness of h'.

This relationship between automata and reachable automata is an instance of what, in Category Theory, is called a co-reflective subcategory:

3.3.8 Definition – co-reflective subcategory

Let $\boldsymbol{D}$ be a subcategory of a category $\boldsymbol{C}$.

1. Let c be a $\boldsymbol{C}$-object. A $\boldsymbol{D}$-co-reflection for c is a $\boldsymbol{C}$-morphism $i:d \to c$ for some $\boldsymbol{D}$-object d such that, for any $\boldsymbol{C}$-morphism $f:d' \to c$ where d' is a $\boldsymbol{D}$-object, there is a unique $\boldsymbol{D}$-morphism $f':d' \to d$ such that $f=f';i$, i.e. the following diagram commutes:



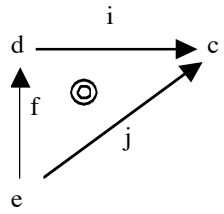
2. The category $\boldsymbol{D}$ is said to be a *co-reflective subcategory* of $\boldsymbol{C}$ iff every $\boldsymbol{C}$-object admits a $\boldsymbol{D}$-co-reflection.

Co-reflections are essentially "unique" in the following sense:

3.3.9 Proposition

Let $\boldsymbol{D}$ be a subcategory of a category $\boldsymbol{C}$.

1. Let c be a $\boldsymbol{C}$-object. If $i:d \to c$ and $j:e \to c$ are both $\boldsymbol{D}$-co-reflections for c, then there is a $\boldsymbol{D}$-isomorphism $f:e \to d$ such that $j=f;i$, i.e. the following diagram commutes:

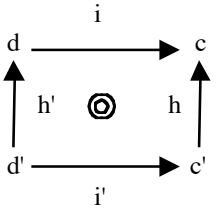


2. Let c be a $\boldsymbol{C}$-object. If $i:d \to c$ is a $\boldsymbol{D}$-co-reflection for c and $f:e \to d$ is a $\boldsymbol{D}$-isomorphism then $f;i:e \to c$ is also a $\boldsymbol{D}$-co-reflection for c.

proof

1. The existence of f satisfying the equation results directly from the definition. The fact that f is an isomorphism can be proved as follows:
   - the existence of $g:d \to e$ such that $i=g;j$ can be concluded for the same reasons;
   - the composition $g;f$ is such that $(g;f);i=g;(f;i)=g;j=i=id_d;i$. We can conclude that $g;f=id_d$ by applying the uniqueness requirement of the definition of co-reflection to id_d.
   - a similar line of reasoning will allow us to conclude that $f;g=id_e$.
2. This is trivially proved.

Intuitively, a co-reflector for an object is a "secretary" that manages its social life, i.e. through which all communication must go. Indeed, consider two objects c and c' together with their secretaries d and d', respectively. Consider an interaction $h:c \to c'$.



The composition $i';h$ is in the same circumstances as the morphism f in the definition. Hence, we conclude that there is an interaction $h':d' \to d$ that factorises (intercepts) h in the sense that $i';h=h';i$.

Given that "secretaries" are also objects of the bigger category, a question arises naturally: who are their secretaries? Do they talk to each other directly or do they need their own secretaries and so on? It seems intuitive to expect that all secretaries communicate directly between them, i.e. that the co-reflection of a $\boldsymbol{D}$-object (secretary) is the identity (itself) up to isomorphism. However, this can only be guaranteed if (and only if) $\boldsymbol{D}$ is a full subcategory:

3.3.10 Proposition

Let ***D*** be a co-reflective subcategory of a category ***C***. Then, ***D*** is a full subcategory of ***C*** iff, for every ***D***-object d, id_d is a ***D***-co-reflection for d.

proof

1. Let ***D*** be a full subcategory of ***C*** and d a ***D***-object. Given any ***C***-morphism $f:d' \to d$ where d' is also a ***D***-object, the equation $f=f';id_d$ establishes $f=f'$ uniquely. So, the only way for the identity id_d not to be a co-reflector for d is that f is not a ***D***-morphism as required by the definition. Naturally, if the category is full, this cannot happen.
2. Consider now an arbitrary ***C***-morphism $f:d' \to d$ where both d and d' are ***D***-objects. If id_d is a ***D***-co-reflection for d then, by definition, there exists a ***D***-morphism $f:d' \to d$ such that $f=f';id_d$, i.e. $f=f'$. Hence, f is also a ***D***-morphism.

Indeed, if the subcategory is not full, the secretaries, when playing their roles as secretaries, may have more restricted means of interaction and, therefore, may not be able to interact as they would do as "normal" objects.

The "social motivation" that we have been using is, in fact, somewhat biased towards objects, i.e. it leads us to consider co-reflective subcategories that are full like that of reachable automata. However, any categorical property is, ultimately, determined by the morphisms. The following example shows that there are co-reflective subcategories that are not full.

3.3.11 Example

We mentioned in 3.3.2 that ***SET*** is a subcategory of ***PAR***, the category of partial functions. Both categories share the same objects (sets), but ***SET*** retains only the functions that are total. Hence, ***SET*** is not a full subcategory of ***PAR***. However, it is easy to prove that every set A admits as a co-reflector the partial inclusion $A^{\perp} \to A$ where $A^{\perp}$, also called the "elevation of A", is the set obtained from A by adding an additional element $\perp$ called "bottom" or "undefined". The partial inclusion is the extension of the identity on A that is undefined on $\perp$. Given now an arbitrary partial function $f:B \to A$ there is a unique elevation of f into a total function $f^{\perp}:B \to A^{\perp}$ by mapping to $\perp$ all the elements of B on which f is undefined. This is the standard technique that mathematicians use to deal with partiality in the context of total functions.

Hence, in this case, the "secretary" is also responsible for transforming from the "partial" mode of communication into the "total" mode. This is because, by not being full, the subcategory defines a more specialised mode of interaction. The co-reflector operates a transformation from the more general to the more specific mode.

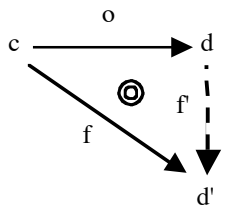
Because Category Theory distinguishes between incoming and outgoing communication, co-reflectors manage, in fact, the incoming

communication only. The dual notion, i.e. the notion that is obtained in the dual category, is called reflector, and manages the outgoing communication:

3.3.12 Definition – reflective subcategory

Let ***D*** be a subcategory of a category ***C***.

1. Let *c* be a ***C***-object. A ***D***-reflection for *c* is a ***C***-morphism $o:c \to d$ for some ***D***-object *d* such that, for any ***C***-morphism $f:c \to d'$ where *d'* is a ***D***-object, there is a unique ***D***-morphism $f':d \to d'$ such that $f=o;f'$ i.e. the following diagram commutes:



2. The category ***D*** is said to be a *reflective subcategory* of ***C*** iff every ***C***-object admits a ***D***-reflection.

Being the dual concept of co-reflections, reflections satisfy all the dual properties of co-reflections. In particular, reflections are essentially "unique" and identities are reflectors iff the subcategory is full. For consistency, in informal discussions, we shall call co-reflectors "insecretaries" or "cosecretaries", and reflectors "outsecretaries" or "secretaries". Examples of reflections are given in 3.6.4.

3.4 Eiffel class specifications

In order to illustrate some of the ways in which Category Theory can be used to capture the semantics of software engineering practice, we introduce a "practical" example related to the Eiffel language. This example will be used in later chapters as well.

As stated in [88], "Eiffel software texts – classes and their routines – may be equipped with elements of formal specification, called assertions, expressing correctness conditions". We shall not detail the language in which these assertions can be written: it can be found in [88].

For our purposes, an Eiffel class specification *e* consists of a triple $<\Sigma,P,I>$ where

- Σ is the class signature consisting of a set (of features). Each feature has its own signature which consists of a pair *<arg,res>* of sequences

of types. Features can be attributes, functions or routines, the sets of which are denoted by $att(\Box)$, $fun(\Box)$ and $rou(\Box)$, respectively
- P provides, for every $r \in rou(\Box)$, a pair $<pre_r, pos_r>$ of Boolean expressions (the pre- and the post-condition of the routine, respectively)
- I is a Boolean expression (the invariant of the class).

The semantics intended for such a specification is the one popularised under the designation "design by contract". Every client of the class when calling a routine r should make sure that the pre-condition for r holds, in which case the post-condition is assured to hold after the call is completed. The class invariant is a condition that is guaranteed by every creation procedure of the class and maintained by any routine.

For an example of a class specification consider bank accounts:

```
deferred class account is
attributes     balance: int, vip: boolean;
routines       deposit(i:nat)
                    require true
                    ensure balance = old balance + i
               withdrawal(i:nat)
                    require balance≥i
                    ensure balance = old balance - i
invariant      vip   balance≥1000
```

Each bank account is capable of handling deposits and withdrawals. These update the balance, captured through an attribute, as specified in the *ensures* clauses. The routine that performs the withdrawals requires the client who makes the call to check that the balance is greater than the amount requested. Accounts that are considered to be *vip* have a balance greater than 1000.

The idea is that specifications are used to indicate *what* software components do, or are required to do, rather than *how* they do it. Changes to the implementations of the specified features are allowed as long as they do not violate the specifications.

Specifications have a "social life" that results from the need to adapt features. Feature adaptation [88] typically comes about when the class is inherited. To be consistent with [88], the notion of morphism that can capture such structural aspects of class specifications has to account for the following operations:

- rename a feature
- merge one or more features; this is called the *join mechanism* and applies to features inherited as deferred (i.e. without a chosen implementation)
- redefine a feature, changing the original signature or pre/post-conditions
- add conditions to the invariant.

There are other circumstances in which features can be changed, like assigning an implementation to a feature that so far was "deferred". Because they relate only to implementations, and do not involve specifications except for discussing the correctness of the changes, i.e. because specifications are not socially active during these operations, we will not consider them in our formalisation.

The redefinition of a feature is subject to several constraints:

- at the level of its signature, the number of arguments and results cannot be changed; changes to their types can be performed subject to a number of conformance rules that, for simplicity, we will ignore
- functions can be redeclared as attributes but not vice-versa; routines cannot be redeclared as attributes or functions, and attributes and functions cannot be redeclared as routines
- pre-conditions can be weakened but not strengthened
- post-conditions can be strengthened but not weakened.

The constraints that apply to pre/post-conditions have in mind the preservation of the contract: any client of a redefined routine has the right to expect a service that complies with the original specification; hence, it cannot be required to test for more conditions before calling the feature and, upon termination, it must get at least what he would normally get from the original feature.

All these constraints lead to the following definition:

A morphism $F: e=<\Box,P,I> \Box e'=<\Box',P',I'>$ of Eiffel class specifications consists of a total function between the class features such that

- for every feature $f\Box\Box$, $arg_{F(f)}= arg_f$ and $res_{F(f)}= res_f$
- for every attribute $a\Box att(\Box)$, $F(a)\Box att(\Box')$
- for every function $f\Box fun(\Box)$, $F(f)\Box att(\Box')\Box fun(\Box')$
- for every routine $r\Box rou(\Box)$, $F(r)\Box rou(\Box')$
- for every $r\Box rou(\Box)$, $F(pre_r) \vdash pre'_{F(r)}$ and $pos'_{F(r)} \vdash F(pos_r)$
- $I' \vdash F(I)$.

As an example of a morphism of Eiffel class specifications, consider the following class specification obtained by inheriting from the previous example:

```
deferred class flexible account is
inherit        account
attributes     credit: nat
redefine       withdrawal(i:nat)
                    require else balance+credit ≥ i
invariant      vip   balance≥10000
```

That is to say, a flexible account extends the account with an attribute credit whose value can be added to the balance for satisfying a withdrawal request. As a counterpart to the added flexibility, the minimum balance for the account to be considered "vip" is now 10000.

The syntax of Eiffel is already such that some of the constraints are automatically met:

- the redeclaration of routines is such that pre-conditions are changed with a ***require else*** clause whose semantics is to add the specified condition to the inherited pre-condition as a disjunct (hence the weakening)
- the post-conditions are changed with a ***ensure then*** clause whose semantics is to add the specified condition to the inherited post-condition as a conjunct (hence the strengthening)
- the invariant clause is added as a conjunct to the invariant inherited from the parent class (hence the strengthening).

It is easy to prove that class specifications and their morphisms constitute a category ***CLASS_SPEC***. Later on in the book we will see how other properties of specifications, like the "join semantics rule" can be accounted for through so-called universal constructions.

3.5 Temporal specifications

In this section, we develop a category whose objects are specifications of process behaviour in Temporal Logic. This category will be used throughout the rest of the book in examples. Whereas our purpose with Eiffel class specifications was to show how Category Theory can apply to "real-life" modelling techniques, with temporal specifications we will try to illustrate, in the simplest way we know, typical properties of Specification Theory as applied to Concurrent Systems.

This example will also serve two other purposes. On the one hand, it will be used to show how relationships between different domains can be developed once they have been formalised in Category Theory. This will be done by relating temporal specifications with a very simple process model that we started building in 3.2.1. The other purpose is to show that there is a degree of "universality" in the kinds of constructions that are typically used across different domains for system modelling, and that this "universality" is very easily made evident, and formally characterised, through the use of Category Theory. Before proceeding, we should make clear that much of this particular topic was jointly developed with Félix Costa. The essential part of the technical details can be found in [32,33].

Specifications are often identified with theories (or theory presentations) in a given logic, an idea that has been around for quite a long time [17], although only more recently explored from the point of view of the temporal logic approach to reactive system specification [38]. In fact, process specifications are usually given as theory *presentations*

rather than theories. By a theory presentation, we mean a pair consisting of a signature and a set of sentences – the non-logical axioms of the specification. The signature identifies the vocabulary symbols that are proper to the object being identified and the sentences provide a description of the properties that are being specified about that object.

We shall be modelling the behaviour of concurrent processes at the level of the actions that are provided by their interfaces. Hence, every signature identifies a set of actions in which a process can engage itself. An example of a signature is that of the specification of a vending machine able to accept coins and deliver cakes and cigars – *{coin,cake,cigar}*.

3.5.1 Definition – signatures of linear temporal logic

A signature of linear temporal logic is a set, the elements of which will be called action symbols.

The language in which the sentences that specify the behaviour of the process are written is that of linear temporal logic. The action symbols provide atomic propositions in the definition of the language associated with a signature:

3.5.2 Definition – language of linear temporal logic

The set of *temporal propositions* ***prop***(Σ) for a signature Σ (Σ-propositions, for short) is inductively defined as follows:

- every action symbol is a temporal proposition
- ***beg*** is a temporal proposition (denoting the initial state)
- if Φ is a temporal proposition so is $(\neg\Phi)$
- if Φ_1 and Φ_2 are temporal propositions so are $(\Phi_1 \ \Phi_2)$, $(\Phi_1 \mathbf{U} \Phi_2)$.

The temporal operator is **U** (until). Its semantics is defined below; informally, $(\Phi_1 \mathbf{U} \Phi_2)$ is intended to hold whenever, from tomorrow, Φ_2 will eventually hold and, until then, but not necessarily then, Φ_1 will hold. Other temporal operators such as ***X*** (next or tomorrow), ***F*** (eventually) and ***G*** (always) can be defined as abbreviations [68]. We will often make use of the operator ***W*** (weak until): $(\Phi_1 \mathbf{W} \Phi_2)$ will hold whenever, from tomorrow, either Φ_2 will eventually hold and until then, but not necessarily then, Φ_1 will hold, or Φ_2 will forever be false and Φ_1 true.

3.5.3 Definition – semantics of linear temporal logic

The language of linear temporal logic is interpreted over infinite sequences of sets of actions. That is, an interpretation structure for a signature Σ is a sequence $\Sigma\Sigma(2^{\Sigma})^{\Sigma}$. These are canonical Kripke structures for linear, discrete, propositional logic [114]. Each infinite sequence represents a possible behaviour for the process being specified. The sets of actions represent the events that take place during the lifetime of

the process. As already explained in 3.2.1, the event that consists of the empty set of actions represents a transition performed by the environment without the participation of the process.

A Σ-proposition φ is said to be true for $\sigma \in (2^{\Sigma})^{\omega}$ at state i, which we write $\sigma \models_{\Sigma,i} \varphi$ iff:

- if $\sigma = \sigma_0, \sigma \models_{\Sigma,i} \varphi$ iff $\sigma_0 \in \Phi(i)$
- $\sigma \models_{\Sigma,i}$ ***beg*** iff $i=0$
- $\sigma \models_{\Sigma,i} (\neg \varphi)$ iff it is not the case that $\sigma \models_{\Sigma,i} \varphi$
- $\sigma \models_{\Sigma,i} (\varphi_1 \vee \varphi_2)$ iff $\sigma \models_{\Sigma,i} \varphi_1$ implies $\sigma \models_{\Sigma,i} \varphi_2$
- $\sigma \models_{\Sigma,i} (\varphi_1 \mathbf{U} \varphi_2)$ iff, for some $j>i$, $\sigma \models_{\Sigma,j} \varphi_2$ and $\sigma \models_{\Sigma,k} \varphi_1$ for every $i<k<j$.

The proposition φ is said to be *true* in σ, written $\sigma \models_{\Sigma} \varphi$, if and only if $\sigma \models_{\Sigma,i} \varphi$ at every state i. We also write $\sigma \models_{\Sigma} \Phi$ for a collection of propositions Φ meaning that each proposition of Φ is true in σ, and $\Sigma \models_{\Sigma} \varphi$ for a collection Σ of sequences meaning that φ is true in every $\sigma \in \Sigma$.

Finally, for every set Φ of Σ-propositions and every Σ-proposition φ, φ is a consequence of Φ – ($\Phi \vdash_{\Sigma} \varphi$) – if and only if φ is true in every sequence that makes all the propositions in Φ true.

The corresponding notion of theory is given as usual for the closure system induced by the consequence relation:

3.5.4 Definition – temporal theories and presentations

1. Let Σ be a signature. A subset Φ of ***prop***(Σ) is said to be *closed* if and only if, for every $\varphi \in$ ***prop***(Σ), $\Phi \vdash_{\Sigma} \varphi$ implies $\varphi \in \Phi$. By $c_{\Sigma}(\Phi)$ we denote the least closed set that contains Φ.
2. A *temporal theory* is a pair $<\Sigma,\Phi>$ where Σ is a signature and Φ is a closed set of Σ-propositions.
3. A *theory presentation* is a pair $<\Sigma,\Phi>$ where Σ is a signature and Φ is a set of Σ-propositions. The presented theory is $<\Sigma,c_{\Sigma}(\Phi)>$.

That is to say, a theory consists of a set of sentences that is closed for consequence: it contains all the theorems that can be derived from its sentences. A presentation is not necessarily closed under consequence: it provides a more "economical" way of specifying the intended behaviour of a system. A presentation consists only of a selected set of properties (also called the "axioms" of the presentation) that are required of the system, leaving the computation of the properties that can be derived from this selected set (its "theorems") to the proof theory of the logic. Whereas theories, in general, and in particular for temporal logic, are infinite, presentations can be finite or, at least, recursivebly enumerable. Hence, specifications are usually identified with presentations, not with theories.

3.5.5 Exercise

Prove that the operators $c_\Box$ defined above satisfy the following properties:

- reflexivity: for every $\Box\Box$***prop***$(\Box)$, $\Box\Box c_\Box(\Box)$
- monotonicity: for every $\Box,\Box\Box$***prop***$(\Box)$, $\Box\Box\Box$ implies $c_\Box(\Box)\Box c_\Box(\Box)$
- idempotence: for every $\Box\Box$***prop***$(\Box)$, $c_\Box(c_\Box(\Box))\Box c_\Box(\Box)$

3.5.6 Example – a vending machine

As an example, consider the following specification of a vending machine:

```
specification  vending machine is
signature      coin, cake, cigar
axioms         beg   ((¬cake□¬cigar) □
                    (coin   (¬cake□¬cigar)Wcoin))
               coin   (¬coin)W(cake cigar)
               (cake cigar)   (¬cake□¬cigar)Wcoin
               cake   (¬cigar)
```

This machine is able to accept coins, deliver cakes and deliver cigars. The machine is initialised so as to accept only coins (first axiom). Once it accepts a coin it can deliver either a cake or a cigar (second axiom), but not both (fourth axiom). After delivering a cake or a cigar it is ready to accept more coins (third axiom).

As already mentioned, the model of process behaviour that we are adopting reflects an (abstract) synchronous, multi-processor architecture in which, at each transition, several actions may be executed concurrently. We are going to show that this synchronous flavour is captured through the notion of specification morphism (interpretation between theories) as a mathematical model of the relationship between systems and their components.

An interpretation between theories is typically defined as a mapping between their signatures that preserves theorems. Notice, once again, the idea of structure preservation presiding to the definition of morphisms. The structure being preserved in this case is the one given by the properties of the processes involved as captured through the theorems of the specifications.

3.5.7 Definition – interpretations between theories

1. Let $\Box$ and $\Box'$ be signatures. Every function $f:\Box\Box\Box'$ extends to a *translation map* ***prop***(f): ***prop***$(\Box)\Box$ ***prop***$(\Box')$ as follows:
   - ***prop***$(f)(beg)=beg$
   - if $a\Box\Box$ then ***prop***$(f)(a)=f(a)$
   - ***prop***$(f)(\neg\Box)=\neg$***prop***$(f)(\Box)$
   - ***prop***$(f)(\Box_1\ \Box_2)=$***prop***$(f)(\Box_1)$ ***prop***$(f)(\Box_2)$

- $\mathbf{prop}(f)(\Phi_1 U \Phi_2) = \mathbf{prop}(f)(\Phi_1) U \mathbf{prop}(f)(\Phi_2)$

2. An *interpretation* between two theories (or theory morphism) $\langle \Sigma_1, \Phi_1 \rangle$ and $\langle \Sigma_2, \Phi_2 \rangle$ is a map $f: \Sigma_1 \to \Sigma_2$ such that $\mathbf{prop}(f)(\Phi_1) \subseteq \Phi_2$.
3. A *morphism* between two presentations $\langle \Sigma_1, \Gamma_1 \rangle$ and $\langle \Sigma_2, \Gamma_2 \rangle$ is a map $f: \Sigma_1 \to \Sigma_2$ such that $\mathbf{prop}(f)(c_{\Sigma_1}(\Gamma_1)) \subseteq c_{\Sigma_2}(\Gamma_2)$.

3.5.8 Proposition

1. Temporal theories and interpretations between theories define a category called $\mathbf{THEO}_{LTL}$.
2. Presentations of temporal theories and their morphisms define a category called $\mathbf{PRES}_{LTL}$. Furthermore, $\mathbf{THEO}_{LTL}$ is a subcategory of $\mathbf{PRES}_{LTL}$.

proof

1. Although we have not made it explicit, the composition law and identity map that we have in mind for theories are the ones inherited from signatures as sets. Hence, this is another example of constructing a category by adding structure: the underlying category is that of sets and the structure being added is the collection of theorems that constitute the specification. Therefore, the properties of the composition law and identity map are inherited from **SET**; we only have to prove the closure properties, i.e. that the composition of theory morphisms as functions is still a theory morphism and that the identity function is a theory morphism. The latter is, of course, trivial because the translation map induced by the identity on signatures is itself the identity. The former can be proved as follows:
   - first of all, given $f: \Sigma \to \Sigma'$ and $g: \Sigma' \to \Sigma''$, we have to prove that $\mathbf{prop}(f;g) = \mathbf{prop}(f);\mathbf{prop}(g)$. That is, the translation map induced by a composite function is the composition of the translation maps induced by the components. This can be proved by structural induction and is left as an exercise.
   - given theory morphisms $f: \langle \Sigma, \Phi \rangle \to \langle \Sigma', \Phi' \rangle$ and $g: \langle \Sigma', \Phi' \rangle \to \langle \Sigma'', \Phi'' \rangle$, $\mathbf{prop}(f;g)(\Phi) = \mathbf{prop}(g)(\mathbf{prop}(f)(\Phi))$ as proved above. Because f is a theory morphism, $\mathbf{prop}(f)(\Phi) \subseteq \Phi'$, which implies $\mathbf{prop}(f;g)(\Phi) \subseteq \mathbf{prop}(g)(\Phi')$. Because g is a theory morphism, $\mathbf{prop}(g)(\Phi') \subseteq \Phi''$, which implies $\mathbf{prop}(f;g)(\Phi) \subseteq \Phi''$. Hence, $(f;g)$ is a theory morphism.
2. The proof that theory presentations and their morphisms define a category can proceed exactly in the same way as for theories. The fact that $\mathbf{THEO}_{LTL}$ is a subcategory of the resulting category has a very trivial proof:
   - clearly, every theory is a presentation: it just happens to be closed.
   - given a theory morphism $f: \langle \Sigma, \Phi \rangle \to \langle \Sigma', \Phi' \rangle$ we have to prove that it defines a presentation morphism, i.e. that $\mathbf{prop}(f)(c_\Sigma(\Phi)) \subseteq c_{\Sigma'}(\Phi')$. Because Φ and Φ' are closed, $c_\Sigma(\Phi) = \Phi$ and $c_{\Sigma'}(\Phi') = \Phi'$. Because f is a theory morphism, $\mathbf{prop}(f)(\Phi) \subseteq \Phi'$.
   - the properties of the composition law and the identity map are trivially proved.

For simplicity, from now on, we will overload the notation and use f instead of $\mathbf{prop}(f)$ unless there is a risk of confusion.

Checking that a signature map defines a morphism between two presentations by checking that every theorem of the source is translated to a theorem of the target is not "practical". Typically, one would prefer to check that only the axioms of the presentation are translated into theorems, from what it should follow that the theorems are preserved. Although this is not true for an arbitrary logic, it is a property of the temporal logic that we defined above as proved below.

In order to do so, we need to be able to relate the models of the signatures involved:

3.5.9 Definition/Proposition – reducts

1. Let Σ and Σ' be signatures and $f:\Sigma\to\Sigma'$ a total function. Given an interpretation structure $\mathcal{I}'\in(2^{\Sigma'})^{\mathbb{N}}$ for Σ', we define its *reduct* $\mathcal{I}'|_f$ as the interpretation structure for Σ defined by $\mathcal{I}'|_f(i)=f^{-1}(\mathcal{I}'(i))$. That is, the reduct of a sequence $\mathcal{I}'$ of Σ'-actions is the sequence that consists, at each point i, of the Σ-actions that are translated through f into $\mathcal{I}'(i)$.
2. Let Σ and Σ' be signatures, $f:\Sigma\to\Sigma'$ a total function, $\mathcal{I}'\in(2^{\Sigma'})^{\mathbb{N}}$ an interpretation structure for Σ', and α a Σ-proposition. Then, for every i, $\mathcal{I}'\models_{\Sigma',i}f(\alpha)$ iff $\mathcal{I}'|_f\models_{\Sigma,i}\alpha$ (the *satisfaction condition*).

proof

The proof of 2 is by induction on the structure of α. The base case (action symbols) results directly from the definition of reduct: $f(\alpha)\in\mathcal{I}'(i)$ iff $\alpha\in f^{-1}(\mathcal{I}'(i))$ $(=\mathcal{I}'|_f(i))$. The induction step presents no difficulties.

3.5.10 Proposition – presentation lemma

1. Given a signature morphism $f:\Sigma_1\to\Sigma_2$ and $\alpha\in\mathbf{prop}(\Sigma_1)$, $f(c_{\Sigma_1}(\alpha))\subseteq c_{\Sigma_2}(f(\alpha))$.
2. Given presentations $\langle\Sigma_1,\Gamma_1\rangle$ and $\langle\Sigma_2,\Gamma_2\rangle$, a map $f:\Sigma_1\to\Sigma_2$ is a morphism between them iff $f(\Gamma_1)\subseteq c_{\Sigma_2}(\Gamma_2)$. This result is usually called "the presentation lemma".

proof

1. Let $\alpha\in c_{\Sigma_1}(\Gamma)$. We have to prove that $f(\alpha)\in c_{\Sigma_2}(f(\Gamma))$. For that purpose, consider an arbitrary interpretation structure $\mathcal{I}'\in(2^{\Sigma'})^{\mathbb{N}}$ in which $f(\Gamma)$ is true. From what was just proved, all the propositions in Γ are true in $\mathcal{I}'|_f$. But, then, so is α because $\alpha\in c_{\Sigma_1}(\Gamma)$. Applying the same result, but in the other direction, we know that $f(\alpha)$ is true in $\mathcal{I}'$.
2. All that needs to be proved is that $f(c_{\Sigma_1}(\Gamma_1))\subseteq c_{\Sigma_2}(\Gamma_2)$ iff $f(\Gamma_1)\subseteq c_{\Sigma_2}(\Gamma_2)$.
   - The forward implication is an immediate consequence of the reflexivity of closure, which allows us to derive that $\Gamma_1\subseteq c_{\Sigma_1}(\Gamma_1)$.
   - Consider now the reverse implication and assume that $f(\Gamma_1)\subseteq c_{\Sigma_2}(\Gamma_2)$. Because closure operators are monotone, we can infer $c_{\Sigma_2}(f(\Gamma_1))\subseteq c_{\Sigma_2}(c_{\Sigma_2}(\Gamma_2))$. Because closure operators are idempotent, $c_{\Sigma_2}(c_{\Sigma_2}(\Gamma_2))=c_{\Sigma_2}(\Gamma_2)$. Hence,

$c_{\Box_2}(f(\Box_1)) \Box c_{\Box_2}(\Box_2)$. Using the result proved in 1, we obtain by transitivity $f(c_{\Box_1}(\Box_1)) \Box c_{\Box_2}(\Box_2)$.

3.5.11 Example – a regulated vending machine

As an example of the use of morphisms for system modelling, consider the following specification:

```
specification  regulated vending machine is
signature      coin, cake, cigar, token
axioms         beg   (¬cake□¬cigar□¬token) □
                   (coin   (¬cake□¬cigar)Wcoin)
               coin  (¬coin)W(cake cigar)
               coin  (¬cigar)Wtoken
               (cake cigar)  (¬cake□¬cigar)Wcoin
               cake  (¬cigar)
               token  (¬cake□¬cigar□¬coin)
```

That is to say, the machine is now extended to be able to accept tokens, and is regulated in such a way that it will only deliver a cigar if, after having accepted a coin, it receives a token. The last axiom says that coins, cakes and cigars cannot be used as tokens...

We can see this specification as resulting from the previous vending machine through the superposition of some mechanism for controlling the sales of cigars. Later on in the book (6.1.24), we shall see how a regulator can be independently specified and connected to the vending machine in order to achieve the required superposition. That is to say, we shall see how the regulated vending machine can be defined as a configuration of which the original vending machine, as well as the regulator, are components.

In the meanwhile, we shall analyse how theory morphisms can be used to identify components of systems. For instance, in the example above, we can establish a morphism between the specifications of the vending machine and the regulated vending machine. The signature morphism maps the actions of the vending machine to those of the regulated vending machine that have the same names. To prove that the signature morphism defines an interpretation between the two specifications, we have to check if every axiom of the specification of the vending machine (3.5.6) is translated to a theorem of the regulated vending machine:

- the initialisation condition is preserved: it is strengthened with the condition $\neg token$
- the second axiom is translated directly into the second axiom
- the third axiom is translated directly into the fourth axiom
- and the fourth axiom is translated directly into the fifth axiom.

The fact that morphisms identify components of systems is captured by the following result:

3.5.12 Proposition

Let $<\Sigma_1,\Gamma_1>$ and $<\Sigma_2,\Gamma_2>$ be theories and $f:\Sigma_1 \to \Sigma_2$ a signature morphism. Then, f defines a theory morphism iff, for every model $\mathcal{T}'$ of $<\Sigma_2,\Gamma_2>$, $\mathcal{T}'|_f$ is a model of $<\Sigma_1,\Gamma_1>$.

proof

1. Assume that f defines a theory morphism and let $\mathcal{T}'$ be a model of $<\Sigma_2,\Gamma_2>$. To prove that $\mathcal{T}'|_f$ is a model of $<\Sigma_1,\Gamma_1>$, let $\alpha \in \Gamma_1$. Because f is a theory morphism, $f(\alpha) \in \Gamma_2$ and, hence, $f(\alpha)$ is true in $\mathcal{T}'$. But, by the fundamental property of reducts, α is true in $\mathcal{T}'|_f$. Hence, $\mathcal{T}'|_f$ is a model of $<\Sigma_1,\Gamma_1>$.
2. Assume now that, for every model $\mathcal{T}'$ of $<\Sigma_2,\Gamma_2>$, $\mathcal{T}'|_f$ is a model of $<\Sigma_1,\Gamma_1>$. Let $\alpha \in \Gamma_1$. We have to prove that $f(\alpha) \in \Gamma_2$. Let $\mathcal{T}'$ be a model of $<\Sigma_2,\Gamma_2>$. Because $\mathcal{T}'|_f$ is a model of $<\Sigma_1,\Gamma_1>$, α is true in $\mathcal{T}'|_f$. By the fundamental property of reducts (opposite direction), $f(\alpha)$ is true in $\mathcal{T}'$.

That is to say, given a morphism between two temporal specifications, every behaviour of the target is projected back to a model of the source. Because the reduct identifies the actions of the source that occur at each point in the execution of the target, we can say that an interpretation between theories identifies in every behaviour that is according to the target specification, a behaviour that is according to the source specification. Hence, the morphism recognises in the source a component of the target.

Notice that it is not necessary that every behaviour of the source (component) be recovered through the reduct. Indeed, as a component of a larger system, some behaviours of the component may be lost because of interactions with other components. This is the case above. The behaviours in which a coin is followed immediately by a cigar are not part of any model of the regulated vending machine because a token is required after the coin before the cigar can be delivered.

3.6 Closure systems

In the previous section, we presented several categories related to the specification of systems on the basis of properties stated in the language of temporal logic. We will show in the next chapters that these categories satisfy a number of properties that make them useful to modularise specifications. However, these properties are in no way particular to the temporal logic that was chosen to express specifications. They are shared by a number of structures that generalise what, sometimes, are called "closure systems". This section presents the initial step in a sequence of constructions that will end up in the definition of institutions [61], π-institutions [45], and general logics [87]. These are all categorical generalisations of the notions of Logic and associated concepts

like theories that bring out the structural properties that make them useful for specification.

3.6.1 Definition – closure system

We call a *closure system* a pair $<L,c>$ where L is a set and $c:2^L \to 2^L$ is a total function satisfying the following properties:

- reflexivity: for every $\Gamma \subseteq L$, $\Gamma \subseteq c(\Gamma)$
- monotonicity: for every $\Gamma,\Delta \subseteq L$, $\Gamma \subseteq \Delta$ implies $c(\Gamma) \subseteq c(\Delta)$
- idempotence: for every $\Gamma \subseteq L$, $c(c(\Gamma)) \subseteq c(\Gamma)$

3.6.2 Definition/Proposition – category of closure systems

We define the category ***CLOS*** of closure systems by defining a morphism $f:<L,c> \to <L',c'>$ to be a map $f:L \to L'$ satisfying, for every $\Gamma \subseteq L$, $f(c(\Gamma)) \subseteq c'(f(\Gamma))$.

proof

Again an example of adding structure to ***SET***.

1. the identity function on sets is trivially a morphism of closure systems;
2. consider now two morphisms $f:<L,c> \to <L',c'>$ and $g:<L',c'> \to <L'',c''>$, and $\Gamma \subseteq L$. Because f is a morphism, $f(c(\Gamma)) \subseteq c'(f(\Gamma))$. This also implies that $g(f(c(\Gamma))) \subseteq g(c'(f(\Gamma)))$. On the other hand, because g is a morphism, $g(c'(f(\Gamma))) \subseteq c''(g(f(\Gamma)))$. Hence, $(f;g)(c(\Gamma)) \subseteq c''((f;g)(\Gamma))$.

As an example of closure systems we can give, for every temporal signature Σ, the pair $clos(\Sigma)$ formed by ***prop***(Σ) as defined in 3.5.2 together with the closure operator defined in 3.5.4. Notice that the result proved in 3.5.10 shows that every signature morphism induces a morphism between the corresponding closure systems.

3.6.3 Definition/Proposition – theories in closure systems

Consider a closure system $<L,c>$.

1. We say that $\Gamma \subseteq L$ is *closed* iff $\Gamma = c(\Gamma)$.
2. We define the category $\boldsymbol{THEO}_{<L,c>}$ whose objects (theories) are the closed subsets of L and morphisms are given by inclusions.
3. We define the category $\boldsymbol{PRES}_{<L,c>}$ whose objects (theory presentations) are the subsets of L and morphisms given by the pre-order $\Gamma \leq \Delta$ iff $c(\Gamma) \subseteq c(\Delta)$.
4. We define the category $\boldsymbol{SPRES}_{<L,c>}$ whose objects (strict presentations) are the subsets of L ordered by inclusion.

proof

These are trivial examples of categories that are given through pre-ordered sets.

We call the objects of $\boldsymbol{SPRES}_{<L,C>}$ "strict presentations" because the morphisms require that the axioms be preserved. The more attentive

reader will have noticed that these notions do not coincide exactly with the definitions used in section 3.5 because they do not contemplate changes of language. We shall generalise these notions in later sections to account for relationships between theories in different languages.

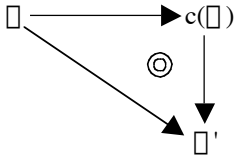
The following proposition defines several relationships between these categories. For simplicity, we omit the reference to the underlying closure system.

3.6.4 Proposition

1. ***THEO*** is a full subcategory of ***PRES*** (and of ***SPRES***), and ***SPRES*** is a subcategory of ***PRES***.
2. ***THEO*** is a reflective subcategory of ***PRES*** (and of ***SPRES***), but ***SPRES*** is not.
3. ***THEO*** is a co-reflective subcategory of ***PRES*** (but not of ***SPRES***) and so is ***SPRES***.

proof

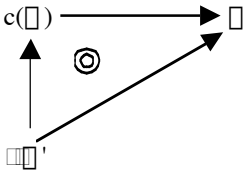
1. The proof that ***THEO*** is a (full) subcategory of ***SPRES*** and ***PRES*** is trivial. The result that ***SPRES*** is a subcategory of ***PRES*** is an immediate consequence of the monotonicity of the closure operator. This subcategory is not full because theorems may be preserved even if the axioms are not.
2. The first results are about closure as a reflector. Indeed, we are going to prove that, given an arbitrary presentation Π, its reflector is its closure $c(\Pi)$. There are two parts in the proofs: (1), the very existence of the reflector as a morphism; (2), its universal property. All the reasoning evolves around the following "diagram":



- In the case of ***THEO*** as a subcategory of ***PRES***, part 1 translates to the inclusion $c(\Pi) \subseteq c(c(\Pi))$, which is a corollary of the reflexivity of the closure relation. Part 2 translates to $c(\Pi) \subseteq c(\Pi')$ *implies* $c(\Pi) \vdash \Pi'$ because the morphism on the right is in ***THEO***. This holds because, being a theory, $c(\Pi')=\Pi'$.
- In the case of ***THEO*** as a subcategory of ***SPRES***, part 1 translates to the inclusion $\Pi \subseteq c(\Pi)$ because the morphism is now strict. Again, this is a corollary of the reflexivity of the closure relation. Part 2 translates to $\Pi \subseteq \Pi'$ *implies* $c(\Pi) \vdash \Pi'$ because the morphism on the right is in ***THEO*** and the one on the left is strict. This is a consequence of the monotonicity of the closure operator and the fact that, being a theory, $c(\Pi')=\Pi'$.
- The fact that ***SPRES*** is not a reflective subcategory of ***PRES*** can be inferred from part 2: $c(\Pi) \subseteq c(\Pi')$ *implies* $c(\Pi) \vdash \Pi'$ does not necessarily hold because, being a presentation, Π' is not necessarily closed.

- Summarising, on the side of reflections, these results capture the fact that, for the purposes of "outward communication", i.e. for being interpreted, presentations can delegate on theories but not on strict presentations because, whereas the former will use the full theory for the interpretation, the latter will only look at relationships between axioms.

3. The second set of results is about the co-reflective properties of closure. The relevant "diagram" now is:



- In the case of ***THEO*** as a subcategory of ***PRES***, part 1 translates to the inclusion $c(c(\Phi))\subseteq c(\Phi)$, which is a corollary of the idempotence of the closure operator. Part 2 translates to $c(\Phi')\subseteq c(\Phi)$ *implies* $\Phi'\subseteq c(\Phi)$ because the morphism on the left is in ***THEO***. This holds because of reflexivity.
- The fact that ***THEO*** is not a co-reflective subcategory of ***SPRES*** can be inferred from part 1: the inclusion $c(\Phi)\subseteq\Phi$ does not necessarily hold because, being a presentation, Φ is not necessarily closed.
- In the case of ***SPRES*** as a subcategory of ***PRES***, part 1 translates again to the inclusion $c(c(\Phi))\subseteq c(\Phi)$. Part 2 also translates to $c(\Phi')\subseteq c(\Phi)$ *implies* $\Phi'\subseteq c(\Phi)$ because the morphism on the left is strict. This holds because of reflexivity.
- Summarising, on the side of co-reflections, the existence of a co-reflection from strict into loose interpretations reflects (pun intended) the fact that in order to interpret another theory presentation, a given theory presentation just needs to consider the axioms of the source. On the other hand, because strict presentations are not expressive enough to interpret theories, they do not admit theories as co-reflectors, i.e. they cannot allow theories to handle their in-communication.

Globally, this result shows that theories and theory presentations with the looser notion of morphism define, essentially, the same structure. The same does not happen with strict presentations. Strict presentations can co-reflect presentations and be reflected by theories but not the other way around because they are not expressive enough to capture closure.

4 Universal constructions

We have already hinted to the fact that Category Theory provides a totally different approach to the characterisation of a given domain of objects, namely to the fact that objects are characterised by their "social life", or "interactions", as captured by morphisms. In this chapter, we take this view one step further by showing how certain objects, or constructions, can be characterised in terms of standard relationships that they exhibit with respect to the rest of the universe (of relevant constructions). It is in this sense that these constructions are usually called "universal". At the same time, we will start shifting our focus from the social life of individual objects to that of groups or societies of interacting objects.

Indeed, this is where we start shifting some of the emphasis from the manipulation of objects to that of diagrams as models of complex systems, i.e. as expressions of what are often called configurations, be it configurations of running systems as networks of simpler components, the way complex programs (text) or specifications are put together from modules, the inheritance structures according to which program modules are organised, etc. This where the term “universal” conjures up general principles such as Goguen's famous dogma [56]:

"given a category of widgets, the operation of putting a system of widgets together to form a super-widget corresponds to taking a colimit of the diagram of widgets that shows how to interconnect them".

One of our goals with this book is to make the Software community aware that, like in the other domains of Science and Engineering, there are “universal laws” or “dogmas” in our area that can manifest themselves in different ways but be unified under a “categorical roof”. For instance, we will be showing that the relationship between “composition" and “conjunction”, as debated in [116], is “universal” in a very precise, mathematical way, which is precisely the one captured by Goguen’s dogma.

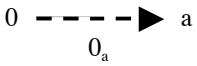
This chapter is only an entry point to the ways Category Theory can address the complexity of system development. Subsequent chapters will introduce further constructions, techniques, and applications.

4.1 Initial and terminal objects

At least at first-sight, the first universal constructions that we define are not so much "constructions" but "identifications" of distinguished objects. What distinguishes them from the other objects are what are usually called "universal properties", a term that is associated, and used interchangeably in the literature, with "universal construction".

4.1.1 Definition – initial objects

An object x of a category $\boldsymbol{C}$ is said to be *initial* iff, for every object y of $\boldsymbol{C}$, there is a unique morphism from x to y. A generic initial object is usually denoted by 0. The unique morphism from an initial object 0 into an object a is denoted by 0_a.



4.1.2 Proposition

1. Any two initial objects in a given category are isomorphic.
2. Any object isomorphic to an initial object is also initial.

proof

1. Let x and y be initial objects, and $f:x \to y$ and $g:y \to x$ the associated morphisms given by their universal properties. Consider now the morphism $(f;g):x \to x$. Because x is initial, we know that there is only one morphism with x as source and target. Hence, $(f;g)=id_x$. The same reasoning applied to y gives us $(g;f)=id_y$, showing that x and y are isomorphic.
2. Left as an exercise.

Hence, we usually refer to *the* initial object, if one exists.

4.1.3 Example

1. In ***SET***, the initial object is the empty set. This is because the empty set can be mapped to any other set in a unique way: through the empty function. This also justifies the notation 0 usually adopted for initial objects.
2. In ***LOGI***, the initial object is $\bot$ (any contradiction). This is because anything can be derived from a contradiction. This again is consistent with the use of the notation 0 (as the truth value for false) for initial objects.
3. In ***PAR***, the initial object is also the empty set: all functions from the empty set are, in fact, total, hence it is not surprising that we get the same initial object than in ***SET***.

4. In $\mathbf{SET}_{\bullet}$, the initial objects are the singletons $\langle\{a\},a\rangle$. This is because there is one and only one way of mapping the designated object of a pointed set: to the designated element of the target pointed set. Notice that, although $\mathbf{SET}_{\bullet}$ was built over **SET**, the initial objects of the two categories do not coincide. Indeed, when adding structure to a given category to make another category, the universal constructions may change precisely because the structure has been changed. Nevertheless, the spirit of "emptiness" is still there – $\langle\{a\},a\rangle$ is empty of "proper" elements.

4.1.4 Definition – terminal objects

An object is terminal in a category **C** iff it is initial in $\mathbf{C}^{op}$. That is, x is terminal in **C** iff, for every object y of **C**, there is a unique morphism from y to x. A generic terminal object is usually denote by 1 and the unique morphism from an object a is denoted by 1_a.

$$1 \xleftarrow{\quad 1_a \quad} a$$

Once again, terminal objects are isomorphic and, therefore, we usually refer to *the* terminal object of a category, if one exists.

4.1.5 Examples

1. In **SET**, the terminal objects are the singletons. This is because there is one, and only one way of mapping any given set to a singleton: by mapping all the elements of the source set (even if there is none…) to the element of the singleton. Any set with more than one element is clearly not terminal because it provides a choice for the target image of any element of the source set, i.e. it does not satisfy the uniqueness criterion. The empty set is not terminal because it only admits total functions from itself, i.e. it does not satisfy the existence criterion.

   Notice that all singletons are indeed isomorphic in **SET**. This is consistent with what we said before about the way Category Theory handles sets as objects: because we are not allowed to look into a set to see what elements it has, there is no way we can distinguish two singletons in terms of their structural properties. This also justifies the use of the notation 1 for arbitrary terminal objects.

   On the other hand, any singleton can be used to identify the different elements of any non-empty set A by noticing that each element $a \in A$ defines, in a unique way, a function $a:1 \to A$. Indeed, following the idea that morphisms characterise the "social life" of the objects of a category, the social relationships that a singleton set can establish with an arbitrary non-empty set characterise precisely the

elements of that set. The reader may wish to return to 2.2.12 and 2.3.5 to see examples that make use of this analogy, the notation $\{\bullet\}$ being used for "the" terminal set.

This idea can be generalised to any terminal object 1_C in an arbitrary category $\mathbf{C}$ as a mechanism for identifying what, in Category Theory, are called points [74], constants [12], or global elements [22] of an object x: morphisms of the form $1 \to x$.

2. In ***LOGI***, the terminal object is $\top$ (any tautology). This is because any tautology can be derived from any other formula. Again, this is also consistent with the use of the notation *1* (the truth value for true) for terminal objects.
3. In ***PAR***, the terminal object is also the empty set: there is always one and only one way of mapping any set to the empty one – through the partial function that is undefined in all the elements of the source!

   Notice that, in spite of being a subcategory of ***PAR***, ***SET*** has different terminal objects. One the one hand, the undefined function not being total, the empty set cannot play the role of terminal object in ***SET***. On the other hand, because sets in ***PAR*** have "a richer social life", i.e. more morphisms, the singletons in ***PAR*** relate differently to other sets than they do in ***SET***; in particular, besides the constant (total) function, they also admit the (partial) undefined one. Indeed, categories cannot be expected to share the same kind of initial/terminal objects with their subcategories, unless there is some special structural property that justifies so (see exercise below).

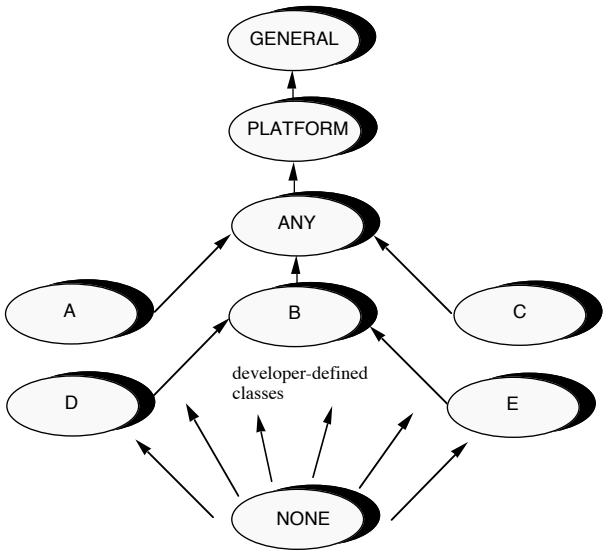
   Hence, in ***PAR***, the initial and the terminal objects coincide. Objects of an arbitrary category that are both initial and terminal are sometimes called *null* [79] or *zero* [1] objects.
4. In $\mathbf{SET}_{\bullet}$, the terminal objects are also the singletons $\langle\{a\},a\rangle$. Hence, in $\mathbf{SET}_{\bullet}$, the initial and the terminal objects also coincide.

4.1.6 Exercises

1. Let $\mathbf{C}$ be an arbitrary category and $a{:}\mathbf{C}$. Show that id_a is initial in $a \downarrow \mathbf{C}$. Further show that if a is terminal in $\mathbf{C}$, id_a is terminal in $a \downarrow \mathbf{C}$.
2. Show that the category $\mathbf{SET}_{\bullet}$ of pointed sets (3.2.1) "corresponds" to the comma category $1 \downarrow \mathbf{SET}$ (3.2.2). This correspondence will be shown later on to define an isomorphism of categories. What can be said about the category $\mathbf{SET} \downarrow 1$ (3.2.3)?
3. Show that, in any pre-order (2.2.5), initial (resp. terminal) objects coincide with the minimum (resp. maximum), if it exists.
4. Show that, if $\mathbf{D}$ is a full subcategory of $\mathbf{C}$, any initial (resp. terminal) object of $\mathbf{C}$ that is an object of $\mathbf{D}$ is also initial (resp. terminal) in $\mathbf{D}$.

4.1.7 Example – Eiffel's inheritance structure

The inheritance structure of Eiffel has an initial object – the class *NONE* that inherits from every other class, and a terminal object – the class *GENERAL* from which every other class inherits.



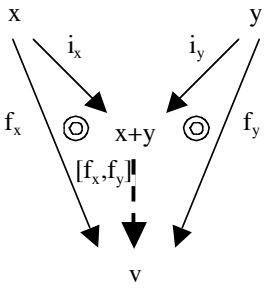
4.2 Sums and products

Intuitively, the universal constructions that are the theme of this chapter concern the possibility of finding objects that are able to capture the social lives of whole collections of objects and morphisms showing how the objects relate to one another. This is because, for instance, we are interested in the study of properties of whole systems, e.g. emergent behaviour, rather than isolated objects. The obvious questions that we have to answer is "what exactly is meant by a *collection*" and "what is the social life of such a collection of interacting objects"…

Leaving the first question out of the discussion for a while, and relying on an intuitive level of understanding for the time being, we start with a simple example. Not the simplest, though: this would be for collections consisting of only one object, in which case only isomorphic objects would have a social life that is able to capture the one of the given object… The simplest non-trivial example is that of a collection of two objects with no interactions between them, for instance two processes running in parallel with no communication between them, or two software modules with no dependencies between them.

Consider first the characterisation of the "out-going" communication for such a collection x, y of objects. We will say that, as a collection, x and y interact with other objects v via morphisms $f_x:x\to v$, $f_y:y\to v$. For instance, to say how a software module v uses x and y collectively, we have to say how it uses each of them in particular. This is because there are no dependencies between x and y, otherwise we would have to express the fact that these dependencies are respected.

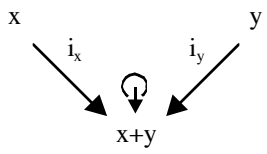
An object that is able to stand for the relationships that a collection x, y of objects has towards its environment is called their sum and denoted by $x+y$. In fact, the sum is more than an object. We need to make explicit how x and y relate to it. This requires morphisms (injections) $i_x:x\to x+y$ and $i_y:y\to x+y$. The ability for this object and the connecting morphisms to characterise the relationships that the collection has towards its environment can then be expressed by the property that any such interaction $f_x:x\to v$, $f_y:y\to v$ can be performed via $x+y$ in the sense that there is a unique morphism $k:x+y\to v$ through which f_x and f_y can be intercepted, i.e. $i_x;k=f_x$ and $i_y;k=f_y$. The morphism k is often represented by $[f_x,f_y]$.



4.2.1 Definition – sum

Let $\boldsymbol{C}$ be a category and x,y objects of $\boldsymbol{C}$. An object z is said to be a *sum* (or *coproduct*) of x and y with injections $i_x:x\to z$ and $i_y:y\to z$ iff for any object v and pair of morphisms $f_x:x\to v$, $f_y:y\to v$ of $\boldsymbol{C}$ there is a unique morphism $k:z\to v$ in $\boldsymbol{C}$ such that $i_x;k=f_x$ and $i_y;k=f_y$.

In order to express, in diagrams, that the objects and morphisms shown are related by a universal construction, we use the symbols [illegible] and [illegible] according to rules that we will make explicit in each case. For instance, in the case of sums,



As it should be expected, the exact identity of the sum should be of no importance, just the way in which it relates to the other objects. Hence, the following property holds:

4.2.2 Proposition

If a sum of x and y exists, it is unique up to isomorphism and is denoted by $x+y$.

proof

Let z be a sum of x and y with injections $i_x:x\to z$ and $i_y:y\to z$ and w another sum of x and y with injections $j_x:x\to w$ and $j_y:y\to w$. By the universal property of z, we can conclude that there is a morphism $k:z\to w$ such that $i_x;k=j_x$ and $i_y;k=j_y$. Applying the same reasoning to w, we conclude that there is a morphism $l:w\to z$ such that $j_x;l=i_x$ and $j_y;l=i_y$. Consider now the morphism $(k;l):z\to z$. It satisfies:

$i_x;(k;l)$

$=(i_x;k);l$	associativity of ;
$=j_x;l$	universal property of i_x
$=i_x$	universal property of j_x

Similarly, we can prove that $i_y;(k;l)=i_y$. The universal property of z, i_x and i_y implies that there is only one morphism $z\to z$ satisfying this property. Hence, $(k;l)=id_z$. Using the universal property of w, j_x and j_y we can prove in exactly the same way that $(l;k)=id_w$. Therefore k is an isomorphism.

4.2.3 Example – logical disjunction

1. In ***LOGI***, sums correspond to disjunctions.
2. In ***PROOF***, sums also correspond to disjunctions. Indeed, the sum of A and B is a sentence characterised by morphisms that capture the introduction and elimination rules for the disjunction as a logical operator:

$$i_A:A\to A\ \ B \qquad\qquad i_B:B\to A\ \ B$$

$$\frac{f_A:A\to C,\ f_B:B\to C}{[f_A,f_B]:A\ \ B\to C}$$

Notice that the conditions required of these three morphisms (derivations) correspond to properties of normalisation that are typical of proof-theory.

Other well-known properties of disjunction can be captured in such a "universal way":

4.2.4 Exercise

Let $\boldsymbol{C}$ be a category, x an arbitrary object and 0 an initial object of $\boldsymbol{C}$. Show that the sum $x+0$ exists and that i_x is an isomorphism, i.e. $x+0$ "is" x.

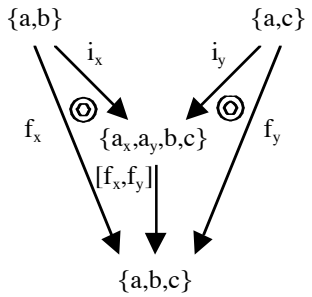
4.2.5 Example – disjoint union of sets

In the category ***SET***, the disjoint union $x \oplus y$ (with corresponding injections) is the sum of x and y.

proof

1. *existence*: consider an arbitrary object v and pair of morphisms $f_x: x \to v$, $f_y: y \to v$. Define $k: x \oplus y \to v$ as follows: given $A \in x \oplus y$, let $k(A)=f_x(a)$ if $A=i_x(a)$ with $a \in x$ and $k(A)=f_y(a)$ if $A=i_y(a)$ with $a \in y$. This is a proper definition of a total function because, on the one hand, every element of $x \oplus y$ is either in the image of x through i_x or the image of y through i_y and, on the other hand, these two images are disjoint (which removes any conflict of choice between which case to apply). The conditions $i_x;k=f_x$ and $i_y;k=f_y$ are satisfied by construction.
2. *uniqueness*: given any other total function $k': x \oplus y \to v$, the conditions $i_x;k'=f_x$ and $i_y;k'=f_y$ define k' completely (and equal to k).

In order to illustrate the construction and show why the union of sets is not (necessarily) their sum, consider the following example where f_x and f_y are set inclusions. The injections are such that $i_x(a)=a_x$ and $i_y(a)=a_y$. By construction, $[f_x,f_y]$ is such that $[f_x,f_y](a_x)=[f_x,f_y](a_y)=a$. The union $\{a,b,c\}$ does not provide a sum through the corresponding inclusions because there is no total function $l: \{a,b,c\} \to \{a_x,a_y,b,c\}$ that satisfies the commutativity conditions. Indeed, these require $l(a)=a_x$ because $a_x=i_x(a)$ and $l(a)=a_y$ because $a_y=i_y(a)$.



Intuitively, this happens because the union is an operation that "looks" inside the sets to which it is being applied in order not to repeat the elements that they have in common. That is to say, it is an operation that relies on an interaction between the sets to which it applies, whereas the sum is defined over objects without any relationship between them.

This construction illustrates how, in Category Theory, all interactions need to be made explicit and external to the entities involved. That is to say, we cannot rely on implicit relationships such as the use of the same names in the definition of different objects. This may sound too "bureaucratic" but we are far from suggesting that Category Theory should be used directly, "naked", as a language for specification,

modelling or, even, programming. We view Category Theory as a mathematical framework that provides support for those activities. Hence, the need for such explicit and external representation of all interactions is, in our opinion, a bonus because it enforces directly fundamental properties of the methodology that is being supported, service-oriented in the most general case, but also component or object-oriented as amply demonstrated in the literature.

In the next section, we address constructions that involve interactions. In the rest of this section, we look at the dual construction of sums: products. We do so extensionally, i.e. by providing the definition of the construction directly. The reason for doing so in spite of knowing already that it is enough to reverse the direction of the arrows, is that, from a "pragmatic" point of view, one is "naturally" biased by the direction of the arrows and, in certain contexts, tradition is that one uses products instead of sums. Our experience in using and teaching Category Theory is that big arguments and misunderstandings are too often caused by being presented with the duals of constructions that otherwise would be very familiar! Hence, it is important that we know these more basic constructions explicitly in both forms rather than having to translate back and forth every time between a category and its dual.

4.2.6 Definition – product

The dual notion of sum is *product*. That is to say, letting $\boldsymbol{C}$ be a category and x, y objects of $\boldsymbol{C}$, an object z is said to be a product of x and y with projections $\pi_x: z \to x$ and $\pi_y: z \to y$ iff for any object v and pair of morphisms $f_x: v \to x, f_y: v \to y$ of $\boldsymbol{C}$ there is a unique morphism $k: v \to z$ in $\boldsymbol{C}$ (often denoted by $\langle f_x, f_y \rangle$) such that $k;i_x = f_x$ and $k;i_y = f_y$.

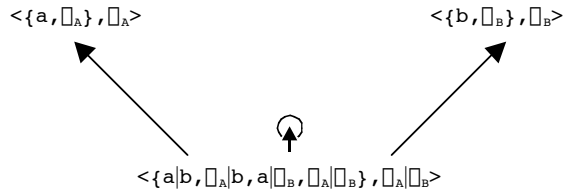
Products handle the relationships from the environment towards collections of two unrelated objects. It is easy to see that, in ***SET***, the product of two sets is given, up to isomorphism, by their Cartesian product. In ***LOGI***, products capture conjunction.

4.2.7 Example – parallel composition without interactions

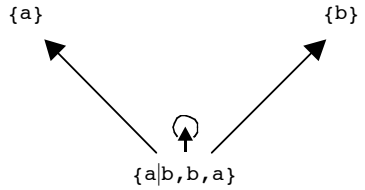
In the category $\boldsymbol{SET}_{\bullet}$ of pointed sets, products are constructed in the same way as in ***SET***. The Cartesian product of two pointed sets includes all pairs of "proper elements", the pairs of which one and only one of the elements is a designated one, and the pair of designated elements. We claim that, together with the projections, this Cartesian product is still a product of the pointed sets. On the one hand, it is trivial to prove that if we elect the pair of designated elements as the designated element of the product, we obtain through the original ***SET***-projections, morphisms of pointed sets. On the other hand, the commutativity requirements make sure that all universal functions $\langle f_x, f_y \rangle$ are morphisms

of pointed sets, i.e. preserve designated elements. We will return to this "style" of proof argument of universal properties for certain categories with "structure" later on in the book.

As an example, consider the use of pointed sets for modelling the alphabets of concurrent processes, as suggested in 3.2.1. Products of alphabets give us all the possible events in which both processes participate ($a|b$, as a representation of the pair $<a,b>$, in the case below), plus all possible events in which only one process participates ($\lambda_A|b$ and $a|\lambda_B$ in the case below), and the "silent" environment event in which none of the processes participates ($\lambda_A|\lambda_B$ in the case below).



In order to simplify the notation, both textual and graphical, we normally hide the designated element as in



Notice in particular that the events in which only one process participates are represented by the event of that process.

In the specific case in which we identify events with synchronisation sets of method execution, the product provides for joint executions as unions of synchronisation sets. To be more precise, when we work in ***POWER*** (see 3.3.2), we can take as a representation of a pair $<a,b>$ the set $a \cup b$. This is because, as sets, the Cartesian product $2^A \times 2^B$ is isomorphic to $2^{A \oplus B}$ and the isomorphism is given, precisely, by the map that associates pairs $<a,b>$ with sets $a \oplus b$. This is to show that, because universal constructions are only unique up to isomorphism, we can choose from the isomorphism class the representation that suits us best. In this case, it justifies a uniform treatment of concurrent process alphabets as synchronisation sets.

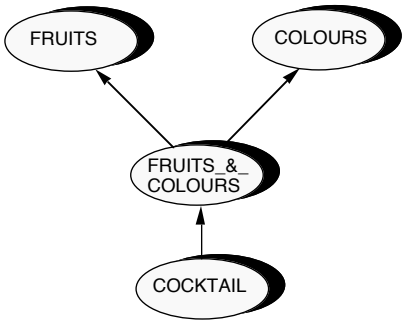
In summary, through products, we obtain the alphabet of the process that is the interleaving of the given ones.

4.2.8 Exercise

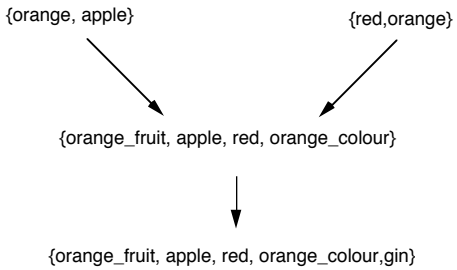
Characterise products and sums of partial functions (i.e. in ***PAR***). Show in particular that sums work like in ***SET***, i.e. they compute the disjoint union, but that products, besides the pairs that result from the Cartesian product, include the disjoint union of the two sets as well. That is to say, products in ***PAR*** are very "similar" to those in $\mathbf{SET}_{\square}$. Why do you think this is so? What about sums in $\mathbf{SET}_{\square}$? How do they compare?

4.2.9 Example – inheritance without name clashes

In Eiffel, products capture "minimal" inheritance without name clashes, a typical example being the following:



We know already that, when the sets of features of the classes in an inheritance graph are considered, the arrows are reversed, so the universal construction on the underlying features is a sum:



The “automatic renaming” associated with this categorical construction is very useful because it makes sure that no confusion arises from the inadvertent use of the same names for different “things” in different contexts. This is why the injections/projections are very much part of the concept of sum/product: they keep a record of the renamings that take place, i.e. of “who is who” or, better, “who comes from where”. However, in many situations, we want to make “joins”, i.e. identify things that are meant to be the same but were included in dif-

ferent contexts. This is the purpose of the universal construction that we illustrate next.

4.3 Pushouts and pullbacks

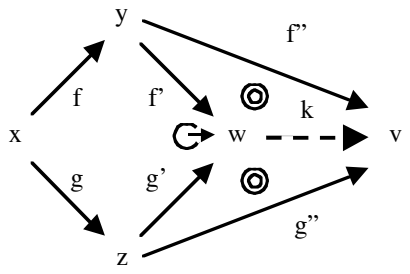
We already know that, in Category Theory, by default, things are different even if they were given the same names in different contexts. How, then, can we say that some things are the same? For instance, how to require that two processes synchronise on given events? How to require that features declared in different object classes be identified when performing multiple inheritance?

A distinguish factor of Category Theory, and one that makes it so suitable for addressing architectural concerns and service-oriented development in Software Engineering, is that such forms of interaction are *exogenous*, i.e. they have to be established *outside* the objects involved. For instance, interactions in object-oriented development are endogenous because feature calling (clientship) is embedded explicitly in the code of the caller (client). In Category Theory, the means that we have for establishing interactions is through third objects that handle communication between the ones that are being interconnected via given morphisms. In the case of sums, we are dealing with interactions at the level of the sources, i.e. we are interested in the social life of pairs of morphisms $f:x \to y$, $g:x \to z$, and in the case of products, the interactions are on the target side – $f:y \to x$, $g:z \to x$.

4.3.1 Definition – pushout

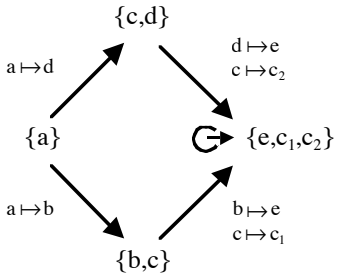
Let $\mathbf{C}$ be a category and $f:x \to y$, $g:x \to z$ morphisms of $\mathbf{C}$. A pushout (or amalgamated sum) of f and g consists of two morphisms $f':y \to w$ and $g':z \to w$ such that

- $f;f'=g;g'$
- for any other two morphisms $f'':y \to v$ and $g'':z \to v$ such that $f;f''=g;g''$, there is a unique morphism $k:w \to v$ in $\mathbf{C}$ such that $f';k=f''$ and $g';k=g''$.



4.3.2 Example – amalgamated sums of sets

In ***SET***, pushouts perform what are usually called "amalgamated sums", i.e. they allow us to identify (join) elements as indicated by the "middle object" and corresponding morphisms. For instance, in the example below, the morphisms indicate that elements *b* and *d* are to be identified. Because nothing is said about *c*, the categorical default applies: the two occurrences are to be distinguished because the fact that the same name was used is accidental.



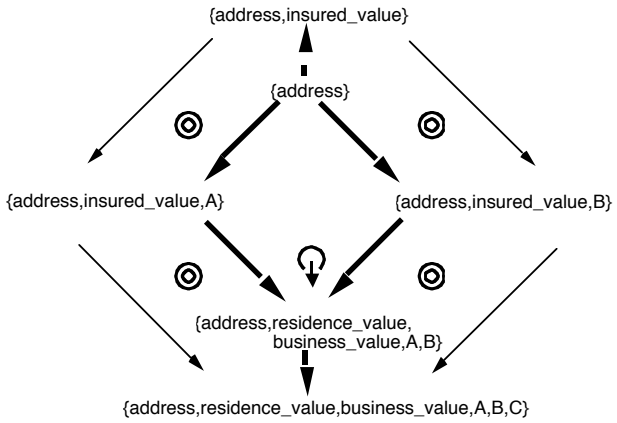
The proof that what we have built is actually a pushout can be outlined as follows. On the one hand, the commutativity requirement is clearly satisfied. On the other hand, for the universal property, consider given functions $f:\{c,d\} \to A$ and $g:\{b,c\} \to A$ satisfying the commutativity requirement: $f(d)=g(b)$.

Any function $k:\{e,c_1,c_2\} \to A$ that satisfies the commutativity requirements of the universal property must be such that $k(e)=f(d)$, $k(e)=g(b)$, $k(c_1)=g(c)$ and $k(c_2)=f(c)$. These requirements leave no other choice for defining k, hence uniqueness is ensured. Because these requirements are consistent, which is due to the fact that $f(d)=g(b)$, existence is also ensured.

More interesting than this proof is the mechanism through which pushouts can be systematically constructed in ***SET***. This will be revealed once we address the whole process of computing pushouts in more general terms.

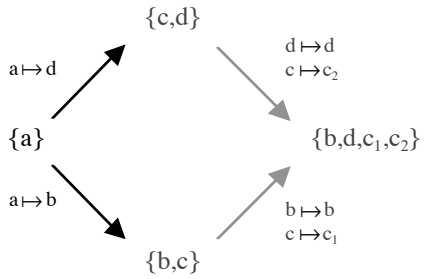
4.3.3 Example – multiple inheritance in Eiffel

Pushouts are the universal construction that allows us to join (merge) features during multiple inheritance. For instance, the construction of *HOME_BUSINESS* requires that the home and business addresses be merged:



Let us now return to the definition of pushout and explain how the interactions operate. The definition consists of two main requirements: a commutativity condition and a "universal" property. The universal property is just an instance of a "ritual" that we shall explain in the next section. For the time being, notice how similar it is to the corresponding property of sums, and try to see where it lies in the definition of initial object…

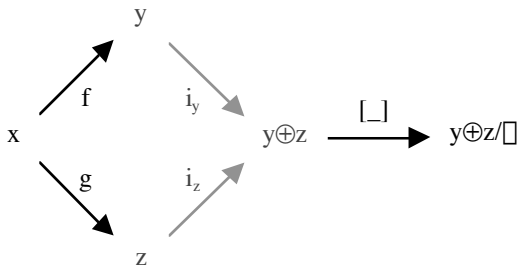
The commutativity requirement is the one responsible for the "amalgamation" or, more generally, for encapsulating the interactions in place. In the case of ***SET***, the amalgamation takes place as a quotient defined over the sum of the objects for an equivalence relation that is defined by the middle object and the connecting morphisms. If we take two morphisms $f: x \to y$, $g: x \to z$ and we can find a sum for y and z, we obtain a square like before except for the fact that it does not necessarily commute. For instance,



Commutativity fails because, if we pick a in the middle object and follow all the paths from it, we do not arrive at the same element: one, via b, terminates in c_1 and the other, via d, in c_2.

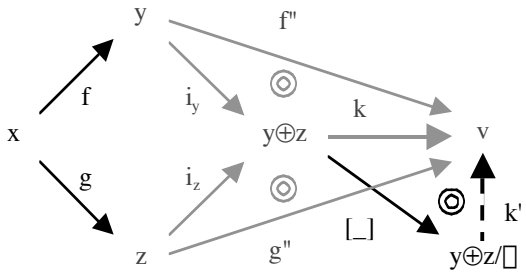
In order to enforce the commutativity requirement, we have to "equalise" the ends of these paths. Category Theory does not perform

miracles (yet), so it cannot make equal entities that are actually different… What we usually do in Mathematics is to define an equivalence relation expressing that, albeit being different, these endpoints should be considered the same as far as the "system" (as expressed by the diagram) is concerned. In ***SET***, this equivalence relation is formally defined as being generated from all pairs $f(i_y(a)) \sim g(i_z(a))$ where $a \in x$. We claim that the quotient set of the sum $y \oplus z$ by this equivalence relation together with the functions $[i_y(\_)]$ and $[i_z(\_)]$ that assign, to the elements of the given sets y and z, their equivalence classes, is a pushout of f and g.



The proof of this result is not that difficult because the whole construction was made to ensure commutativity, i.e. $[i_y(f(a))]=[i_z(g(a))]$ for every $a \in x$, which is precisely the set of pairs $i_y(f(a)) \sim i_z(g(a))$ that generate the equivalence relation. The universal property can now be derived from the universal properties of the sum (in grey in the figures) and quotients:

1. Consider any other two morphisms $f'':y \to v$ and $g'':z \to v$ such that $f;f''=g;g''$.

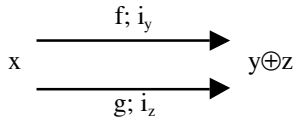


2. We know from the universal property of the sum that there is a unique morphism $k:y \oplus z \to v$ such that $i_y;k=f''$ and $i_z;k=g''$.
3. The morphism k also equalises the pairs $\langle i_y(f(a)), i_z(g(a))\rangle$; indeed,

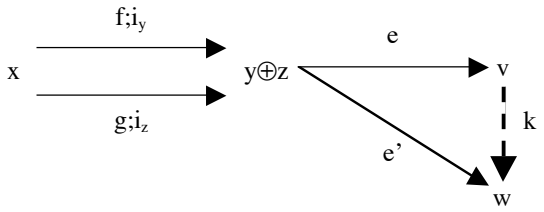
$k(i_y(f(a)))$
$=f''(f(a))$ from 2
$=g''(g(a))$ from 1
$=k(i_z(g(a)))$ from 2

4. From the properties of quotients, we know that there is a unique way in which this map can be factorised through *[_]* i.e. there is a unique $k':y\oplus z/\!\!\sim \to v$ such that $k=[\_];k'$. Using associativity of morphism composition, this provides us with the required unique morphism satisfying $(i_y;[\_]);k'=f$" and $(i_z;[\_]);k'=g$".

This is how amalgamated sums work on sets. How can we generalise the quotient to categories in general? Taking the diagram above to be over an arbitrary category $\boldsymbol{C}$, the purpose of the quotient is to make the following diagram commute:



The idea is to do so via a third morphism $e:y\oplus z \to v$ whose purpose is to replace the initial equality $f;i_y=g;i_z$ by $(f;i_y);e=(g;i_z);e$. However, there are many ways of doing so. One of them is to choose v to be a terminal object (if one exists), but this is clearly too intrusive on the given morphisms because it over-equalises them. We would prefer to do it in a minimal way, which is what the universal property of quotients provides: for any other morphism $e':y\oplus z \to w$ that also equalises the morphisms, i.e. such that $(f;i_y);e'=(g;i_z);e'$, there should be a unique $k:v \to w$ such that $e;k=e'$.



This is precisely another instance of a universal construction:

4.3.4 Definition – co-equaliser

Let $\boldsymbol{C}$ be a category and $f:x \to y$, $g:x \to y$ morphisms of $\boldsymbol{C}$. A *co-equaliser* of f and g consists of a morphisms $e:y \to z$ such that

- $f;e=g;e$
- for any other morphisms $e':y \to v$ such that $f;e'=g;e'$, there is a unique morphism $k:z \to v$ in $\boldsymbol{C}$ such that $e;k=e'$.

Hence, pushouts can be obtained from sums and co-equalisers.

4.3.5 Exercises

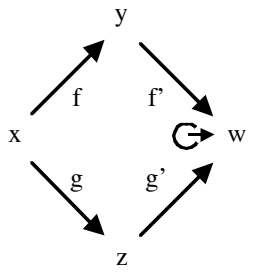
1. Formalise the observation above about pushouts being obtained from sums and co-equalisers and prove it.
2. Prove that co-equalisers are a particular case of pushouts.

3. Prove that, if initial objects exist, sums can be obtained from pushouts.

There are a number of properties that are both typical and useful. They are left here as exercises, with a strong recommendation.

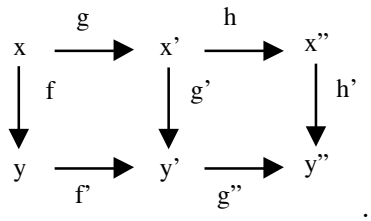
4.3.6 Exercises

1. Prove that the universal arrow e in a co-equaliser is epic (thus generalising the fact that a quotient map is surjective).
2. Consider a pushout diagram



Prove that, if g is epic, so is f'.

3. Consider a commutative diagram



Prove that if both internal squares are pushouts so is the external rectangle, and that if the external rectangle and the left square are pushouts, so is the right square.

Let us consider now the dual of pushouts.

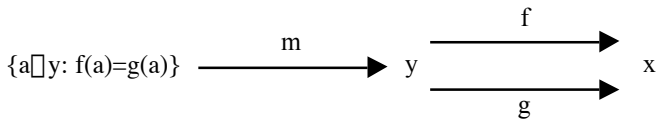
4.3.7 Definition – pullback

Let $\boldsymbol{C}$ be a category and $f:y \to x$, $g:z \to x$ morphisms of $\boldsymbol{C}$. A *pullback* (or fibred product) of f and g consists of two morphisms $f':w \to y$ and $g':w \to z$ such that

- $f';f=g';g$
- for any other two morphisms $f'':v \to y$ and $g'':v \to z$ such that $f'';f=g'';g$, there is a unique morphism $k:v \to w$ in $\boldsymbol{C}$ such that $k;f'=f''$ and $k;g'=g''$.

Pullbacks can also be explained from products and a construction that equalises arrows, except that, this time, the equalising needs to be made on the source and not the target side of the arrows. Not surprisingly, this universal construction is named equaliser, the dual of the co-equalisers.

Whereas, in set-theoretic terms, we saw that equalising on the target side corresponds to taking a quotient to group entities that should be "equal" in the same equivalence relation, equalising on the source side is even conceptually easier: it is enough to restrict the domain by throwing away the elements over which the two functions disagree. That is, we equalise through the inclusion $\{a \in y: f(a)=g(a)\} \hookrightarrow y$.



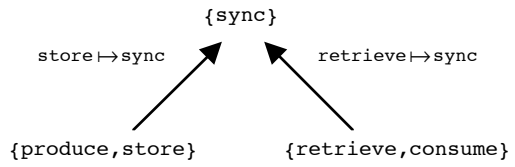
Hence, computing a fibred product consists in computing a product followed by a "purge" of the pairs that violate the commutativity requirement.

We are going to illustrate this procedure with process alphabets.

4.3.8 Example – parallel composition with interactions

We have already argued that in the category $\boldsymbol{SET}_{\bullet}$ of pointed sets, products are constructed in the same way as in ***SET*** through Cartesian products. When the pointed sets capture process alphabets, we saw that this construction captures parallel composition without synchronisation in the sense that all the pairs of events are generated in an interleaving semantics. Fibred products allow us to compute parallel composition with synchronisation constraints.

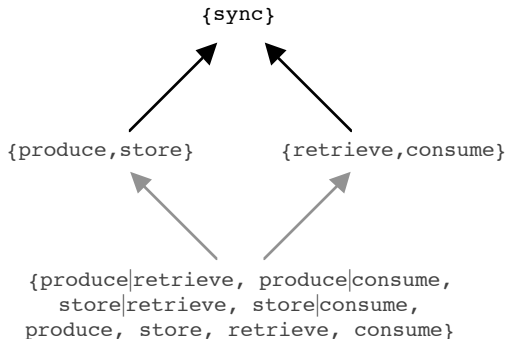
For instance, consider two process alphabets $\langle \{produce, store, \bullet_P\}, \bullet_P \rangle$ and $\langle \{consume, retrieve, \bullet_C\}, \bullet_C \rangle$. The alphabet of the parallel composition of the two processes when required to synchronise in the *store* and *retrieve* events can be obtained through the pullback of:



Notice that, as mentioned in 4.2.7, and for simplicity, we have omitted the designated elements from the representations of the pointed sets and will only show the proper events. The middle object models the alphabet of the "cable" that is being used to interconnect the two proc-

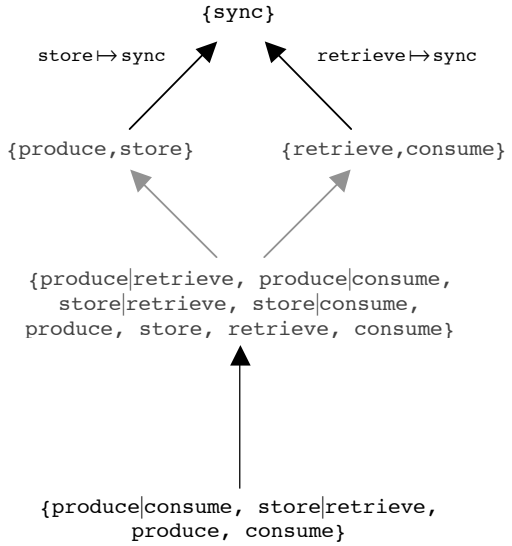
esses. The maps mean that *store* and *retrieve* are being synchronised, *sync* providing the place of their "rendez-vous".

The product of the two alphabets gives us:



Recall that the events of the given processes are recovered as a result of synchronisations with the designated event, thus capturing system events in which only one of the two processes participates.

Notice that the diagram does not commute. For instance, the system event *store|consume* is mapped, on the left, to *sync* but, on the right, to the designated event of the cable. In order to make the diagram commute, we throw away all system events on which the maps to the channel do not agree:



Basically, the events that remain are all possible combinations of executing *produce* and *consume* because there is no synchronisation constraint on them, plus the synchronisation that is explicitly required.

4.3.9 Exercise

Follow-up on 4.2.8 by characterising fibred products and amalgamated sums of partial functions (i.e. in ***PAR***) and relating them to ***SET*** and $\boldsymbol{SET}_{\bullet}$.

4.3.10 Exercises

1. Define explicitly the notion of equaliser and prove that pullbacks can be obtained from products and equalisers as suggested.
2. Prove that equalisers are a particular case of pullbacks.
3. Prove that, if terminal objects exist, products can be obtained from pullbacks.
4. Prove that the equalising arrow m is a mono.

4.4 Limits and colimits

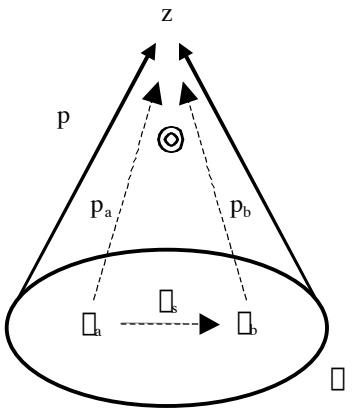
It should be clear by now that the notion of "collective behaviour" that we wish to capture through universal constructions takes diagrams as the expression of the collection of objects and interactions that constitute what we could call a "system". In fact, we tend to use diagrams for dealing with "complex" entities for which the objects of the category provides components and the morphisms the means for interconnecting them. Hence, for instance, a typical use of diagrams is for defining configurations. The universal constructions that we are addressing in this chapter allow us to define the semantics of such complex entities by internalising the configuration and collapsing the structure into an object that captures the collective behaviour.

An aspect of these universal constructions that is important to keep in mind is the fact that they deliver more than an object: this object comes together with morphisms that relate it to the objects out of which it was constructed. It is through these morphisms that we can understand how properties of the system (complex object) emerge from the properties of its components and the interconnections between them. Hence, the constructions are better understood in terms of structures that consist of objects together with configurations to which they relate. These are called (co)cones.

4.4.1 Definition – co-cone

Let $\Delta: I_0 \to \boldsymbol{C}$ be a diagram in a category $\boldsymbol{C}$. A co-cone with base Δ is an object z of $\boldsymbol{C}$ together with a family $\{p_a: \Delta_a \to z\}_{a \in I_0}$ of morphisms of $\boldsymbol{C}$, usually denoted by $p: \Delta \to z$. The object z is said to be the vertex of the co-cone, and, for each $a \in I_0$, the morphism p_a is said to be the edge of

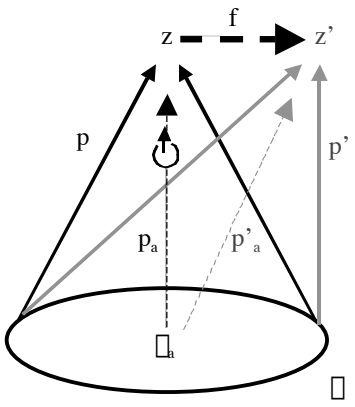
the co-cone at point a. A co-cone p with base $\Delta:I\to \mathbf{C}$ and vertex z is said to be commutative iff for every arrow $s:a\to b$ of graph I, $\Delta_a;p_b=p_a$.



We tend to consider the family p of morphisms as identifying how the source is represented in, or a component of, the target – the object z. The commutativity property is important because it ensures that the interconnections that are expressed in the base through the morphisms are also represented in z. Hence, z is an object that is able to "represent" the base objects and their interactions. However, it may not do so in a "minimal" way. If one such minimal representation exists, we call it a colimit of the diagram.

4.4.2 Definition – colimit

Let $\Delta:I\to \mathbf{C}$ be a diagram in a category $\mathbf{C}$. A colimit of Δ is a commutative co-cone $p:\Delta\to z$ such that, for every other commutative co-cone $p':\Delta\to z'$, there is a unique morphism $f:z\to z'$ such that $p;f=p'$, i.e. $p_a;f=p'_a$ for every edge. Colimit co-cones are decorated with



Notice how all the ingredients that were used in the universal constructions that we studied in the previous sections are present in this definition. On the one hand, a commutativity requirement. On the other hand, a universal property that ensures minimality.

4.4.3 Exercise

Show that initial objects, sums, co-equalisers and pushouts are instances of colimits by identifying the shape of the base diagrams and checking that the properties required are equivalent.

Co-cones over a given base can be organised in a category in an "obvious way":

4.4.4 Proposition

Let $\Delta : I \to \boldsymbol{C}$ be a diagram in a category $\boldsymbol{C}$. A category $\boldsymbol{CO\_CONE}(\Delta)$ is defined whose objects are the commutative co-cones with base Δ and the morphisms f between co-cones $p : \Delta \to z$ and $q : \Delta \to w$ are the morphisms $f : z \to w$ such that $p ; f = q$, i.e. $p_a ; f = q_a$ for every edge.

4.4.5 Exercises

1. Prove the previous result.
2. Prove that the colimits of a diagram Δ are the initial objects of $\boldsymbol{CO\_CONE}(\Delta)$.
3. Conclude that colimits are unique up to isomorphism.

4.4.6 Definition – co-completeness

A category is (finitely) co-complete if all (finite) diagrams have colimits.

There are several results on the finite co-completeness of categories. A commonly used one is:

4.4.7 Proposition

A category $\boldsymbol{C}$ is finitely co-complete iff it has initial objects and pushouts of all pairs of morphisms with common source.

Using this result and the examples that we studied in the previous sections, we can conclude, for instance, that both ***SET*** and ***LOGI*** are finitely co-complete. These notions are also straightforward generalisations of well-known constructions over ordered sets, namely least upper bounds and greatest lower bounds.

A more "interesting" example can be given over Eiffel class specifications:

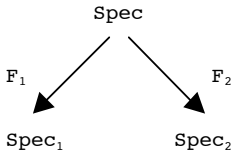
4.4.8 Example – Eiffel's "join semantics" rule

The category ***CLASS_SPEC*** is finitely co-complete.

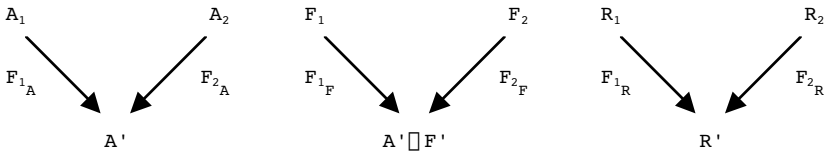
proof

As it happens so many times, the proof is much more interesting than the result. Using the previous result, we show that ***CLASS_SPEC*** admits initial objects and puhouts.

1. It is easy to prove that the class specification that has no features is initial.
2. On the other hand, pushouts work as follows:
   - given a diagram



     we first compute the pushout for the underlying diagram of signatures (sets of attributes, functions, and routines). This pushout returns, for each feature



     Recall that, according to the definition of morphism given in section 3.4, functions can be mapped to attributes (but not the other way around)! Hence, equivalence classes can mix together attributes and functions. The pushout classifies an equivalence class as an attribute iff it contains at least one attribute, and as a function iff it only contains functions. Routines are not mixed together with functions or attributes.
   - for every routine $r' \in R'$, we compute its pre/post-conditions as follows
     if $r'=F_{1R}(r_1)=F_{2R}(r_2)$ then
     $pre_r = pre_{r_1} \vee pre_{r_2}$ and $post_r = post_{r_1} \wedge post_{r_2}$
     if $r'=F_{1R}(r_1) \in F_{2R}(R_2)$ then $pre_r = pre_{r_1}$ and $post_r = post_{r_1}$
     if $r'=F_{2R}(r_2) \in F_{1R}(R_1)$ then $pre_r = pre_{r_2}$ and $post_r = post_{r_2}$
   - the new invariant is $I' = F_1(I_1) \wedge F_2(I_2)$.

The proof that the maps F_1 and F_2 that result from these constructions are indeed morphisms of class specifications is trivial. The commutativity property is inherited from the pushouts in ***SET*** that determine the new attributes and routines. The universal property is left as an exercise and, basically, reflects the universal properties of conjunction and disjunction that we already identified in the previous sections.

Note how this construction matches the *Join Semantics rule* and inheritance of invariants via concatenation of parent invariants [88].

4.4.9 Exercise

Workout in detail the way pushouts operate on attributes and functions of class specifications.

4.4.10 Definition

The dual notion of co-cone is *cone* and the dual notion of colimit is *limit*. Limit cones are decorated with

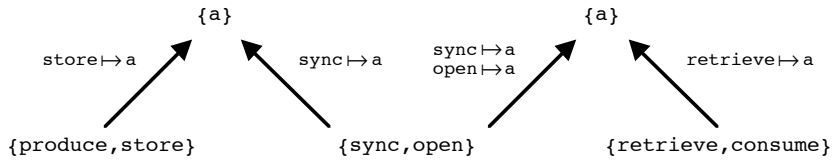
Hence, limits generalise terminal objects, products, equalisers and pullbacks. From the examples studied in the previous sections, we can also conclude that both $\boldsymbol{SET}$ and $\boldsymbol{SET}_{\square}$ are finitely complete.

4.4.11 Example – parallel composition with interactions

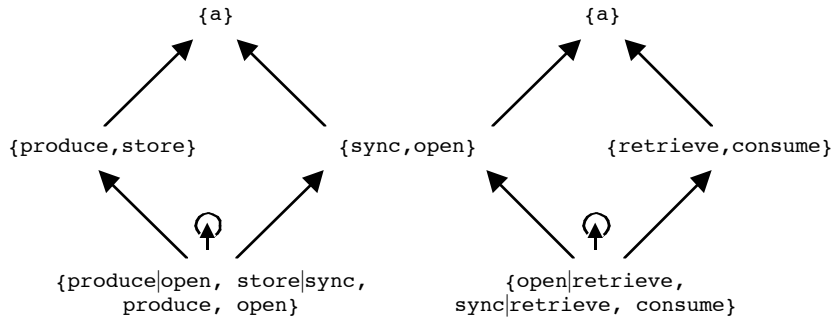
In order to illustrate the calculation of limits, consider process alphabets once again. In the previous section, we showed how we can synchronise a consumer and a producer on the *store*/*retrieve* events. This interconnection is, however, too tight because it ties the consumer completely to that producer and does not allow it to consume from other producers. Hence, the situation that we would like now to model is the one in which the consumer retrieves from a producer but leaves open the possibility of retrieving from other producers as well. This form of interconnection cannot be achieved simply through a channel as before.

What we have to do is to make explicit a communication protocol that can be placed in between the producer and the consumer. The alphabet of this protocol needs to account for the synchronisation between the consumer and the producer, which we model through an event *sync*, and the open communication between the consumer and other possible producers, which we model through an event *open*.

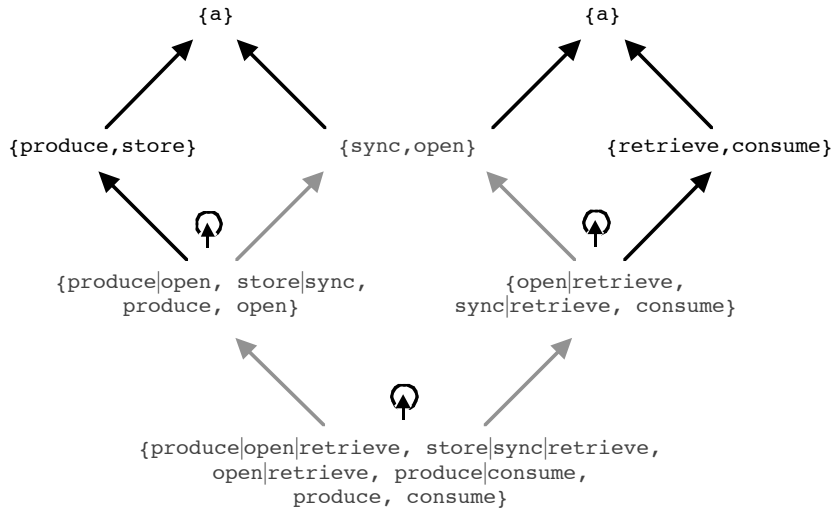
Omitting, as before, the designated events, this configuration is given by the following diagram:



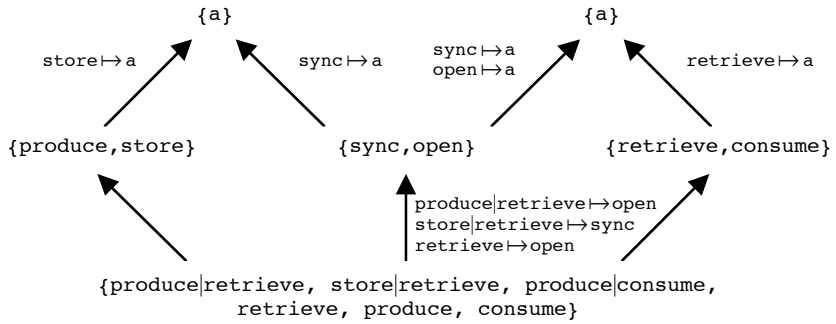
The dual of 4.4.7 can be used to compute the limit of this diagram by computing successive pullbacks. We start with the two obvious ones:



We finalise with the pullback of the new arrows, which provides for the global interaction:



Using the privilege of choosing among the isomorphic representations of the limit the one that best reflects our intuition or purpose, we can simplify the notation. For instance, we can omit the reference to the actions of the communication protocol in the synchronisation sets, although at the expense of making the morphisms less obvious because they are no longer projections. Hiding the intermediate constructions at the same time, we obtain the following commutative cone:



This representation makes it obvious that, with respect to the interconnection given in 4.3.8, we are now allowing the *retrieve* to occur independently of *store*, which justifies why it now appears as a single event (though synchronised with *open*) and synchronised with *produce* (and *open*).

Although we have systematically motivated universal constructions such as these in terms of parallel composition of processes, with and without interaction, we have remained at the level of the composition of process alphabets, i.e. we have not taken into account their behaviours. There are two reasons for that. On the one hand, as we shall see, all the complexity of interaction resides on alphabets. On the other hand, the way behaviours can be brought into the picture can be used to illustrate another categorical structure and construction. Hence, we defer the "completion" of this topic to later on (6.3.2 and 6.3.7).

For similar reasons, we defer the illustration of the application of universal constructions to configurations of specifications to later on (6.1.24) so that we can use it to illustrate another class of categorical structures and constructions.

5 Functors

5.1 The social life of categories

It should come to no surprise that we are also interested in the social life of categories and, therefore, need to define a corresponding notion of morphism. Indeed, we have said more than once that, when defining a category, we are capturing only a particular view or structure of the underlying objects. There may well be more than one such notion of structure for the same collection of objects. For instance, in the case of object-oriented development, we have already mentioned the structure that relates to the way objects can be interconnected into systems, and the view that captures refinement. There may also be relationships that we wish to study between two different domains, for instance Eiffel specifications and processes. Hence, we need a way of reasoning about relationships between categories.

Because categories are structured objects, more precisely, graphs with the additional structure given by the identity arrows and the composition law, the definition of morphisms between categories (functors) will now look "standard":

5.1.1 Definition – functor

Let ***C*** and ***D*** be categories. A *functor* $\Phi: \mathbf{C} \to \mathbf{D}$ is a graph homomorphism (2.1.7) from ***graph(C)*** into ***graph(D)*** such that:

- $\Phi_1(f;g) = \Phi_1(f);\Phi_1(g)$ for each path fg in ***graph(C)***
- $\Phi_1(id_x) = id_{\Phi_0(x)}$ for each node x in ***graph(C)***.

5.1.2 Examples

1. For any category ***C***, the identity functor $\mathbf{id_C}: \mathbf{C} \to \mathbf{C}$ consists of the identity map on objects and the identity map on morphisms.
2. For any given functor $\Phi: \mathbf{C} \to \mathbf{D}$, its dual is $\Phi^{op}: \mathbf{D}^{op} \to \mathbf{C}^{op}$ is defined by the graph homomorphismthat operates the same mappings on nodes and arrows.
3. We define a functor ***nodes***: ***GRAPH*** $\to$ ***SET*** as follows:

- ***nodes***$(G_0,G_1,src,trg)=G_0$
- ***nodes***$(\alpha:G \to H)= \alpha_0:G_0 \to H_0$

4. We define a functor ***PROOF*** $\to$ ***LOGI*** by mapping sentences to themselves and every proof between two sentences to a logical implication between them. This is an example of a *forgetful functor*: the details of the proof are forgotten; only the fact that there exists a proof remains.
5. Another example of a forgetful functor is ***sign****:* $\mathbf{PRES}_{LTL} \to$ ***SET*** mapping presentations and their morphisms to the underlying signatures and signature morphisms. A similar forgetful functor is obtained for $\mathbf{THEO}_{LTL}$.
6. We define a functor ***ANCESTOR*** $\to$ ***SET***^{op} by mapping classes to the set of their features and inheritance relationships to the functions that rename the features. Notice that the target category is the dual of ***SET***. This is because, as we have illustrated in 2.1.3, the edges of the inheritance graph and the corresponding graph of features have opposite directions. Functors of the form $\alpha:\mathbf{C} \to \mathbf{D}^{op}$ (or $\alpha:\mathbf{C}^{op} \to \mathbf{D}$) are sometimes called *contravarian*t between ***C*** and ***D***. "Normal" functors are then called *covariant*.

5.1.3 Example – relating programs and specifications

In Computing, functors are often used to express relationships between different levels of abstraction, namely by providing a way of abstracting properties from representations. In this context, the abstractions are sometimes called "specifications", and the representations "programs". For instance, although we have worked with specifications of Eiffel classes and not with the programs that implement the methods of such classes, it is clear that such specifications are at a lower level of abstraction than the temporal specifications that we defined in section 3.5. For instance, the temporal specifications can be used to describe general properties of the possible behaviours of a system whereas Eiffel specifications are concerned with more specific operational aspects of the execution of actions, like their effects on the attributes.

The notion of a program satisfying a specification is typically formalised through a (satisfaction) relation. When specifications consist of sets of logical formulas such as those defined in section 3.5, the collection of specifications that are satisfied by a given program is ordered by inclusion and has a maximum: the union of all the sets of properties satisfied by the program. In this case, we can assign to every program *P* a "canonical" specification ***spec****(P)* – its strongest specification.

We have also seen that morphisms between models of system behaviour can be used to capture notions of simulation or refinement. When ***spec*** is a functor, this means that the refinement relationship defined on

"programs" is captured by the corresponding notion of refinement at the level of specifications. When specifications are given as theory presentations, this means that program refinement is property-preserving, which is what one usually expects from a refinement relation. Hence, there is a natural way in which the satisfaction relation between programs and specifications can be expected to be functorial.

As an example, consider the following mapping ***spec*** from ***CLASS_SPEC*** to $\boldsymbol{PRES}_{FOLTL}$, where by $\boldsymbol{PRES}_{FOLTL}$ we are denoting the category of theory presentations for the first-order extension of the temporal logic presented in 3.5.4[8]:

- the image of every class signature is itself. That is, the features of the class are taken as symbols of the vocabulary of the logic.
- for every routine *r*, its specification *(pre,pos)* is mapped to the temporal formula

  $$(r \Rightarrow pre \Rightarrow att \quad \mathbf{X}pos^{*})$$

  By *att* we denote the conjunction

  $$\bigwedge_{a \in att(\Sigma)} (a = x_a)$$

  where the variables x_a are all new, and by pos^{*} we denote the formula that is obtained from the expression *pos* by replacing every occurrence of every expression ***(old** a)*, where *a* is an attribute, by the variable x_a.
- the invariant *I* is mapped to itself.

For instance, the specification of the bank account given in 3.5.6 is mapped to the following set of temporal formulas:

$$\texttt{deposit(i)} \Rightarrow \texttt{balance=}x_{\texttt{balance}} \quad \mathbf{X}\texttt{(balance=}x_{\texttt{balance}}\texttt{+i)}$$
$$\texttt{withdrawal(i)} \Rightarrow \texttt{balance} \geq \texttt{i} \wedge \texttt{balance=}x_{\texttt{balance}} \quad \mathbf{X}\texttt{(balance=}x_{\texttt{balance}}\texttt{−i)}$$
$$\texttt{vip} \quad \texttt{balance} \geq \texttt{1000}$$

To show that we obtain, indeed, a functor, we have to show in particular that morphisms of class specifications are property preserving. Consider a morphism $F\colon e=\langle \Sigma,P,I\rangle \Rightarrow e'=\langle \Sigma',P',I'\rangle$ of class specifications.

- Given any routine *r* of Σ, its image *F(r)* is such that its specification *(pre',pos')* is mapped to: $(F(r) \Rightarrow pre' \Rightarrow att' \quad \mathbf{X}pos'^{*})$. We know that

  1. $F(pre) \vdash pre'$ because *F* is a morphism
  2. $pos' \vdash F(pos)$ because *F* is a morphism
  3. $att' \quad \mathbf{X}pos'^{*} \vdash F(att) \quad \mathbf{X}F(pos^{**})$ from 2

  from which we can conclude that

[8] For simplicity, and because the properties of the categories for the propositional and first-order versions of temporal logic in which we are interested are the same, we omit the extension. The reader interested in the logic itself can consult [66] as well as [39] for its use in a categorical framework.

$(F(r)(pre'(att' \quad Xpos'^{*}) \vdash F(r(pre(att \quad Xpos^{*})$

- By definition of morphism, we also have $I' \vdash F(I)$

Hence, every axiom of ***spec***(Σ) is translated through F to a theorem of ***spec***(Σ').

It should come to no surprise that even categories can be organised in a category:

5.1.4 Definition/proposition – the category of categories

1. Let $\Phi:\boldsymbol{C} \to \boldsymbol{D}$ and $\Psi:\boldsymbol{D} \to \boldsymbol{E}$ be functors. By $\Phi;\Psi$ we denote the functor defined by $(\Phi;\Psi)_0=\Phi_0;\Psi_0$ and $(\Phi;\Psi)_1=\Phi_1;\Psi_1$. This law of composition is associative and admits identities as defined in 5.1.2.1.
2. We can thus define the category ***CAT*** whose objects are the categories and whose morphisms are the functors.

proof

The proofs are trivial. However, bear in mind that, in defining ***CAT***, there are problems of "size" in the same way as we alerted at the beginning of section 2.1. Again, see a more "mathematical-oriented" book on Category Theory, e.g. [79], for such matters.

Having a notion of morphism between categories, we can apply to categories and functors all the machinery that we have so far presented for manipulating objects and their morphisms. For instance, the notion of product that we defined in section 3.1 corresponds to a universal construction in the category ***CAT***.

5.1.5 Definition/proposition – product of categories

Given categories $\boldsymbol{C}$ and $\boldsymbol{D}$, we define functors $\Pi_C:\boldsymbol{C}\times\boldsymbol{D} \to \boldsymbol{C}$ and $\Pi_D:\boldsymbol{C}\times\boldsymbol{D} \to \boldsymbol{D}$, called the projections of the product, by mapping objects and morphisms of $\boldsymbol{C}\times\boldsymbol{D}$ to their components in $\boldsymbol{C}$ and $\boldsymbol{D}$, respectively. These functors satisfy the universal property of functors in the following sense: given any category $\boldsymbol{E}$, and functors $\Phi_C:\boldsymbol{E} \to \boldsymbol{C}$ and $\Phi_D:\boldsymbol{E} \to \boldsymbol{D}$, there is a unique functor $\Phi:\boldsymbol{E} \to \boldsymbol{C}\times\boldsymbol{D}$ such that $\Phi_C=\Phi;\Pi_C$ and $\Phi_D=\Phi;\Pi_D$. We normally denote Φ by $\langle\Phi_C,\Phi_D\rangle$.

We can also define a product over functors:

5.1.6 Definition/proposition – product of functors

Given functors $\Phi_1:\boldsymbol{C}_1 \to \boldsymbol{D}_1$ and $\Phi_2:\boldsymbol{C}_2 \to \boldsymbol{D}_2$, their product $\Phi_1\times\Phi_2:\boldsymbol{C}_1\times\boldsymbol{D}_1 \to \boldsymbol{C}_2\times\boldsymbol{D}_2$ is defined by $(\Phi_1\times\Phi_2)\langle c_1,c_2\rangle=\langle\Phi_1(c_1),\Phi_2(c_2)\rangle$ on objects and $(\Phi_1\times\Phi_2)\langle f_1,f_2\rangle=\langle\Phi_1(f_1),\Phi_2(f_2)\rangle$ on morphisms.

Functors come in all “shapes and sizes” and, as morphisms between categories, they provide the means for characterising the structural properties of categories in the way they relate to other categories. Hence, it is useful to study the properties of functors and the way they

allow us to reveal the structure of categories. We start with some elementary properties that result from the "functional" nature of functors as mappings between the sets of nodes and arrows.

5.1.7 Definition

Let $F:\mathbf{C}\to\mathbf{D}$ be a functor.

1. F is an *isomorphism* iff there is a functor $G:\mathbf{D}\to\mathbf{C}$ such that $F;G=id_C$ and $G;F=id_D$. In this case, $\mathbf{C}$ and $\mathbf{D}$ are said to be *isomorphic* and G to be the *inverse* of F.
2. F is an *embedding* iff F_1 is injective, i.e. F is injective on morphisms.
3. F is *faithful* iff all the restrictions $F_1:hom_C(x,y)\to hom_D(F_0(x),F_0(y))$ are injective.
4. F is *full* iff all the restrictions $F_1:hom_C(x,y)\to hom_D(F_0(x),F_0(y))$ are surjective.

5.1.8 Exercise

Show that faithful functors are not necessarily embeddings: only when they are also injective on objects.

5.1.9 Proposition

Let $F:\mathbf{C}\to\mathbf{D}$ and $G:\mathbf{D}\to\mathbf{E}$ be functors.

1. If F and G are isomorphisms (resp. embeddings, faithful, full), so is $F;G$.
2. If $F;G$ is an embedding (resp. faithful), so is F.
3. If $F;G$ is full, so is G.

5.1.10 Example

Every subcategory $\mathbf{D}$ of a category $\mathbf{C}$ defines an inclusion functor $I_{D,C}:\mathbf{D}\to\mathbf{C}$. This functor is an embedding, and is full iff $\mathbf{D}$ is a full subcategory of $\mathbf{C}$.

Embeddings are, intuitively, the best approximations to subcategories. In fact, in a world in which concepts are normally taken up to isomorphism, it seems intuitive not to make any difference between embeddings and inclusions of subcategories because all that is at stake are the "identities" of the objects involved, not their properties or structure. This intuition is supported by the following result:

5.1.11 Proposition

A functor $F:\mathbf{D}\to\mathbf{C}$ is a (full) embedding iff there is a (full) subcategory $\mathbf{C}'$ of $\mathbf{C}$ and an isomorphism $G:\mathbf{D}\to\mathbf{C}'$ such that $F=G;I_{C',C}$.

The relationships that we studied, in section 3.3, between subcategories and isomorphisms extend to functors:

5.1.12 Proposition

Let $F:\mathbf{C}\to\mathbf{D}$ be a functor.

1. *F preserves isomorphisms*, i.e. if $f:x\to y$ is a $\mathbf{C}$-isomorphism, then $F(f)$ is also an isomorphism.
2. if F is faithful and full, then it *reflects isomorphisms*, i.e. if $F(f)$ for $f:x\to y$ is an isomorphism, then f is itself an isomorphism.

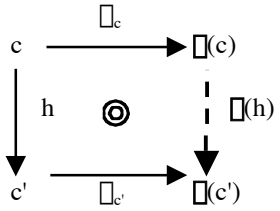
The notions of preservation and reflection will be extended in 5.2.1 to universal constructions.

We end this section with the definition of a class of functors that arise from (co)reflective subcategories.

5.1.13 Definition/proposition – reflector

Let $\mathbf{D}$ be a reflective subcategory of a category $\mathbf{C}$. We define a functor $F:\mathbf{C}\to\mathbf{D}$ as follows:

- every $\mathbf{C}$-object c has a $\mathbf{D}$-reflection arrow $\eta_c:c\to d$. We define $F(c)=d$;
- consider now a morphism $h:c\to c'$. The composition $h;\eta_{c'}$ is such that the definition of $\mathbf{D}$-reflection arrow for c guarantees the existence and uniqueness of a morphism $h':F(c)\to F(c')$ such that $h;\eta_{c'}=\eta_c;h'$. We define $F(h)=h'$.

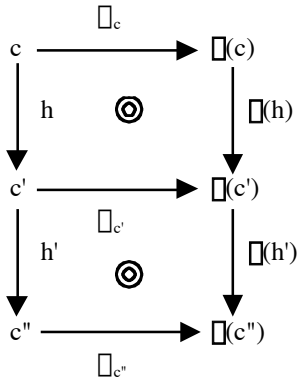


This functor is called a *reflector for* $\mathbf{C}$, more precisely *for the inclusion functor* $I_{\mathbf{D},\mathbf{C}}:\mathbf{D}\to\mathbf{C}$.

proof

We have to prove that a functor is indeed defined.

1. The fact that a graph homomorphism is defined (i.e. sources and targets of arrows are respected) is trivially checked.
2. By definition, $F(id_c)$ is the unique $\mathbf{D}$-morphism $h':F(c)\to F(c)$ satisfying $id_c;\eta_c=\eta_c;h'$. Because $id_{F(c)}$ also satisfies that equation, we get $id_{F(c)}=F(id_c)$.
3. Consider now the composition law.



The composition $\Phi(h);\Phi(h')$ is such that $(h;h');\eta_{c''} = \eta_c;(\Phi(h);\Phi(h'))$. This equality can be obtained from the equations satisfied individually by $\Phi(h)$ and $\Phi(h')$. Hence, by definition, $\Phi(h);\Phi(h')$ is the morphism $\Phi(h;h')$.

By duality, we obtain the notion of *co-reflector*. Notice that we had already made use of the idea of co-reflector in section 3.3 to explain the intuitions behind co-reflective subcategories. When applied to the categories defined in 3.5.4 for temporal specifications, we obtain several functors that perform the closure of sets of axioms, both as reflectors (from ***PRES*** and from ***SPRES*** to ***THEO***) and co-reflectors (from ***PRES*** both to ***SPRES*** and ***THEO***). Notice in particular that the functor from ***PRES*** to ***THEO*** is both a reflector and a co-reflector.

This proposition also shows that the relationship between automata and reachable automata is functorial. The map that eliminates non-reachable states as defined after 3.3.7 is a co-reflector.

We will generalise these kinds of functors in section 7.2. They are particular cases of classes of functors that generate free or canonical structures, or approximate other kinds of structures.

5.2 Universal constructions vs functors

In chapter 4, we motivated the study of universal constructions like (co)limits in terms of the ability to capture the collective behaviour of systems of interconnected components. In the previous section, we have just seen how functors provide us the means to map and relate different levels of abstraction in system development or different aspects of system description. One of the obvious questions that arises concerns the ability of functors to relate such universal constructions when applied in the categories related by the functors.

For an example of what we mean, consider what we consider to be one of the most profound "recent" contributions that have been made

in the area of Programming Languages and Models: the separation between "Computation" and "Coordination" [50]. The best introduction we know to this topic can be found in [3]. In a nutshell, this whole area of research evolves around the ability to separate what in systems are the structures responsible for the computations that are performed and the mechanisms that are made available for coordinating the interactions that may be required between them. A decade of research has shown that languages that support this separation of concerns can improve our ability to handle complex systems. Its importance for Software Architectures has led to very close synergies between the two areas and there is a lot of cross-fertilisation going on.

One of the challenges of this area has been to provide a mathematical characterisation of this separation that is independent of the particular languages that have been developed for this paradigm. Such a characterisation is important, on the one hand, to establish a systematic study of its properties and relationships to other paradigms and, on the other hand, extend and support the new paradigm with tools. This is what we started to do in [35] through the use of Category Theory.

The basic idea of our approach is to model this separation by a forgetful functor $\boldsymbol{int}: \boldsymbol{SYS} \to \boldsymbol{INT}$ where the category ***SYS*** stands for "systems", i.e. for whatever representations (models, behaviours, specifications, etc) we are using for addressing systems as a whole, and the category ***INT*** is intended to capture the mechanisms that, in these representations, are responsible for the coordination aspects. We shall refer to the objects of ***INT*** as "interfaces" following the idea that interconnections should be established only on the basis of what systems make available for interaction with other systems (e.g. communication channels), not at the level of the computations that they perform.

An example that will help us clarify more precisely what we have in mind can be given in terms of the linear temporal specifications that we have introduced in section 3.5. Therein, we motivated the fact that we were modelling the behaviour of concurrent processes at the level of the actions that are provided by their public interfaces in the sense that every signature identifies a set of actions in which a process can engage itself. The axioms of a specification provide an abstraction of the computations (traces) that are performed locally through the properties that they satisfy. The idea is, then, to take signatures as interfaces and characterise the ability of linear temporal logic specifications to separate computation and coordination in terms of properties of the functor $\boldsymbol{PRES}_{LTL} \to \boldsymbol{SET}$ that maps presentations and their morphisms to the underlying signatures and signature morphisms.

Which properties should we require of ***int*** to capture the proposed separation? Basically, we have to capture the fact that any interconnection of systems is established via their interfaces.

For instance, an important property is that ***int*** should be faithful. This means that morphisms of programs should not induce more relationships between programs than those that can be captured through their underlying interfaces. That is to say, by taking into consideration the computational part, we should not get additional observational power over the external behaviour of systems.

Another important property concerns the way colimits are computed. We have already mentioned that the colimit of a diagram expressing how a system is configured in terms of simpler components and interconnections between them, returns a model of the global behaviour of the system and the morphisms that relate the components to the global system. If only the interfaces matter for establishing the required interconnections, then the colimit of the diagram of systems should be obtainable from the colimit of the underlying diagram of interfaces. In other words, if we interconnect system components through a diagram, then any colimit of the underlying diagram of *interfaces* should be able to be lifted to a colimit of the original diagram of system components.

This property can be stated more precisely as follows: given any diagram $\boldsymbol{dia}:\boldsymbol{I}\to\boldsymbol{SYS}$ and colimit $(\boldsymbol{int}(S_i)\to C)_{i:\boldsymbol{I}}$ of $(\boldsymbol{dia};\boldsymbol{int})$ there exists a colimit $(S_i\to S)_{i:\boldsymbol{I}}$ of $\boldsymbol{dia}$ such that $\boldsymbol{int}(S_i\to S)=(\boldsymbol{int}(S_i)\to C)$. When a functor satisfies a property like this one we say that it *lifts colimits*.

This means that when we interconnect system components, any colimit of the underlying diagram of interfaces establishes an interface for which a computational part exists that captures the joint behaviour of the interconnected components. Notice that this property does not tell us how to construct the lift, it just ensures that it exists. We shall return to this point later on. This property is really about (non)-interference between computation and coordination: on the one hand, the computations assigned to the components cannot interfere with the viability (in the sense of the existence of a colimit) of the underlying configuration of interfaces; on the other hand, the computations assigned to the components cannot interfere in the calculation of the interface of the resulting system.

A kind of "inverse property" is also quite intuitive: that every interconnection of system components be an interconnection of the underlying interfaces. In particular, that computations do not make viable a configuration of system components whose underlying configuration of interfaces is not. This property is verified when ***int*** *preserves colimits*: given any diagram $\boldsymbol{dia}:\boldsymbol{I}\to\boldsymbol{SYS}$ and colimit $(S_i\to S)_{i:\boldsymbol{I}}$ of $\boldsymbol{dia}$, $(\boldsymbol{int}(S_i)\to\boldsymbol{int}(S))_{i:\boldsymbol{I}}$ is a colimit of $(\boldsymbol{dia};\boldsymbol{int})$. These two properties together imply that any colimit in ***SYS*** can be computed by first translating the diagram to ***INT***, then computing the colimit in ***INT***, and finally lifting the result back to ***SYS***.

So: does the (forgetful) functor $\mathbf{sign}:\mathbf{PRES}_{LTL} \to \mathbf{SET}$ satisfy these properties? The answer is "yes", but we will defer the proof and the recipe for constructing colimits to the next section. This is because this functor is an instance of a class that captures many useful structures in Computing and, hence, worth studying on its own, which includes the properties that we have just discussed. We shall return to the issue of separating "Computation" and "Coordination" in several other places in the book, including examples.

Summarising, it is useful to classify functors vis-à-vis the way they relate universal constructions in the source and target categories:

5.2.1 Definition

A functor $F:\mathbf{C} \to \mathbf{D}$

1. *preserves*
   - a colimit $p:D \to c$ of a diagram $D:I \to \mathbf{C}$ iff the co-cone $\{F(p_a): F(D_a) \to F(c)\}_{a \in I_0}$, denoted by $F(p):F;D \to F(c)$, is a colimit of $F;D$.
   - colimits of shape $\mathbf{I}$ iff it preserves the colimits of all diagrams $D:I \to \mathbf{C}$.
   - colimits iff it preverses the colimits of any diagram in $\mathbf{C}$.
2. *lifts* colimits iff for any diagram $D:I \to \mathbf{C}$ and colimit $p': F;D \to d$ of $F;D$, there is a co-cone $p:D \to c$ that is a colimit of D and $p'=F(p)$. The lift is *unique* (or F is said to lift colimits uniquely) when there is a unique co-cone $p:D \to c$ satisfying the two properties.
3. *reflects* colimits iff for any diagram $D:I \to \mathbf{C}$ and co-cone $p:D \to c$, if $F(p)$ is a colimit *of* $F;D$, then p is a colimit of D.
4. *creates* colimits iff for any diagram $D:I \to \mathbf{C}$ and colimit $p': F;D \to d$ of $F;D$, there is a unique co-cone $p:D \to c$ such that $p'=F(p)$ and, moreover, p is a colimit of D.

Definitions 2, 3 and 4 extend to specific classes of colimits (sums, co-equalisers, etc) as formulated for preservation. The dual notions have the obvious designations.

The following exercise can help in understanding the difference between these notions:

5.2.2 Exercise

Prove that:

1. Every functor that creates colimits also reflects them.
2. A functor creates colimits iff it reflects and lifts colimits uniquely.

Note that the requirement on reflecting colimits is essential in case 2. For instance, it is easy to prove that the (forgetful) functor $\mathbf{THEO}_{LTL} \to \mathbf{SET}$ that maps theories and their morphisms to the under-

lying signatures and signature morphisms lifts colimits uniquely but it is easy to see that it does not create them: the colimit will choose the minimal theory among the whole class of theories that have the given signature and give rise to a co-cone; this class is only singular when some inconsistency arises from the interconnections and/or the base theories.

We can also relate these properties to 4.4.7:

5.2.3 Exercise

Prove that if $\mathbf{C}$ is finitely co-complete then $\Box:\mathbf{C}\Box\ \mathbf{D}$ preserves finite colimits iff it preserves initial objects and pushouts.

More results and related notions can be found, for instance, in [1,22].

PART TWO – advanced topics

6 Functor-based constructions

Functors provide us not only with the ability to investigate relationships between categories, as shown in the previous chapter and continued in the first section of this chapter, but also to build new categories based on such relationships. In this chapter, we present some of the functor-based constructions that we have found useful in our day-to-day.

6.1 Functor-distinguished kinds of categories

In this section, we show how the properties of certain categories can be derived from the properties of functors that relate them to other categories. That is to say, we investigate functors as a means of revealing structural properties of categories and of their objects.

We start with the notion of concrete category, which appears already in [79, page 26] and is extensively explored in [1]. It is also a notion that we have found to be extremely useful in Computing, namely because we typically build new categories over old ones by adding some structure but without interfering with the structure of the original category. This construction by "conservative extension" can be captured by a faithful functor $u: D \to C$ where C is the old category and D is the new one. Typically, the functor "forgets" the structure that is being added to the objects of C to produce objects of D.

6.1.1 Definition – concrete categories

A *concrete* category over a category C is a pair $\langle D, U \rangle$ where $U: D \to C$ is a faithful functor.

Notice that we are using the terminology introduced in [1]. The notion of concrete category introduced in [79] is a particularisation of the one above to the case where the category C is ***SET***. Concrete categories over ***SET*** are called *constructs* in [1]. The category C is sometimes called the *base* category of $\langle D, U \rangle$ and U is called the *forgetful* or *underlying* functor .

Because the underlying functor is faithful, i.e. injective on hom-sets, for each pair of D-objects $\langle x,y \rangle$, $hom_D(x,y)$ is usually regarded as a subset of $hom_C(U(x),U(y))$. Following [1], we shall often use the expres-

sion "*f:*Φ*(x)*$\to$$\Phi$*(y) is a* ***D****-morphism*" to mean that there exists a (necessarily) unique ***D***-morphism $x \to y$ whose image by Φ is *f*.

We have already come across a few concrete categories:

6.1.2 Examples

1. The category ***CLOS*** of closure systems defined in section 3.6 is concrete over ***SET***: the forgetful functor maps closure systems to the underlying sets (languages).
2. The categories $\boldsymbol{PRES}_{LTL}$, $\boldsymbol{SPRES}_{LTL}$ and $\boldsymbol{THEO}_{LTL}$ defined in 3.5.4 are all concrete over the category ***SET*** of sets. The underlying functors, which we will name $\boldsymbol{sign}_{LTL}$, "forget" the sets of axioms/theorems, mapping theories and their presentations to the corresponding signatures.
3. Another example is the category ***AUTOM*** of automata as defined in 2.2.12. This category is concrete over the product $\boldsymbol{SET} \times \boldsymbol{SET} \times \boldsymbol{SET}$. The underlying functor forgets the input, output and transition functions and projects every automaton to its sets of inputs, states and outputs.

A typical situation in which concrete categories arise is one in which the underlying functor represents some sort of classification or typing mechanism that is strong enough to extend to the morphisms and, hence, to the structure defined over the objects by the morphisms. In such circumstances, one is usually interested in studying the way all the objects that share the same classification or type relate to one another.

6.1.3 Definition – fibres

Given a concrete category $\langle \boldsymbol{D}, \Phi \rangle$ over ***C*** and a ***C***-object *c*, the *fibre* of *c* is the pre-order that consists of all the objects *d* of ***D*** that are mapped to *c*, i.e. such that $\Phi(d)=c$, ordered by $d_1 \leq_c d_2$ iff $id_c:\Phi(d_1) \to \Phi(d_2)$ is a ***D***-morphism.

The structure of the fibres reveals a lot about the properties of the concrete category. A detailed discussion can be found in [1]. We shall limit our study to three cases that will be used further on in the book.

6.1.4 Definition

A concrete category $\langle \boldsymbol{D}, \Phi \rangle$ over ***C*** is called:

1. *amnestic* provided that its fibres are partially ordered, i.e. $d_1 \leq_c d_2$ and $d_2 \leq_c d_1$ implies $d_1=d_2$ for all ***C***-objects *c* and objects d_1, d_2 in the fibre of *c*.
2. *fibre-complete* if its fibres are complete lattices (i.e. admit arbitrary meets and joints).
3. *fibre-discrete* if its fibres are ordered by equality.

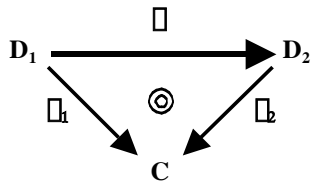
Amnestic concrete categories are such that the morphisms of $\boldsymbol{D}$ do not introduce additional properties over the structures that are being superposed on $\boldsymbol{C}$. For instance, $\boldsymbol{SPRES}_{LTL}$ and $\boldsymbol{THEO}_{LTL}$ are amnestic because their morphisms do not introduce any structural properties over the sets of axioms/theorems. However, $\boldsymbol{PRES}_{LTL}$ is not amnestic because its morphisms capture the notion of closure. Two presentations over the same temporal signature can be isomorphic without being equal. On the other hand, strictly isomorphic presentations and isomorphic theories over the same temporal signature are necessarily equal: the former because the consequences of the axioms are not taken into consideration, and the latter because they are already closed under consequence. In other words, morphisms of theories just take into account the elements of a set (the theorems) whereas morphisms of presentations have to compute the closure of a set.

Concrete categories that are fibre-complete superpose over the objects of $\boldsymbol{C}$ information that the morphisms organise in a "convenient" way, allowing for operations to be performed internally within each fibre. We shall see some examples further on. Hence, being fibre-complete is a step further than amnesticity in the degree of interaction that exists between the existing and the superposed structure. Fibre-discrete categories present still a step further: they are such that the extension that $\boldsymbol{D}$ makes over the objects of $\boldsymbol{C}$ is inessential, i.e. it has no intrinsic structure or meaning – it acts just like a comment.

Because concrete categories have "added structure", we should provide a notion of functor that reflects that structure:

6.1.5 Definition – concrete functors

A *concrete functor* Φ between two concrete categories $\langle \boldsymbol{D}_1, \Box_1 \rangle$ and $\langle \boldsymbol{D}_2, \Box_2 \rangle$ over the same underlying category $\boldsymbol{C}$ is a functor $\Phi : \boldsymbol{D}_1 \to \boldsymbol{D}_2$ such that $\Box_1 = \Phi ; \Box_2$.



Because the morphisms of a concrete category are "imported" from its underlying category, concrete functors cannot act on them and, hence, are fully determined by their values on objects:

6.1.6 Proposition

Given concrete functors Φ and Ψ between two concrete categories $\langle \boldsymbol{D}_1, \Box_1 \rangle$ and $\langle \boldsymbol{D}_2, \Box_2 \rangle$, $\Phi = \Psi$ if, for every $\boldsymbol{D}_1$-object d, $\Phi(d) = \Psi(d)$.

proof

Let $f:d \to d'$ be a morphism of $\boldsymbol{D}_1$. Given that both functors agree on objects, both $\Phi(f)$ and $\Psi(f)$ have the same source and target. Because both Φ and Ψ are concrete, we have $\Phi_1(f)=\Phi_2(\Phi(f))=\Phi_2(\Psi(f))$. Finally, because Φ_2 is faithful, we can conclude that $\Phi(f)=\Psi(f)$. Hence, both functors agree on morphisms as well.

6.1.7 Remark – concrete subcategories

Given that subcategories determine embeddings (see 5.1.10), and that the composition of faithful functors is also faithful (see 5.1.9), every subcategory $\boldsymbol{D}'$ of a category $\boldsymbol{D}$ that is concrete over $\boldsymbol{C}$ with underlying functor Φ can be regarded also as a concrete category over $\boldsymbol{C}$ whose underlying functor is the composition $\Phi_{\boldsymbol{D}',\boldsymbol{D}};\Phi$. We then say that $\langle \boldsymbol{D}', \Phi_{\boldsymbol{D}',\boldsymbol{D}};\Phi \rangle$ is a concrete subcategory of $\langle \boldsymbol{D}, \Phi \rangle$.

Consider now the notions of reflective and co-reflective subcategories that we discussed in section 3.3. Intuitively, for the (co)reflection to be "concrete", i.e. to be consistent with the classification that the underlying functor provides, we would like to remain within the same fibre, i.e. we would like that the (co)reflection arrows be identities:

6.1.8 Definition – concretely (co)reflective subcategories

A concrete subcategory $\langle \boldsymbol{D}_1, \Phi_1 \rangle$ of $\langle \boldsymbol{D}_2, \Phi_2 \rangle$ is *concretely (co)reflective* iff, for each object of $\boldsymbol{D}_2$, there is a (co)reflection arrow that is mapped by Φ_2 to the identity.

For instance, although ***REACH*** is a co-reflective subcategory of ***AUTOM***, it is not concretely co-reflective when both categories are considered as being concrete over $\boldsymbol{SET} \to \boldsymbol{SET} \to \boldsymbol{SET}$. This is because the reachable automata are not necessarily in the same fibre as the automata from which they are computed – their state space may have been reduced.

"Concreteness" also extends to universal constructions in the sense that, once we "look" at a category $\boldsymbol{D}$ as being concrete through the underlying functor $\Phi:\boldsymbol{D} \to \boldsymbol{C}$, we can classify the way universal constructions in $\boldsymbol{D}$ relate to $\boldsymbol{C}$ through Φ:

6.1.9 Definition – concrete universal constructions

Consider a concrete category $\langle \boldsymbol{D}, \Phi \rangle$ over $\boldsymbol{C}$ and a diagram $\Delta:\boldsymbol{I} \to \boldsymbol{D}$. A (co)limit of Δ is said to be a *concrete (co)limit* of Δ in $\langle \boldsymbol{D}, \Phi \rangle$ iff it is preserved by Φ.

That is to say, universal constructions are concrete when they map to the underlying category. In some categories, all universal constructions are concrete. We shall see an example in 6.3.6. In other concrete categories, some universal constructions may be concrete and others not so.

6.1.10 Exercise

Workout examples of limits and colimits in ***AUTOM*** that are not concrete over ***SET***$\times$***SET***$\times$***SET***.

The construction of concrete (co)limits, when they exist, can be systematised through a process that consists in projecting the (co)cones to the base category where a (co)limit is computed and then lifted back. This process can be extended to more general notions of fibre as discussed below.

The notion of fibre is not exclusive to concrete categories. It can be generalised to arbitrary functors as follows:

6.1.11 Definition – fibres

Consider a functor $\Box$:***D***$\Box$ ***C***.

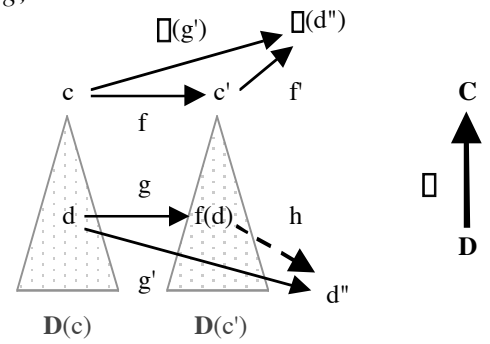
1. Given a ***C***-object c, the *fibre of* c, which we denote by ***D***(c), is the subcategory of ***D*** that consists of all the objects d that are mapped to c, i.e. such that $\Box(d)=c$, together with the ***D***-morphisms $f:d_1\Box d_2$ such that $\Box(f)=id_c$.
2. The functor $\Box$ is said to be *amnestic* if, in its fibres, no two distinct objects are isomorphic. That is, if an isomorphism $f:d_1\Box d_2$ in ***D*** is such that $\Box(f)=id_c$ for some object c of ***C***, then f is itself an identity.

The discussion around concrete categories focused basically on the structure of fibres. Another interesting aspect worth discussing is the relationships that morphisms on the underlying category induce over the fibres corresponding to their sources and targets.

For instance, consider again the category $\boldsymbol{THEO}_{LTL}$ of temporal theories defined in 3.5.4 and the forgetful functor $\boldsymbol{sign}_{LTL}$ that maps theories and their morphisms to their signature components. Consider an arbitrary signature $\Box$. A signature morphism $f:\Box\Box\Box'$ provides a translation from the symbols of one signature to symbols of the other. We have seen in 3.5.7 that such a translation extends to the temporal language associated with $\Box$. Can we further extend this translation to the fibres of $\Box$, i.e. can we define a mechanism that translates theories with signature $\Box$ to theories over $\Box$"? What about the reverse? Can we define a mechanism that translates theories with signature $\Box$' to theories over $\Box$?

To answer these questions, we have first to provide a reasonable definition of "translation" (and inverse translation) between fibres induced by a morphism. Consider a functor $\Box$:***D***$\Box$ ***C***, a ***C***-morphism $f:c\Box c'$, and a ***D***-object d in the fibre of c, i.e. $\Box(d)=c$. By the "image" of d under f we mean some object $f(d)$ in the fibre of c' that is "closest" to d. By "closest" we mean the following: f can be lifted to a morphism between d and $f(d)$, i.e. there must exist some morphism $g:d\Box f(d)$ such that $\Box(g)=f$; for any other object d" and morphism $g':d\Box d$" that leaves d" at a "distance" $f':c'\Box\Box(d")$ of f, i.e. such that $\Box(g')=f;f'$, this distance can be

covered in ***D*** in a unique way by a morphism $h:f(d)\to d''$ such that $\Phi(h)=f'$ and $g'=g;h$.



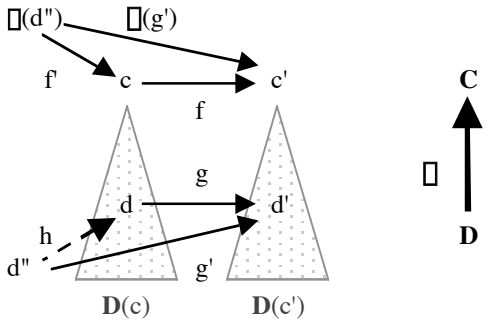
By the inverse translation, we mean the dual notion.

The following definitions are adapted from [12], a good source for the study of fibre-related matters.

6.1.12 Definition – (co)cartesian morphisms

Let $\Phi:\mathbf{D}\to\mathbf{C}$ be a functor and $f:c\to c'$ a ***C***-morphism.

1. Let $d:\mathbf{D}(c)$. A ***D***-morphism $g:d\to d'$ is said to be *cocartesian* for f and d iff (1) $\Phi(g)=f$ and (2) for every $g':d\to d''$ and $f':c'\to\Phi(d'')$ such that $\Phi(g')=f;f'$, there is a unique morphism $h:d'\to d''$ such that $\Phi(h)=f'$ and $g'=g;h$.
2. Let $d':\mathbf{D}(c')$, i.e. a ***D***-object such that $\Phi(d')=c'$. A ***D***-morphism $g:d\to d'$ is said to be *cartesian* for f and d' iff (1) $\Phi(g)=f$ and (2) for every $g':d''\to d'$ and $f':\Phi(d'')\to c$ such that $\Phi(g')=f';f$, there is a unique morphism $h:d''\to d$ such that $\Phi(h)=f'$ and $g'=h;g$.



6.1.13 Definition – (co)fibration

Let $\Phi:\mathbf{D}\to\mathbf{C}$ be a functor.

1. We say that Φ is a *fibration* if, for every ***C***-morphism $f:c\to c'$ and ***D***-object d' in the fibre of c', there is a cartesian morphism for f and d'.

2. We say that $\Box$ is a *cofibration* provided that, for every $\boldsymbol{C}$-morphism $f:c\Box c'$ and $\boldsymbol{D}$-object d in the fibre of c, there is a cocartesian morphism for f and d.

6.1.14 Example – specifications as (co)fibrations

The forgetful functors that define $\boldsymbol{PRES}_{LTL}$, $\boldsymbol{SPRES}_{LTL}$ and $\boldsymbol{THEO}_{LTL}$ as concrete categories over the category $\boldsymbol{SET}$ are all fibrations and cofibrations at the same time, but with different (co)cartesian morphisms among themselves. Given a signature morphism $f:\Box\Box\Box'$:

1. a cartesian morphism for a theory $<\Box,\Box'>$ is $f:<\Box,f^{-1}(\Box')>\Box<\Box,\Box'>$ and a cocartesian morphism for a theory $<\Box,\Box>$ is $f:<\Box,\Box>\Box<\Box',c(f(\Box))>$.
2. a cartesian morphism for a presentation $<\Box',\Box'>$ is $f:<\Box,f^{-1}(c(\Box'))>\Box<\Box',\Box'>$ and a cocartesian morphism for a presentation $<\Box,\Box>$ is $f:<\Box,\Box>\Box<\Box',f(\Box)>$.
3. a cartesian morphism for a strict presentation $<\Box',\Box'>$ is $f:<\Box,f^{-1}(\Box')>\Box<\Box',\Box'>$ and a cocartesian morphism for a strict presentation $<\Box,\Box>$ is $f:<\Box,\Box>\Box<\Box',f(\Box)>$.

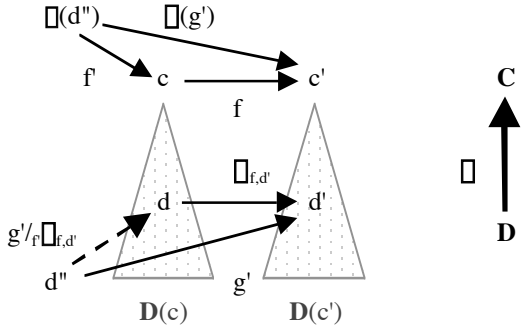
The differences that we can witness between these three cases have to do with both the properties of the closure operator and the morphisms that characterise the categories. For instance, even if $\Box$ is closed for consequence, $f(\Box)$ is not necessarily so. Hence, the cocartesian morphisms for theories have to return the closure of the image set. This is not necessary for presentations because it is already implicit in the morphisms, nor for strict presentations because they do not involve the consequence operator at all. The cartesian morphisms, on the contrary, do not need to compute the closure of the inverse image set. This is because, if $\Box$ is closed, so is $f^{-1}(\Box)$. However, presentations do need to compute the closure explicitly because they only do it implicitly to the target, not the source.

These examples illustrate the fact that there may be more than one cocartesian morphism for a given signature morphism and theory presentation. This is because the set of axioms of the specification can be determined only up to logical equivalence. In the case of theories, the fact that the set of sentences must be closed ensures that the lifts are unique. Hence, although it is possible to generalise the translation induced by signature morphisms to presentations, there may be more than one way of doing so. This is the general case of any (co)fibration. Amnestic concrete (co)fibrations, however, guarantee uniqueness of the lifting and, hence, of the choice for (co)cartesian morphisms.

6.1.15 Definition – cleavage, cloven fibration

Let $\Box:\mathbf{D}\to\mathbf{C}$ be a functor. A choice of a cartesian morphism for every $\mathbf{C}$-morphism $f:c\to\Box(d')$ and $\mathbf{D}$-object d' is called a *cleavage*. A fibration equipped with a cleavage is called *cloven*.

We shall often denote by $\Box_{f,d'}$ the cartesian morphism selected for f and d' by the cleavage. Given $g':d''\to d'$ and $f':\Box(d'')\to c$ such that $\Box(g')=f';f$, we know that there is a unique morphism $h:d''\to d$ such that $\Box(h)=f'$ and $g'=h;\Box_{f,d'}$. We shall denote h by $g'/_f\Box_{f,d'}$. When f' is id_c, we omit the subscript.



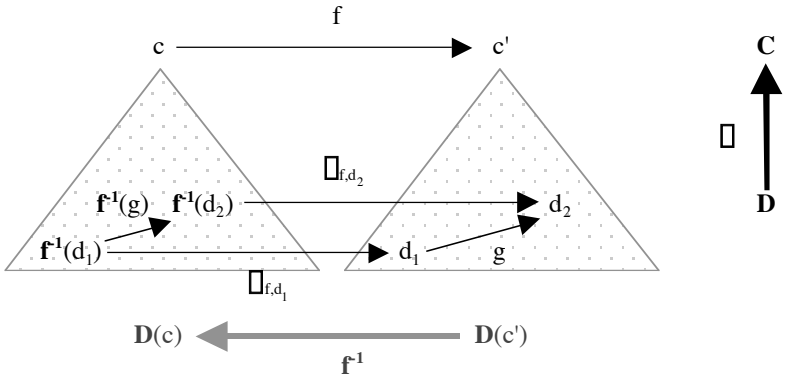
The dual notion is called cocleavage, and a cofibration equipped with a cocleavage is also said to be cloven.

Summarising, a cloven (co)fibration $\Box:\mathbf{D}\to\mathbf{C}$ provides us with a way of lifting morphisms to translations between the objects of their fibres in the sense that every $\mathbf{C}$-morphism $f:c\to c'$ defines a map from the objects of $\mathbf{D}(c')$ to the objects of $\mathbf{D}(c)$ in the case of a fibration, and from the objects of $\mathbf{D}(c)$ to the objects of $\mathbf{D}(c')$ in the case of a cofibration. Can these translations be generalised to the morphisms of the fibres? That is, can we generalise this mapping to a functor between the fibres?

6.1.16 Proposition

Let $\Box:\mathbf{D}\to\mathbf{C}$ be a functor and $f:c\to c'$ a $\mathbf{C}$-morphism.

1. If $\Box$ is a cloven fibration, then f defines a functor $\mathbf{f}^{-1}:\mathbf{D}(c')\to\mathbf{D}(c)$ as follows
   - given $d':\mathbf{D}(c')$, $\mathbf{f}^{-1}(d')$ is the source of the cartesian morphism $\Box_{f,d'}:d\to d'$ that the cleavage associates with the fibration;
   - given $g:d_1\to d_2$ in $\mathbf{D}(c')$, $\mathbf{f}^{-1}(g)$ is the morphism $\mathbf{f}^{-1}(d_1)\to\mathbf{f}^{-1}(d_2)$ that results from the universal property of the cartesian morphism $\Box_{f,d_2}:\mathbf{f}^{-1}(d_2)\to d_2$ when applied to $\Box_{f,d_1};g$ and id_c. That is to say, $\mathbf{f}^{-1}(g)$ is the morphism that we also denote by $(\Box_{f,d_1};g)/\Box_{f,d_2}$. Notice that this morphism in the only one that satisfies $\Box_{f,d_1};g=\mathbf{f}^{-1}(g);\Box_{f,d_2}$ and $\Box(\mathbf{f}^{-1}(g))=id_c$.



2. If Φ is a cloven cofibration, then f defines a functor $f:\mathbf{D}(c)\to\mathbf{D}(c')$ in the dual way, i.e. by working on the target side of the cocartesian morphism.

proof

1. There are three properties to prove. The reader is invited to fill-in the details:
   - the functor is well defined in the sense that the image objects and morphisms exist and are of the right types;
   - because $id_{f^{-1}(d)}$ satisfies the properties of the universal property that characterises $f^{-1}(id_d)$, the uniqueness associated with that universal property guarantees that the cleavage has no other choice;
   - the same line of reasoning applies to conclude that $f^{-1}(g_1;g_2)=f^{-1}(g_1);f^{-1}(g_2)$
2. By duality.

This definition raises an immediate question: what if $f=id_c$? Are f^{-1} and f the identity functor? What if $f=f_1;f_2$? Are $f^{-1}$and f the compositions $f^{-1}_1;f^{-1}_2$ and $f_1;f_2$, respectively? The answer is twofold: they can, but they do not have to. Consider the first part of the answer, this time instantiated for cofibrations.

6.1.17 Proposition

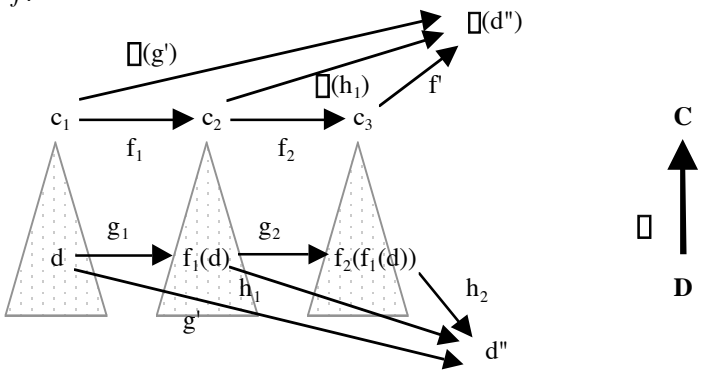
Let $\Phi:\mathbf{D}\to\mathbf{C}$ be a functor.

1. Given a $\mathbf{C}$-object c and an object d in the fibre of c, the identity id_d is both a cartesian and a cocartesian morphism for id_c and d.
2. Given $\mathbf{C}$-morphisms $f_1:c_1\to c_2$ and $f_2:c_2\to c_3$, an object d in the fibre of c_1, and "translations" (cocartesian morphisms) $g_1:d\to f_1(d)$ and $g_2:f_1(d)\to f_2(f_1(d))$, the composition $g_1;g_2$ provides a cocartesian morphism for $f_1;f_2$ and d.

proof

1. Exercise.
2. There are two properties to prove:

- The composite translation lifts the composition in **C**:
  $\Phi(g_1;g_2)=\Phi(g_1);\Phi(g_2)=f_1;f_2$
- The lift has the required couniversal property:
  Let $g':d\to d''$ and $f':c_3\to\Phi(d'')$ be such that $\Phi(g')=f_1;f_2;f'$. The couniversal property of g_1 applied to g' and $(f_2;f')$ gives a unique $h_1:f_1(d)\to d''$ such that $g_1;h_1=g'$ and $\Phi(h_1)=f_2;f'$. We can now apply the couniversal property of g_2 to h_1 and f' to infer the existence and uniqueness of $h_2:f_2(f_1(d)),)\to d''$ such that $g_2;h_2=h_1$ and $\Phi(h_2)=f'$. We are now going to prove that h_2 has the require properties. On the one hand, $\Phi(h_2)=f'$. On the other hand, if we take $h'_2:f_2(f_1(d)),)\to d''$ such that $g_1;g_2;h'_2=g'$ and $\Phi(h'_2)=f'$, we derive $g_2;h'_2=h_1$ because we have $g_1;(g_2;h'_2)=g'$ and $\Phi(g_2;h'_2)=\Phi(g_2);\Phi(h'_2)=f_2;f'$, and h_1 is the unique morphism satisfying these two properties; but, then, we conclude that $h'_2=h_2$ because h_1 is the unique morphism satisfying $g_2;h_2=h_1$ and $\Phi(h_2)=f'$.



The fact that cleavages do not need to choose (co)cartesian morphisms satisfying these properties leads to the following definition.

6.1.18 Definition – split fibration

Let $\Phi:D\to C$ be a cloven fibration. If, for every C-object c, id_c^{-1} is $id_{C(c)}$ and, for every decomposition $f=f_1;f_2$, f^{-1} is the composition $f^{-1}_2;f^{-1}_1$, then the fibration is said to be *split*.

6.1.19 Remark

The more alert reader may have noticed that the universal property of the (co)cartesian morphism is taken over a space that is larger than the fibre of either c or c'. This is why the "distance" f' is introduced as a kind of "type converter". This means that the translation or inverse translation is chosen not only over the objects that have the exact type but also those that "type check". The reason we point this out is to recall that the "categorical way" of defining a concept, namely of choosing a universal property, has to be "morphism-oriented" in order to yield "good" properties such as the one that we have just proved. The

reader is encouraged to experiment with the weaker notion of (co)cartesian morphism for which the universal property is required only over the fibres, and check which of the properties that we have proved still hold.

Split (co)fibrations allow us to wrap up the idea of translation between fibres with which we started in neat categorical clothing:

6.1.20 Proposition

Let $\mathcal{D}:\mathbf{D}\to\mathbf{C}$ be a functor.

1. If $\mathcal{D}$ is a split fibration, then it defines a functor $\mathbf{ind}(\mathcal{D}):\mathbf{C}^{op}\to\mathbf{CAT}$ by mapping every $\mathbf{C}$-object c to its fibre $\mathbf{D}(c)$ and every morphism $f:c\to c'$ to the functor $\mathbf{f}^{-1}:\mathbf{D}(c')\to\mathbf{D}(c)$ as above.
2. If $\mathcal{D}$ is a split cofibration, then it defines a functor $\mathbf{ind}(\mathcal{D}):\mathbf{C}\to\mathbf{CAT}$ by mapping every $\mathbf{C}$-object c to its fibre $\mathbf{D}(c)$ and every morphism $f:c\to c'$ to the functor $\mathbf{f}:\mathbf{D}(c)\to\mathbf{D}(c')$ as above.

proof

Left as an exercise.

Functors of the form $\mathbf{C}^{op}\to\mathbf{CAT}$ are called *indexed categories*, a structure widely applied in Computing (see, for instance, [104]) precisely because it generalises what are usually called indexed-sets, i.e. mechanisms for indexing given collections of objects with other kind of objects (indexes). We shall focus on indexed categories in section 6.4.

One of the reasons to study fibrations is the close relationship that exists between their universal properties and those of the fibres.

6.1.21 Definition – fibre completeness

A cloven fibration $\mathcal{D}:\mathbf{D}\to\mathbf{C}$ is said to be *fibre-complete* if its fibres are complete categories and the inverse translation functors induced on the fibres preserve limits.

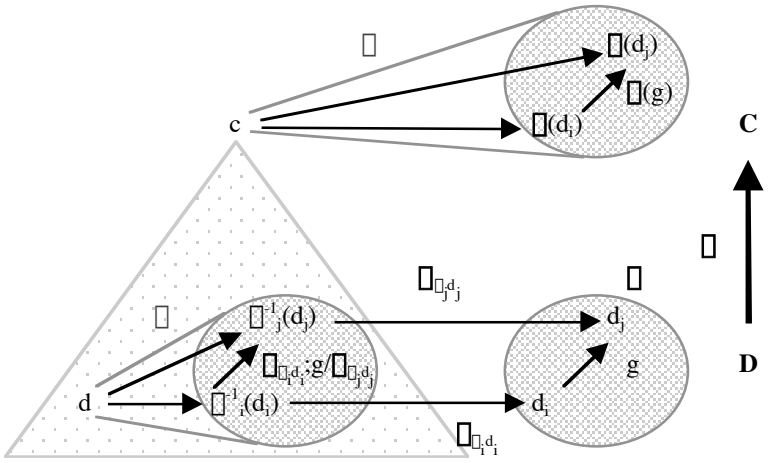
6.1.22 Proposition

Let $\mathcal{D}:\mathbf{D}\to\mathbf{C}$ be a split fibration.

1. If $\mathcal{D}$ is fibre-complete, then it lifts limits.
2. If, in addition, $\mathcal{D}$ is amnestic, the lift is unique.

proof

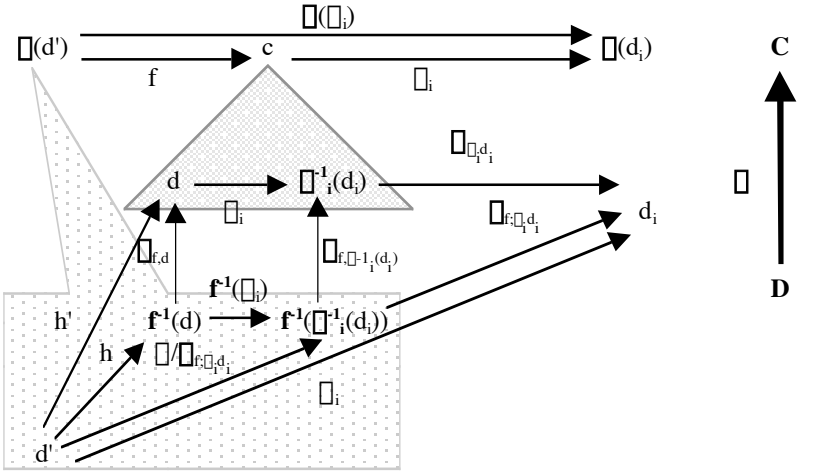
1. Consider a diagram $\mathcal{D}:\mathbf{I}\to\mathbf{D}$ and a limit $\lambda:c\to\mathcal{D}\mathcal{D}$ for the underlying $\mathbf{C}$-diagram. We are going to provide a step-by-step construction of a limit for $\mathcal{D}$ that lifts λ.



- first, we take the inverse image of $\mathcal{I}$ induced by $\mathcal{F}$. We obtain a diagram within the fibre of *c*, related to the original one by the cartesian morphisms $\varphi_{\mathcal{I}d_i}$. Notice that morphisms $g:d_i \to d_j$ are translated to $\varphi_{\mathcal{I}d_i};g/\varphi_{\mathcal{I}d_j}:\mathcal{F}^{-1}(d_i) \to \mathcal{F}^{-1}(d_j)$
- then, given that the fibre of *c* is complete, we compute the limit $\lambda:d \to \mathcal{F}^{-1}(\mathcal{I})$. We are going to prove that the cone obtained through the compositions $\lambda_i;\varphi_{\mathcal{I}d_i}$ is a limit for $\mathcal{I}$.
- the cone is commutative: consider $g:d_i \to d_j$ in $\mathcal{I}$; we have $\lambda_i;\varphi_{\mathcal{I}d_i};g=\lambda_i;\varphi_{\mathcal{I}d_i};g/\varphi_{\mathcal{I}d_j};\varphi_{\mathcal{I}d_j}=\lambda_j;\varphi_{\mathcal{I}d_j}$.
- the universal property of the cone is satisfied: consider another commutative cone $\mu:d' \to \mathcal{I}$, because the image of $\mathcal{F}$ under $\mathcal{I}$ is a commutative cone, the universal property of $\mathcal{I}:c \to \mathcal{I}(\mathcal{I})$ implies the existence of a unique $f:\mathcal{I}(d') \to c$ such that $f;\mathcal{I}_i=\mathcal{I}(\mu_i)$ for every $i \in I$; unfortunately, d' is not necessarily in the fibre of *c* and, hence, we cannot use the universal properties of the limit.
- Because the inverse translation induced by *f* preserves limits, the cone $f^{-1}(\mathcal{I}):f^{-1}(d) \to f^{-1}(\mathcal{I}^{-1}(\mathcal{I}))$ is itself a limit in the fibre of $\mathcal{I}(d')$; given that $\mu_i/\varphi_{f;\mathcal{I}d}:d' \to (f;\mathcal{I})^{-1}(\mathcal{I})$ is also a commutative cone and $(f;\mathcal{I})^{-1}(\mathcal{I})=f^{-1}(\mathcal{I}^{-1}(\mathcal{I}))$ is a consequence of $\mathcal{F}$ being split, we can infer the existence of a unique $h:d' \to f^{-1}(d)$ such that $h;f^{-1}(\mathcal{I})=\mu_i/\varphi_{f;\mathcal{I}d_i}$.

We now prove that $h;\lambda_{f,d}$ satisfies the required properties.

- $(h;\lambda_{f,d});(\lambda_i;\varphi_{\mathcal{I}d_i})$
  $=h;f^{-1}(\lambda_i);\varphi_{f;\mathcal{I}^{-1}(d_i)};\varphi_{\mathcal{I}d_i}$
  $=\mu_i/\varphi_{f;\mathcal{I}d_i};\varphi_{f;\mathcal{I}^{-1}(d_i)};\varphi_{\mathcal{I}d_i}$
  $=\mu_i/\varphi_{f;\mathcal{I}d_i};\varphi_{f;\mathcal{I}d}$
  $=\mu_i$.
- Let $h':d' \to d$ satisfy $h';(\lambda_i;\varphi_{\mathcal{I}d_i})=\mu_i$. We then have
  λ_I
  $=h'/\lambda_{f,d};\lambda_{f,d};\lambda_i;\varphi_{\mathcal{I}d_i}$
  $=h'/\lambda_{f,d};f^{-1}(\lambda_i);\varphi_{f;\mathcal{I}^{-1}(d_i)};\varphi_{\mathcal{I}d}$
  $=h'/\lambda_{f,d};f^{-1}(\lambda_i);\varphi_{f;\mathcal{I}d_i}$
  which implies $h'/\lambda_{f,d};f^{-1}(\lambda_i)=\mu_i/\varphi_{f;\mathcal{I}d_i}$ and, hence, $h'/\lambda_{f,d}=h$, and $h;\lambda_{f,d}=h'$.



2. Left as an exercise.

6.1.23 Corollary

Let $\Phi: \boldsymbol{D} \to \boldsymbol{C}$ be a split fibration. If Φ is fibre-complete and $\boldsymbol{C}$ is complete, then $\boldsymbol{D}$ is also complete.

These results and their duals tell us how to compute universal constructions over (co)fibrations that are fibre-(co)complete: the diagram is projected to the underlying category and its (co)limit is computed; the original diagram is (co)translated to the fibre of the apex and its (co)limit is computed within the fibre.

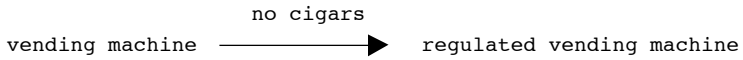
6.1.24 Example – colimits of specification diagrams

The fibres of $\boldsymbol{PRES}_{LTL}$, $\boldsymbol{SPRES}_{LTL}$ and $\boldsymbol{THEO}_{LTL}$ as concrete categories over $\boldsymbol{SET}$ are all complete and co-complete as ordered sets. It is also easy to see that the (co)translations are (co)continuous. Taking into account the operations that define these universal constructions, we have the following procedure for calculating limits and colimits of a diagram Δ with $\Delta = \langle I, \Delta_i \rangle$ in these categories.

1. Calculate the limit $\lambda: \Lambda \to \Delta$ or colimit $\lambda: \Delta \to \Lambda$ of the underlying diagram of signatures.
2. Lift the result by computing the specification-component according to the following rules:

	limit	**colimit**
$\boldsymbol{PRES}_{LTL}$	$\Lambda: \langle \Lambda, \bigcup_{i \in I} \lambda_i^{-1}(c(\Delta_i)) \rangle \to \Delta$	$\Lambda: \Delta \to \langle \Lambda, \bigcup_{i \in I} \lambda_i(\Delta_i) \rangle$
$\boldsymbol{SPRES}_{LTL}$	$\Lambda: \langle \Lambda, \bigcup_{i \in I} \lambda_i^{-1}(\Delta_i) \rangle \to \Delta$	$\Lambda: \Delta \to \langle \Lambda, \bigcup_{i \in I} \lambda_i(\Delta_i) \rangle$
$\boldsymbol{THEO}_{LTL}$	$\Lambda: \langle \Lambda, \bigcup_{i \in I} \lambda_i^{-1}(\Delta_i) \rangle \to \Delta$	$\Lambda: \Delta \to \langle \Lambda, c(\bigcup_{i \in I} \lambda_i(\Delta_i)) \rangle$

In order to illustrate these constructions, consider the specification of the vending machine that we studied in section 3.5. Therein, we saw how the original specification (3.5.6) could be extended to include a mechanism for regulating the sale of cigars, which was captured by a morphism (3.5.11)



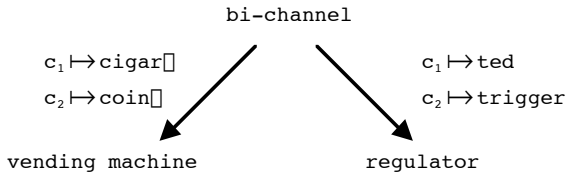
The need for such kind of extensions arises whenever there is a change in the original requirements. Such changes may be very frequent in business domains that are very volatile, for instance as a result of fierce competition forcing companies to maintain a level of service that matches or beats the offer of their rivals. This prompts the need for mechanisms that allow systems to evolve in ways that localise the impact of changes. For instance, a better way of accounting for the need to regulate the sale of cigars is to interconnect the original vending machine with an external device (regulator). This would indicate that the required change does not need to be intrusive of the existing system in the sense that it is not necessary to change the way the existing system is implemented. Instead, we just need to implement the regulator and connect it to the system, even while it is running, i.e. without interruption of service.

The required regulator can be specified as follows:

```
specification  regulator is
signature      trigger, ted, tor
axioms         beg    (¬tor)
               trigger  (¬ted)Wtor
               tor   (¬ted)
```

The rationale is the following. When the trigger occurs, action *ted* (the one being regula*ted*) is blocked until the regula*tor* occurs.

As discussed in chapter 4, we use diagrams to express systems as configurations of interconnected components, the (co)limits of which return the object that represents the system as a component, all the interconnections having been encapsulated. The regulated vending machine can be put together by synchronising action *trigger* of the regulator with action *coin* of the vending machine, and the regula*ted* action with *cigar*. The corresponding configuration diagram is:

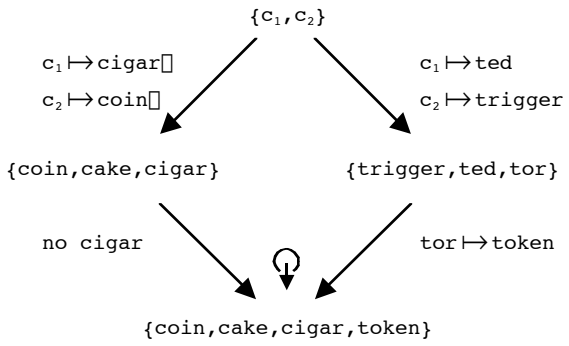


where

specification bi-channel is
signature c_1, c_2
axioms

Notice that there is nothing in the regulator that names the vending machine: the regulator is intended to be a component that can be re-used in different contexts. The same applies, a fortiori, to the vending machine because it was developed before the need to be regulated was determined. As a result, there is no implicit interaction between the two components. The interconnection through which the vending machine becomes regulated is completely externalised in the bi-channel and the two morphisms. This is one basic difference between object and service-oriented development as already mentioned: whereas objects, through clientship, code up interactions in the way features are called, services cannot name other services because they are not specified with particular interactions in mind. Moreover, service integration is only performed when it is needed, in run-time, and for the services that will have been identified or selected *at that time* from the configuration.

According to the rules above, the pushout of this configuration diagram can be calculated in two steps. First, we compute the pushout of the underlying diagram of signatures. Because pushouts are determined only up to isomorphism, we choose the signature that matches that of the regulated vending machine.



The second step consists in lifting the result back to the category of specifications. According to the rules given above, this can be achieved by choosing as axioms the translations of the axioms of the component specifications. From the vending machine we get:

beg (¬cake ¬cigar) ⇒ (coin (¬cake ¬cigar)**W**coin
coin (¬coin)**W**(cake cigar)
(cake cigar) (¬cake ¬cigar)Wcoin
cake (¬cigar)

And from the regulator we get:

```
beg    (¬token)
coin   (¬cigar)Wtoken
token   (¬cigar)
```

Notice that, because of the renaming imposed by the interconnection, the axioms are now in the shared language, i.e. they express properties of the same entities. For instance, the regulator now applies to coins and cigars. As a result, the axioms from the different components "interfere" and make new properties (the new requirements) emerge at the level of the resulting system: for instance, the vending machine how requires a token to be able to dispense cigars.

As we have already mentioned in the introduction, this form of composition is the reason why the categorical approach brings software systems into the realm of "general, complex systems", the global behaviour of which is characterised in terms of properties or phenomena that emerge from components and interactions. This is a view that is now shared across different Sciences, from biological to economical and sociological systems [70]. Furthermore, taking the union of the translations of the sets of axioms is logically equivalent to taking their conjunction. Hence, as claimed in the introductory remarks to chapter 4, we are indeed complying with the "conjunction as composition" view used in [116]. In section 6.3, we will present yet another instance of this same "dogma", this time applied to process models.

It is easy to see that this set of axioms is logically equivalent to the one we gave originally in (3.5.11) for the regulated vending machine, which further shows that the lifting performed through the forgetful functor is not unique. Also notice that we recover the original extension (morphism) – *no cigar* – as one of the co-cone projections. Hence, basically, what we have done is factorise the extension by externalising the regulator that was coded over the original specification.

We should emphasise that, having performed the externalisation, the view of the system that interests us is the one given by the (configuration) diagram. The colimit construction is "only" useful as a semantics for the diagram, namely as a means of checking that the required properties will emerge from the interconnections. Further evolution of the system should be performed on the configuration, not the global specification resulting from the colimit.

Other examples of the application of these techniques in Computing can be found in the area of Concurrency Theory, namely in the work of G.Winskel [112], and F.Costa [21]. The idea is that operations on processes like synchronisation can be defined at the level of the actions that the processes can perform (their alphabet) through some algebraic operations and then lifted to the category of processes using a

(co)fibration. An example in this area will be given in section 6.3. Finally, the third part of the book will be dedicated to a systematisation of this process of structuring complex systems through what have been called "software architectures".

6.2 Structured objects and morphisms

One of the best examples of the added expressive power that functors bring into the categorical discourse is the ability to work in frameworks that involve more than one category. An exhaustive study of distinguished objects and arrows with respect to a functor can be found in [1]. In the sequel, we shall present some of the concepts that we have found to provide a good introduction and, thus, motivate the reader to search more about this topic.

6.2.1 Example – realisations of a specification

For instance, we argued in 5.1.3 that the notion of satisfaction of specifications by programs can give rise to functors ***spec: PROG → SPEC*** that map programs to the strongest specification that they satisfy. The notion of satisfaction of a specification *S* by a program *P* can then be captured by the existence of a morphism *σ:S → spec(P)*. Indeed, the morphism expresses the fact that the properties specified through *S* are entailed by the strongest specification of *P*.

Such a "structured" morphism *σ:S → spec(P)* accounts for representations of abstract concepts of the specification in terms of the syntax of the program. That is to say, the morphism reflects design decisions taken during the implementation of *S* in terms of *P*, say the choice of specific representations for the state of a system. Hence, there is an interest in manipulating morphisms of the form *σ:S → spec(P)* as capturing the possible realisations of specifications in terms of programs.

6.2.2 Definition – structured morphisms

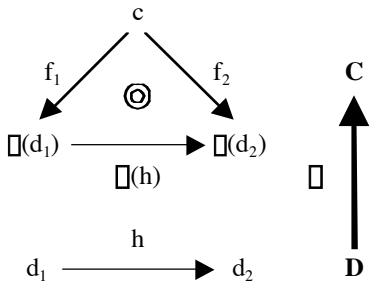
Given a functor *U:**D** → **C***, a *U-structured morphism* is a ***C***-morphism *f:c → U(d)* where *c* is an object of ***C*** and *d* is an object of ***D***.

Structured morphisms can be organised in categories that generalise the notion of comma-category that we studied in section 3.2:

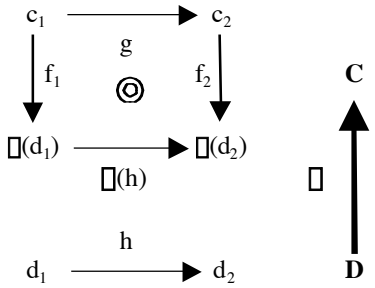
6.2.3 Definition – comma-categories

1. Comma-categories, as defined in 3.3.2, can be generalised to the case of a functor *U:**D** → **C*** and an object *c:**C***, to define a category *c/U* that has for objects all pairs <*f:c → U(d)*,*d*> where f is a *U*-structured mor-

phism with domain c. The morphisms between $\langle f_1 : c \to \Delta(d_1), d_1 \rangle$ and $\langle f_2 : c \to \Delta(d_2), d_2 \rangle$ are all the $\mathbf{D}$-morphisms $h : d_1 \to d_2$ such that $f_1 ; \Delta(h) = f_2$. These categories are called under-cone categories in [22, page 48].



2. The construction above can be further generalised to define the category $\_\downarrow\Delta$ (or $\mathbf{C}\downarrow\Delta$) that has for objects all the Δ-structured morphisms, i.e. triples $\langle c, f : c \to \Delta(d), d \rangle$. The morphisms between $\langle c_1, f_1 : c_1 \to \Delta(d_1), d_1 \rangle$ and $\langle c_2, f_2 : c_2 \to \Delta(d_2), d_2 \rangle$ are all the pairs $\langle g, h \rangle$ of $\mathbf{C}$-morphisms $g : c_1 \to c_2$ and $\mathbf{D}$-morphisms $h : d_1 \to d_2$ such that $f_1 ; \Delta(h) = g ; f_2$.



proof

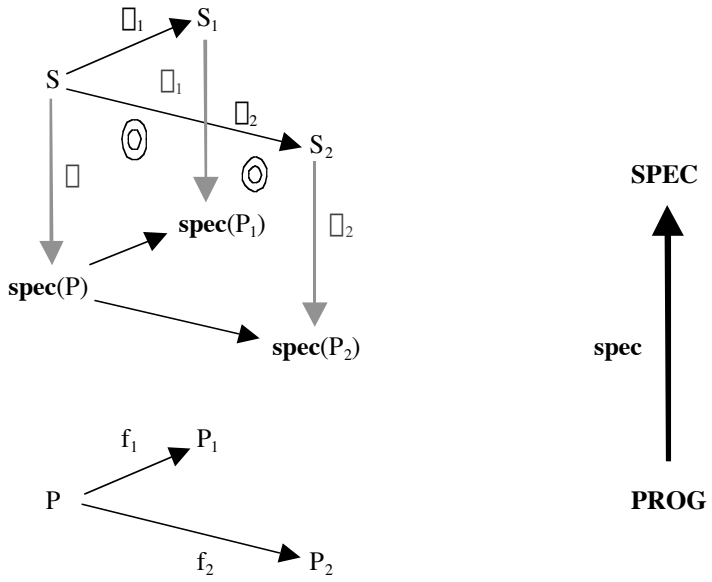
The proof that these constructions do yield categories is left as an exercise. The reader is also invited to consult [79, pages 46-48] for further generalisations of the notion of comma category, including the one that justifies their name!

6.2.4 Remark – compositionality in software development

In chapter 4, we showed how universal constructions could be used to formalise notions of composition that occur in system development. These were applied to system specification in 6.1.24 including an illustration of their role in supporting evolution through the interconnection, in run-time, of new components that can bring about new emergent propoerties.

However, ideally, such constructions should apply to realisations and not to specifications alone in the sense that, when we compose specifications for which we have already correct implementations, we should be able to obtain a correct implementation for the resulting specification as a composition of the implementations of its components. This is usually called *compositionality* of system development. We are now going to show how this form of compositionality can be formulated in this setting and derived from the existence of an abstraction functor.

First of all, we are now interested in extending the notion of realisation to diagrams as capturing configurations of complex systems: a realisation of a specification diagram $\Delta: \mathbf{I} \to \mathbf{SPEC}$ by a program diagram $\Delta: \mathbf{I} \to \mathbf{PROG}$ is an $|\mathbf{I}|$-indexed family $(\Delta_i: \Delta(i) \to \mathbf{spec}(\Delta(i)))_{i \in |\mathbf{I}|}$ of realisations such that, for every $f: i \to j$ in $\mathbf{I}$, $\Delta(f); \Delta_j = \Delta_i; \mathbf{spec}(\Delta(f))$. That is to say, in addition to the individual components, the interconnections have to be realised as well.



Compositionality of the relationship between programs and specifications can be expressed through the fact that the colimit of Δ – the diagram of programs – is a realisation of Δ – the diagram of specifications – in an essentially unique way.

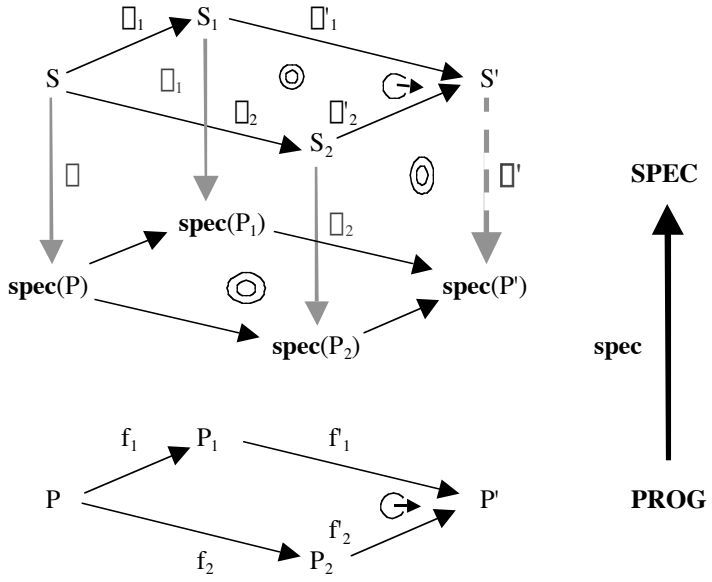
We can actually prove that, as long as ***spec*** is a functor, this property holds regardless of the nature of the specifications and programs. Indeed, because functors preserve the commutativity of diagrams, the image of the program diagram commutes, i.e. in the case of the diagram in the figure,

$$\mathbf{spec}(f_1); \mathbf{spec}(f'_1) = \mathbf{spec}(f_2); \mathbf{spec}(f'_2)$$

This entails that

$\sigma_1;\sigma_1;\mathbf{spec}(f'_1)=\sigma;\mathbf{spec}(f_1);\mathbf{spec}(f'_1)=\sigma;\mathbf{spec}(f_2);\mathbf{spec}(f'_2)=\sigma_2;\sigma_2;\mathbf{spec}(f'_2)$

Because $<\sigma'_1,\sigma'_2,S'>$ is a pushout of $<S,\sigma_1,\sigma_2>$, we can conclude that there exists a unique morphism $\sigma:S'\to \mathbf{spec}(P')$ such that $\sigma_1;\mathbf{spec}(f'_1)=\sigma'_1;\sigma'$ and $\sigma_2;\mathbf{spec}(f'_2)=\sigma'_2;\sigma'$. Notice that these two equations express the fact that the pairs $<\sigma_i;\mathbf{spec}(f'_i)>$ are, indeed, morphisms of realisations. Also notice that ***spec*** is not required to preserve the pushout of programs!



Because compositionality is a property that is not exactly easy to achieve, as more than 20 years of research in the area has shown, this means that the existence of a functor relating two domains of system modelling expresses a very strong structural relationship between them. In [31], we have given examples of situations in which compositionality fails because some of the properties required of functors are not met by the way specifications relate to programs. Basically, what is at stake is the balance that must be struck between the nature of the properties that the abstraction map is able to derive, and the ability of program morphisms to preserve such properties. In [31] it is also shown that, in the absence of a functorial relationship, properties of systems may emerge that result from the need to regulate the interconnections between the components. This happens when the "semantics" of the programming language is not strong enough to ensure the preservation of the properties made observable through the specification language. This is similar

to what happens in social systems in which regulators (e.g. the police) are necessary to enforce laws that are not "natural" but imposed by the society for coordinating the behaviour of individuals.

An example of such a situation can be given in terms of Eiffel class specifications and linear temporal logic. If we had chosen to restrict the behaviours of Eiffel classes to those that are normative in the sense that routines are only executed when their pre-conditions hold, then the following sentence would be included in the strongest specification of every routine:

$$r \quad pre_r$$

That is to say, if the routine r is about to happen, its pre-condition holds. It is easy to see that this property is not preserved by class morphisms. Indeed, given a class morphism f, all that f guarantees is that $F(pre_r) \vdash pre_{F(r)}$. Hence, from $(F(r) \quad pre_{F(r)})$ we cannot infer the translation through F of $(r \quad pre_r)$, i.e. the property $(F(r) \quad F(pre_r))$. This means that an implementation of a class specification that refuses to execute every routine when the pre-condition fails, cannot be reused, with the same semantics of refusal, when the specification is inherited into a larger one. In other words, inheritance does not preserve the property of refusing execution of routines for which pre-conditions fail. This is probably why the semantics of pre-conditions in Eiffel does not include such refusals…

Although the construction in 6.2.4 was motivated by the extension of the notion of realisation to diagrams as compositions of complex systems, it admits a dual reading as extending composition to realisations, i.e. moving to the category $\_\,$***spec*** whose objects are, precisely, the realisations. Actually, this dual reading is supported by a dual visualisation of the figures above: whereas we analysed them from the point of view of "vertical" relationships (the realisations) between "horizontal structures" (the specification and the program configurations), we are now analysing them as horizontal relationships (a configuration) between vertical structures (the realisations of the component specifications).

Indeed, because, for every $f:i \to j$ in $\boldsymbol{I}$, $\sigma(f);\sigma_j=\sigma_i;$***spec***$(\rho(f))$, σ and ρ define a diagram $<\sigma,\rho>:\boldsymbol{I} \to \_\,$***spec*** such that, for every i, $<\sigma,\rho>(i)$ is $<\sigma(i),\sigma_i,\rho(i)>$ and, for every $f:i \to j$, $<\sigma,\rho>(f)$ is $<\sigma(f),\rho(f)>$. Compositionality expresses the existence (and uniqueness) of a colimit for this diagram of realisations when σ and ρ admit colimits.

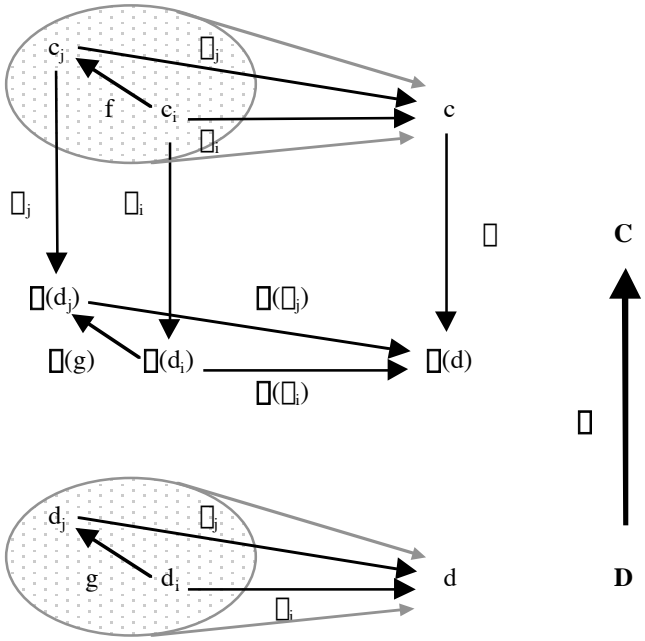
Compositionality in the sense that we have just described is a property that can be stated and proved in the general setting of comma-categories:

6.2.5 Definition/proposition – projection creates colimits

1. Given a functor $F:D \to C$, we define two functors $F_C:\_/F \to C$ and $F_D:\_/F \to D$ by projecting objects and morphisms of the comma-category $\_/F$ to their C and D components, respectively.
2. The functor $\langle F_C, F_D \rangle:\_/F \to C \times D$ creates colimits.

proof

1. Left as an exercise.
2. Consider a diagram $G:I \to \_/F$ and colimits $\gamma:G;F_C \Rightarrow c$ and $\delta:G;F_D \Rightarrow d$ for its projections $G;F_C$ and $G;F_D$, respectively.



- existence of a co-cone in $\_/F$ whose image is the pair of projections: basically, we have to show that there is a morphism $\alpha:c \to F(d)$ such that $\gamma_i;F(\alpha)=\delta_i;\varepsilon$. To prove this, consider the C-co-cone defined by $\{\gamma_i;F(\delta_i)\} \to F(d)$. This co-cone is commutative because, given any morphism $f:c_i \to c_j$ in the base, if $g:d_i \to d_j$ is the morphism in D that, together with f, constitutes a morphism in $\_/F$, we have that

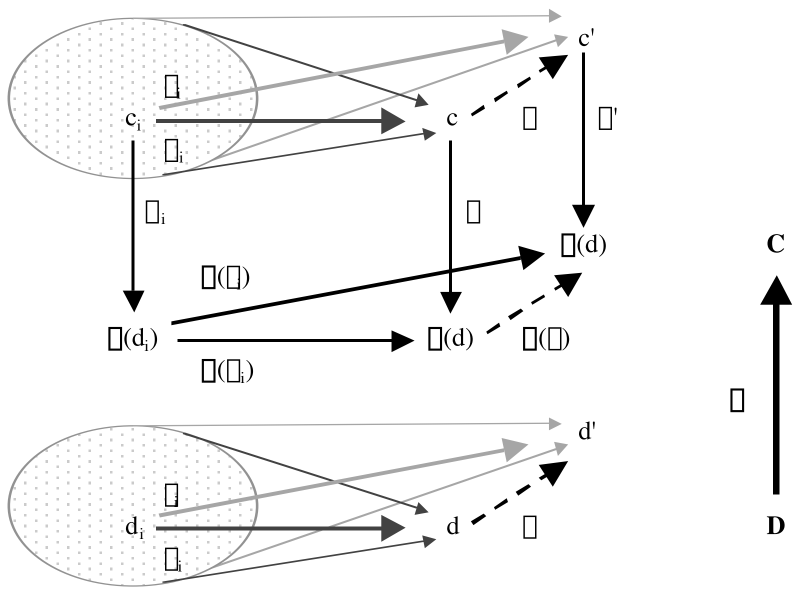
$\delta_i=g;\delta_j$	the D-co-cone is commutative
$F(\gamma_i)=F(g);F(\gamma_j)$	functors preserve composition
$f;\gamma_j=\gamma_i;F(g)$	$\langle f,g \rangle$ is a morphism in $\_/F$
$f;\gamma_j;F(\delta_j)=\gamma_i;F(g);F(\delta_j)$	from the previous equality
$f;\gamma_j;F(\delta_j)=\gamma_i;F(\delta_i)$	applying the second equality

Hence, because $\gamma:G;F_C \Rightarrow c$ is a colimit, there is a (unique) morphism $\alpha:c \to F(d)$ such that $\gamma_i;F(\alpha)=\delta_i;\varepsilon$.

- uniqueness of the co-cone: for the co-cone to be in $\_/□$, the equalities $□_i;□(□_i)=□_i;□$ have to hold and the fact that the co-cone in ***C*** is a colimit implies that □ is the only morphism $c□ □(d)$ that satisfies that property.
- the co-cone is a colimit: let $<□,□>:□□ <c',□',d'>$ be a commutative co-cone. Its projections $□:□;□_C□ c'$ and $□:□;□_d□ d'$ are also commutative. Therefore, because □ and □ are colimits, there are unique $□:c□ c'$ and $□:d□ d'$ such that $□_i=□_i;□$ and $□_i=□_i;□$. We want to prove, first of all, that $<□,□>$ is a morphism of $\_/□$, i.e. $□;□'=□;□(□)$. For that purpose, we are going to prove that the co-cone defined by $(□_i;□(□))$ is commutative: given an arbitrary $<f:c_i□ c_j,g:d_i□ d_j>$ in the base for a structured object $<c_j,□_j,d_j>$,

  $f;□_j;□(□)$
  $=f;□_j;□'$ — $<□_j,□_j>$ being a morphism, $□_j;□(□_j)=□_j;□'$
  $=□_i;□'$ — □ being commutative, $f;□_j=□_i$
  $=□_i;□(□)$ — $<□_i,□_i>$ being a morphism, $□_i;□(□_i)=□_i;□'$

  Hence, there is a unique morphism $□:c□ □(d)$ such that $□_i;□=□_i;□(□)$. But both $□;□'$ and $□;□(□)$ satisfy that property. Therefore, they are equal. Finally, we have to prove that $<□,□>$ is the only morphism satisfying $<□_i,□_i>=<□_i,□_i>;<□,□>$. But this is because any morphism $<□',□'>$ satisfying the same equation is such that $□_i=□_i;□'$ and $□_i=□_i;□'$, which implies $□'=□$ and $□'=□$.



6.3 Functor-structured categories

In this section, we present a construction that yields concrete categories of a very simple nature but with wide applicability:

6.3.1 Definition – functor-structured categories

Let Φ:$\boldsymbol{C}\to$ ***SET*** be a functor. We define the category ***spa***(Φ) whose objects are the pairs $<c,S>$ where c:$\boldsymbol{C}$ and $S\in\Phi(c)$ and whose morphisms f:$<c,S>\to<d,T>$ are the morphisms f:$c\to d$ of $\boldsymbol{C}$ such that $\Phi(f)(S)\subseteq T$.

proof

As already illustrated so many times in section 3.2, we have to prove that the composition law and the identity map inherited from $\boldsymbol{C}$ are applicable.

- Given morphisms f:$<c,S>\to<d,T>$ and g:$<d,T>\to<e,R>$, we have

1.	$\Phi(f)(S)\subseteq T$	because f is a morphism of ***spa***(Φ)
2.	$\Phi(g)(\Phi(f)(S))\subseteq\Phi(g)(T)$	from 1
3.	$\Phi(g)(T)\subseteq R$	because g is a morphism of ***spa***(Φ)
4.	$\Phi(g)(\Phi(f)(S))\subseteq R$	from 2, 3, and transitivity of inclusion
5.	$(\Phi(f);\Phi(g))(S)\subseteq R$	from 4 and function composition in ***SET***
6.	$\Phi(f;g)(S)\subseteq R$	from 5 and the properties of functors.

- The case of the identity map is trivial: given a pair $<c,S>$, the identity on c is such that $\Phi(id_c)(S)=id_{\Phi(c)}(S)=S$.

Functor-structured categories, sometimes also called *spa-categories*, are studied in detail in [1] where some additional terminology is introduced. The objects of such categories are usually called Φ-spaces and their morphisms Φ-maps.

6.3.2 Example – processes

We have already mentioned (see 3.2.1 and the discussion that follows it) that we may regard a pointed set as a signature or alphabet of a process: the proper elements of the set denote actions or events in which the process can get involved, and the designated element denotes an action of the environment, i.e. an action in which the process is not involved.

We can associate with every pointed set $A_\Box$, the set of possible trajectories over $A_\Box$: ***tra***$(A_\Box)=\{\Box:\Box\to A\}$. That is, a trajectory for an alphabet is an infinite sequence of actions; it represents one possible behaviour in a given environment. For instance, in the case of the alphabet $<\{produce,store,\Box_P\},\Box_P>$ of a *producer*, the set of all behaviours in which *produce* and *store* suceed each other is given by:

$$(\Box_P^{*}\texttt{produce}\,\Box_P^{*}\texttt{store})^{\infty}$$

Notice that we make explicit the occurrence of the designated event because we are capturing behaviours that take place in given environments. Hence, finite behaviours can be represented by infinite sequences that, after a certain point, consist of the designated element, i.e. consist only of environment steps.

$$(\Box_P^{*}\texttt{produce}\,\Box_P^{*}\texttt{store})^{*}\Box_P^{\infty}$$

This association defines a functor $\mathbf{tra}:\mathbf{SET}_{\bullet} \to \mathbf{SET}$ if we extend it to morphisms as follows: $\mathbf{tra}(f:A_{\bullet} \to B_{\bullet})(\alpha)=\alpha;f$, i.e. $\mathbf{tra}(f)(\alpha)(i)=f(\alpha(i))$. That is, every morphism of pointed sets induces a translation between the corresponding trajectories by pointwise application of the function between the base sets. We usually denote $\mathbf{tra}(f)$ by $f^{\Box}$.

We can now define the category ***PROC*** of processes as being ***spa(tra)***: a process consists of a pair $<A_{\bullet},\Gamma>$ where $\Gamma \subseteq \mathbf{tra}(A_{\bullet})$, i.e. a process consists of an alphabet and a set of trajectories that capture its behaviour; a process morphism $f:<A_{\bullet 1},\Gamma_1> \to <A_{\bullet 2},\Gamma_2>$ is a morphism $f:A_{\bullet 1} \to A_{\bullet 2}$ between the underlying pointed sets such that $f^{\Box}(\Gamma_1) \subseteq \Gamma_2$, i.e., such that every trajectory of the source is translated to a trajectory of the target.

We have already mentioned in 3.2.1 that, intuitively, a morphism $f:P_1 \to P_2$ identifies process P_2 as a component of process P_1: it identifies, for every action of P_1, what is the participation of P_2 in that action; if the event of P_1 is mapped to the designated event of P_2, this means that P_2 does not participate in it, making it an environment step for P_2. Notice that the preservation of the designated element, as enforced through the morphism, is consistent with the view of P_2 as a component of P_1: an environment step for P_1 must necessarily be an environment step for P_2. Hence, P_1 identifies part of the environment of P_2. The condition on the trajectories requires that every possible behaviour of P_1 (the system) be mapped to one of the allowed behaviours of P_2 (the component). That is to say, the system cannot exhibit behaviours that are not allowed by the component. However, it is possible that behaviours of the component do not show up in the system, e.g. because interactions with other components within the system prevent them from occurring.

Functor-structured categories provide us examples of some of the kinds of categories that we discussed in section 6.1.

6.3.3 Proposition

Let $\Omega:\mathbf{C} \to \mathbf{SET}$ be a functor.

1. The functor $\Omega: \mathbf{spa}(\Omega) \to \mathbf{C}$ that forgets the ***SET***-component of each object defines $\mathbf{spa}(\Omega)$ as a concrete category over $\mathbf{C}$.
2. The forgetful functor $\Omega: \mathbf{spa}(\Omega) \to \mathbf{C}$ is both a split fibre-complete fibration and a split fibre-co-complete cofibration. Given a $\mathbf{C}$-morphism $f:c \to c'$, the cartesian morphism for $<c',S'>$ is $f:<c,\Omega(f)^{-1}(S')> \to <c',S'>$ and the cocartesian morphism for $<c,S>$ is $f:<c,S> \to <c',\Omega(f)(S)>$.

proof

Left as an exercise.

6.3.4 Example – processes

We call ***alph*** the forgetful functor defined by 6.3.3(1) on ***PROC***: it projects processes and their morphisms to the underlying alphabets (pointed sets).

Notice how the (co)cartesian morphisms have an intuitive interpretation in ***PROC***. The cartesian morphism returns, for a given process and alphabet morphism having the process as target, the least deterministic system over that alphabet in which the process can fit, through the morphism, as a component: any additional behaviours of the system would violate the allowed behaviours of the given process.

For instance, consider the alphabet of the *producer/consumer* system defined in 4.3.8, whose proper events are:

$$\texttt{store|retrieve, produce|consume, produce, consume}$$

The cartesian morphism relative to the following behaviour of the *producer* component

$$(\square_P{}^*\texttt{produce}\,\square_P{}^*\texttt{store})^\infty$$

defines the following behaviour for the system

$$(\{\square_S,\texttt{consume}\}^*\{\texttt{produce|consume,produce}\}\{\square_S,\texttt{consume}\}^*\texttt{store|retrieve})^\infty$$

That is to say, we obtain the system trajectories in which the *producer* component alternates between executing *produce* and *store*. Notice how the environment of *producer* is captured by one of

$$\square_S,\ \texttt{consume}$$

In other words, an environment step for the component is either performed by the environment of the whole system ($\square_S$) or by the *consumer* component without interacting with *producer*.

On the other hand, the cocartesian morphism returns, for a given process and alphabet morphism having the process as source, the most deterministic process over the target alphabet that it admits, through the morphism, as a component: any additional behaviours that the component may have will not be able to be observed in the given system.

For instance, consider the system behaviour that consists of all trajectories in which only one exchange is performed between *consumer* and *producer*:

$$\square_S{}^*\texttt{produce}\,\square_S{}^*\texttt{store|retrieve}\,\square_S{}^*\texttt{consume}\,\square_S{}^\infty$$

The cartesian morphism relative to the *consumer* component returns:

$$\square_C{}^*\texttt{retrieve}\,\square_C{}^*\texttt{consume}\,\square_C{}^\infty$$

Any other trajectory in the language of *consumer* will not show up during the execution of the given system. Notice that the cartesian

morphism relative to this behaviour of the *consumer* component defines the following behaviour for the system

$\{\Sigma_S, \texttt{produce}\}^*$
$\texttt{store}|\texttt{retrieve}$
$\{\Sigma_S, \texttt{produce}\}^*$
$\{\texttt{produce}|\texttt{consume}, \texttt{consume}\}$
$\{\Sigma_S, \texttt{produce}\}^\infty$

This process is less deterministic than the one we started from because it captures only restrictions imposed by the *consumer* component. This means that the original system behaviour can only emerge from the interactions of the *consumer* with other components of the system. Indeed, it is easy to check that

$$\Sigma_S^*\texttt{produce}\,\Sigma_S^*\texttt{store}|\texttt{retrieve}\,\Sigma_S^*\texttt{consume}\,\Sigma_S^\infty$$

is in the intersection of the two behaviours defined by the cartesian morphisms relative to both components:

$$\Sigma_C\texttt{retrieve}\,\Sigma_C^*\texttt{consume}\,\Sigma_C^\infty$$

for the *consumer*, and

$$(\Sigma_P^*\texttt{produce}\,\Sigma_P^*\texttt{store})^\infty$$

for the *producer*. We will see below that this intersection, which captures parallel composition, is computed by a limit (pullback). This is another instance of the "conjunction as composition" idea [116] and Goguen's dogma [56].

In summary, the cartesian and cocartesian morphism allow us to gauge the roles that a given process can play through a morphism as a system or as a component.

The characterisation that 6.3.3 provides for (co)cartesian morphisms allows us to derive some useful properties:

6.3.5 Corollary

Let $\Phi: C \to \mathbf{SET}$ be a functor and $\Pi: spa(\Phi) \to C$ the forgetful functor induced by the ***spa***-construction[9].

1. Π lifts limits uniquely. Any limit $\lambda: c \to \Phi$ for $\Phi: I \to spa(\Phi)$ with $\Phi = \langle c_i, S_i \rangle$ is lifted uniquely to $\lambda: \langle c, S \rangle \to \Phi$ where $S = \bigcap_{i \in I} \Phi(\lambda_i)^{-1}(S_i)$.
2. Π lifts colimits uniquely. Any colimit $\lambda: \Phi \to c$ for $\Phi: I \to spa(\Phi)$ with $\Phi = \langle c_i, S_i \rangle$ is lifted uniquely to $\lambda: \Phi \to \langle c, S \rangle$ where $S = \bigcup_{i \in I} \Phi(\lambda_i)(S_i)$.

[9] For readability, we shall omit the forgetful functor when referring to objects and morphisms of ***C*** that are projected from ***spa***(Φ).

6.3.6 Corollary

Let $\Phi: \boldsymbol{C} \to \boldsymbol{SET}$ be a functor.

1. If $\boldsymbol{C}$ is complete so is $\boldsymbol{spa}(\Phi)$, all limits being concrete (6.1.9).
2. If $\boldsymbol{C}$ is co-complete so is $\boldsymbol{spa}(\Phi)$, all colimits being concrete.

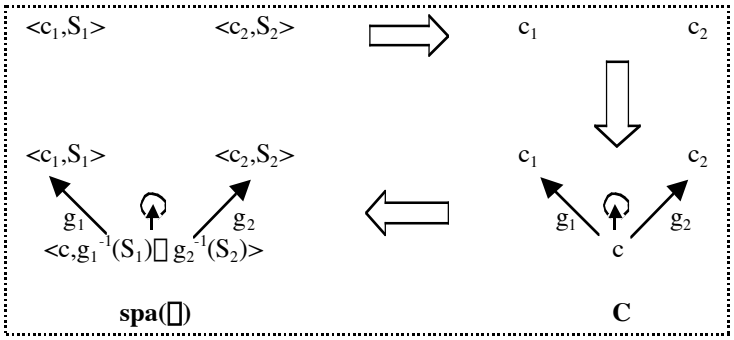
Therefore, we can apply the recipe for calculating limits and colimits in $\boldsymbol{spa}(\Phi)$ that we discussed in section 6.1:

a. project the diagram to the base category and calculate its (co)limit;
b. lift the result by computing the intersection of the inverse images of the set components (for a limit) or the union of the direct images of the set components (for the colimit).

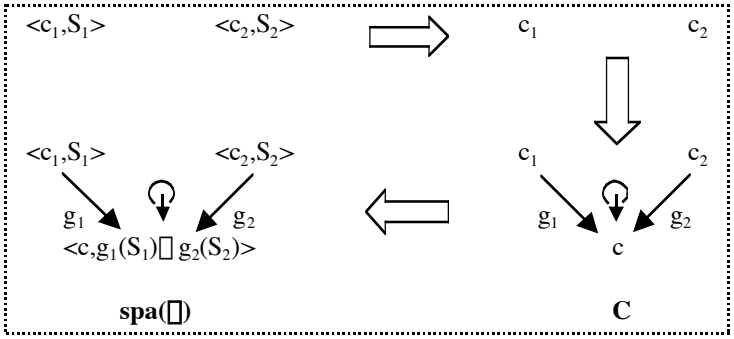
It is useful to see how this construction can be instantiated to particular universal constructions:

- Lifting of the initial and terminal objects:
  If 0_C is an initial object of $\boldsymbol{C}$, then $<0_C,\emptyset>$ is initial for $\boldsymbol{spa}(\Phi)$.
  If 1_C is a terminal object of $\boldsymbol{C}$, $<1_C,\Phi(1_C)>$ is terminal for $\boldsymbol{spa}(\Phi)$.
- Universal constructions in three steps: projection of the $\boldsymbol{spa}(\Phi)$-diagram to $\boldsymbol{C}$; universal construction performed in $\boldsymbol{C}$; lifting the result back to $\boldsymbol{spa}(\Phi)$.

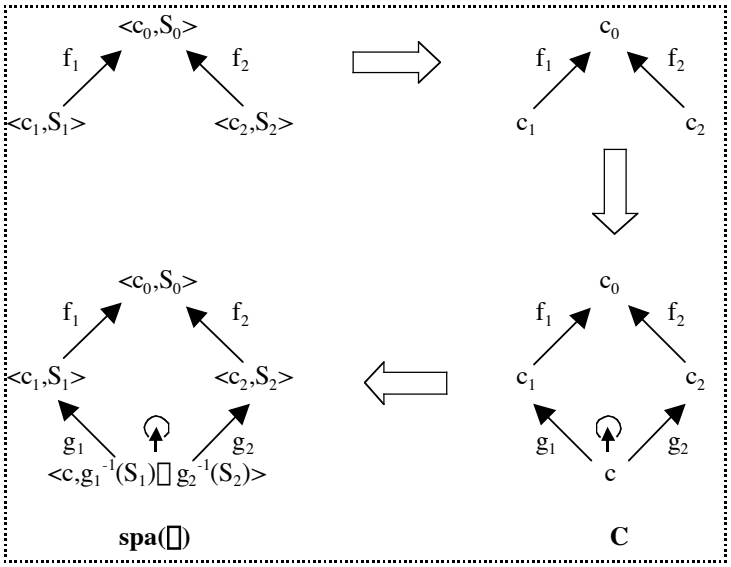
Product:



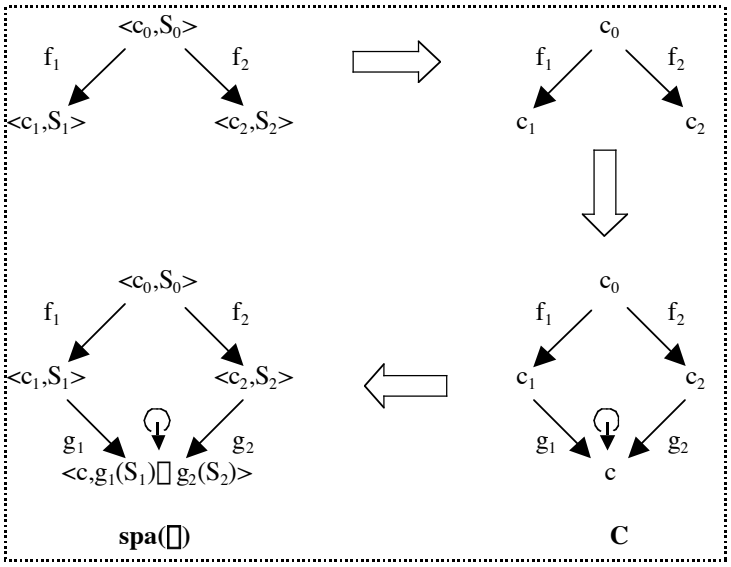
Sum:



Pullback:



Pushout:



Notice that the lifting of the product and the pullback are exactly the same: the "shared" object is only used for determining the ***C***-component of the apex of the pullback and does not interfere with the calculation of the ***SET*** component. This is another example of the kind

of separation that can be achieved between "coordination" and "computation" that we motivated already in section 5.2. We shall return to these aspects in section 7.5 on but this is a simple illustration of the principle.

6.3.7 Example – parallel composition of processes

When applied to processes, these constructions provide us with a mathematical semantics for typical operations of process calculi. Recalling the universal constructions that we studied in chapter 4 for pointed sets, we have:

1. The terminal process is $<\{\Box_\emptyset\},\{\Box_\emptyset \Box\}>$: its alphabet contains only the witness for actions of the environment, and its behaviour reflects exactly that – it witnesses life go by. That is, we have an idle process.
2. The initial process is $<\{\Box_\emptyset\},\emptyset>$: its alphabet is exactly the same as for the terminal process, but its behaviour is completely different – it does nothing, not even witnessing life go by! That is, we have a blocking process that deadlocks any system to which it is interconnected (see below).
3. We have already seen in 4.2.7 that products of pointed sets model the interleaving of alphabets in the sense that they compute the set of all the synchronisation pairs of actions between the components plus the individual actions themselves. When we take into account the behaviour of processes, products return the infinite sequences of such parallel actions that, once projected into the components, result into behaviours of the components. That is to say, the product of processes $<A_1,\Box_1>$ and $<A_2,\Box_2>$ is obtained by computing the product $<A,g_1,g_2>$ of the alphabets and by taking as set of behaviours the intersection of the inverse images of the sets of behaviours of the components. This intersection consists of the sequences $\Box:\Box\Box A$ such that $g_1^\Box(\Box)\Box\Box_1$ and $g_2^\Box(\Box)\Box\Box_2$. Hence, what we obtain is the traditional trace-based semantics of the parallel composition of processes.

   Notice the difference between the idle and the blocking process. When put in parallel with another process, the idle process is "absorbed", i.e. the result of the parallel composition is the other process: this is because the behaviour of the idle process is already present in any other process (except the blocking one). Indeed, the product of a terminal object with any other object is, up to isomorphism, that object. Hence, the idle process does nothing and lets the others do as they please.

   On the contrary, the blocking process "absorbs" any other process with which it is put in parallel: it does nothing and does not let the others do anything. This is because the product of the initial object

with any other process P returns a process with the alphabet of P but with an empty behaviour. As we said at the very beginning of the book, Category Theory is all about the social behaviour of objects…

4. Pullbacks allow us to select only the behaviours that satisfy certain synchronisation requirements. We also saw in 4.3.8 that such requirements are expressed through the morphisms that connect the components to the "channel" that interconnects them. Each action of the channel acts as a point for *rendez-vous* synchronisation in the sense that the actions that participate in a *rendez-vous* are not allowed to occur in isolation, i.e. are not part of the alphabet of the resulting process. In what concerns the resulting behaviour, its computation is exactly as for products: it consists of the sequences of actions determined by the pullback of alphabets that are projected into behaviours of the components. Hence, all the synchronisation mechanism is achieved at the level of the alphabets. In summary, pullbacks model parallel composition with synchronisation.
5. What we have just observed about pullbacks can be generalised to limits in general. The limit of the diagram of alphabets internalises all the interconnections established via the morphisms as synchronisation sets: each component may be involved with at most one action in a synchronisation set. That is, internal synchronisations cannot be established and a given component may not participate in a given *rendez-vous*. When we take into account the behaviour of the processes involved, limits return the infinite sequences of such synchronisation sets that are projected into allowed behaviours of the components.

6.4 The Grothendieck construction

In section 6.1, we showed how every split fibration $\pi:\mathbf{D}\to\mathbf{C}$ defines a functor $\mathit{ind}(\pi):\mathbf{C}^{op}\to\mathbf{CAT}$ that maps every object of $\mathbf{C}$ to its fibre. In the case of a split cofibration, we obtain $\mathit{ind}(\pi):\mathbf{C}\to\mathbf{CAT}$. The objects of $\mathbf{C}$ can be seen as indexes, or types, that are used for classifying the objects of $\mathbf{D}$. The functor π makes the type assignments and the functor $\mathit{ind}(\pi)$ groups the objects according to their types. On the morphism side, the functor $\mathit{ind}(\pi)$ tells us how to translate between fibres when moving from one type to another through a type morphism. For fibrations, this translation is contravariant (like when taking inverse images), whereas for cofibrations the translation is covariant (like when taking direct images).

For instance, if $\mathbf{C}$ is the inheritance hierarchy of an object-oriented system, and $\mathbf{D}$ models the population of objects, we can think of a cofi-

bration that assigns to every object the type of which it is a direct instance. For every inheritance morphism, the cocartesian morphisms define how each object of the child class is also an instance of the parent type (which feature renaming applies, etc). The ***CAT***-based functor returns, for each type, the population of direct instances associated with the type and, for each inheritance morphism, the functor that maps instances of the child to instances of the parent.

The fact that we work with a fibration or a cofibration, and that the associated ***CAT***-based functor is contravariant or covariant, is not very important. After all, it is all a matter of arrow direction. However, as we have already pointed out, it is important that the direction of the arrows is chosen so as to be "natural" in the context in which they are going to be used. For instance, in the case of inheritance, hierarchies are traditionally depicted as graphs with the arrows pointing to the parents, and it would be counterintuitive to formalise them with morphisms going in the opposite direction. All this to say that, in this section, one direction will be preferred for definitions and results, but examples will be given in their "natural" direction. The reader is encouraged to detail the dual constructions as an exercise.

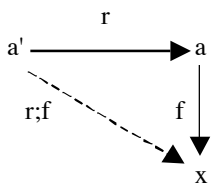
We have also mentioned that indexed categories are very much part of the categorical folklore danced in Computing [e.g. 12,22,104]. We will follow [104] more closely because it shares with us the same kind of "engineering" inspiration that we set ourselves to promote. The reader will also find in [104] many more examples and a much more extended coverage of this topic. Take this section as an appetiser to what can be a much bigger meal.

6.4.1 Definition – indexed categories

An indexed category over a category ***I*** (of indexes) is a functor of the form $\mathbf{I}^{op} \to \mathbf{CAT}$.

6.4.2 Example – comma-categories

In 3.2.2 we showed how, given a category ***C*** and an object $a:\mathbf{C}$, we can define the category of objects under a – $a/\mathbf{C}$. Given a morphism $r:a' \to a$, there is a natural way of translating objects under a to objects under a': given $f:a \to x$, we map it to $(r;f):a' \to x$.



It is easy to prove that this translation is indeed a functor $a \to C \to a' \to C$ so that we obtain an indexed category $\_ \to C : C^{op} \to CAT$.

6.4.3 Exercise

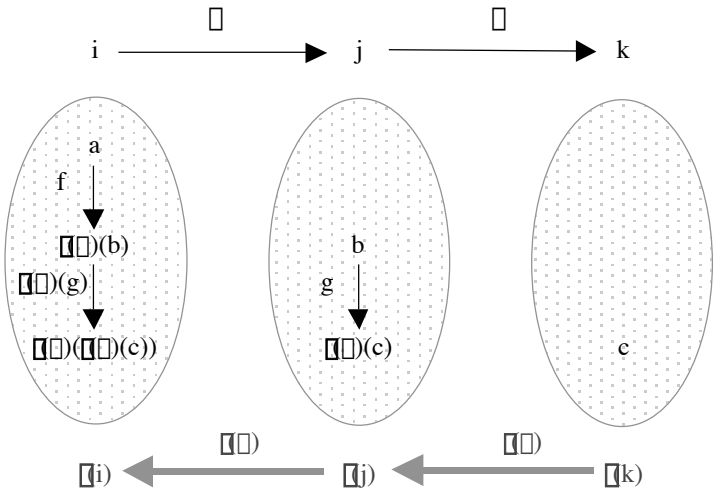
Complete the previous example by doing the proofs and generalising it to the comma-categories of the form $a \to I$ as defined in 6.2.3.

Once we look at functors $I^{op} \to CAT$ as performing an indexing of certain classes of objects, it seems natural to think about the amalgamation of all these classes into a single population, using the indexes to distinguish between the objects according to their provenance. This operation is traditionally called the "Grothendieck construction" [12,22] in honour of A.Grothendieck. We will use the less "flattering" and, perhaps, more "flatfooted" terminology proposed in [104], and refer to this operation as "flattening" the indexed category.

6.4.4 Definition – flattening an indexed category

Given an indexed category $I^{op} \to CAT$ we define a category $FLAT(I)$ as follows:

1. The objects of $FLAT(I)$ are all the pairs $\langle i,a \rangle$ where $i:I$ is an index and $a:I(i)$ is an object of "type" i.
2. The morphisms $\langle i,a \rangle \to \langle j,b \rangle$ are all the pairs $\langle \alpha,f \rangle$ where $\alpha : i \to j$ is a morphism of indexes and $f : a \to I(\alpha)(b)$ is a morphism in $I(i)$.
3. The composition of morphisms is defined componentwise: given $\langle \alpha,f \rangle : \langle i,a \rangle \to \langle j,b \rangle$ and $\langle \beta,g \rangle : \langle j,b \rangle \to \langle k,c \rangle$, we define $\langle \alpha,f \rangle ; \langle \beta,g \rangle = \langle \alpha ; \beta ; f ; I(\alpha)(g) \rangle$.
4. The identity for an object $\langle i,a \rangle$ is $\langle id_i ; id_a \rangle$.



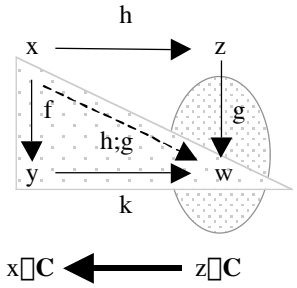
proof

The proof that a category is defined in this way is left as an exercise.

6.4.5 Example

Consider the flattening of the indexed category $\_\Box C :: C^{op} \Box CAT$ defined by the comma-category construction as studied above. The objects of $FLAT(\_\Box C)$ are triples $\langle x, f:x \Box y, y \rangle$ and the morphisms $\langle x, f:x \Box y, y \rangle \Box \langle z, g:z \Box w, w \rangle$ are the pairs $\langle h,k \rangle$ such that $h:x \Box z$, $k:y \Box w$ and $h;g=f;k$.

The equation results from the requirement that k be a morphism in $x \Box C$ between $f:x \Box y$ and $h;g:x \Box w$ which is the translation of $g:z \Box w$ defined by h. This category is usually called the "arrow category of C".



6.4.6 Exercise

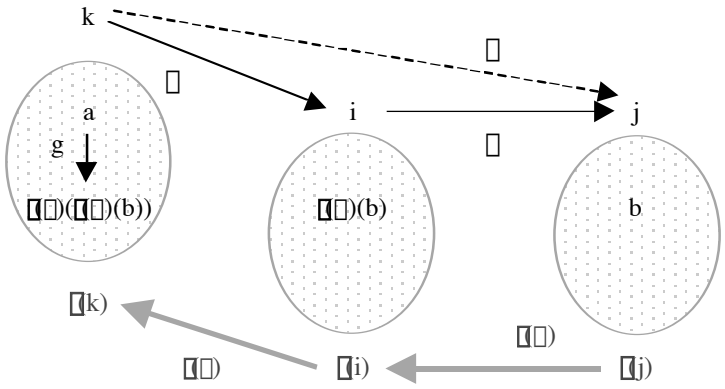
Prove that the category $\_\Box\Box$ defined in 6.2.3 "is" the flattening of the indexed-category of exercise 6.4.3.

The flattened indexed category comes equipped with some structure:

6.4.7 Proposition

Consider an indexed category $\Box I^{op} \Box CAT$.

1. We define a functor $fib(\Box):FLAT(\Box) \Box I$ by projecting objects and morphisms to their I-components.
2. This functor is faithful, defining $FLAT(\Box)$ as a concrete category over I.
3. This functor is a split fibration: the cartesian morphism associated with $\Box:i \Box j$ and $\langle j,b \rangle$ is $\langle \Box, id_{\Box(b)} \rangle : \langle i, (\Box\Box)(b) \rangle \Box \langle j,b \rangle$. That is to say, we simply use the translation mechanism of the indexed category to perform the required lift.



proof

We leave the proof of the first two properties to the reader.

3. We have to prove that the proposed cartesian morphisms satisfy the required properties.
   - First of all, it is easy to see that the morphism is well defined: $(f_i)(b)$ is indeed indexed by i. Moreover, its projection over $\mathbf{I}$ is f as required.
   - Consider now the universal property. Let $\langle f,g\rangle:\langle k,a\rangle \to \langle j,b\rangle$ and $f':k \to i$ satisfy $f=f';f$. Because, by definition, $g:a \to (f_i)(b)$ and $(f_i)(b)=(f';f)(b)=(f_{f'})((f_i)(b))$, we obtain $\langle f,g\rangle:\langle k,a\rangle \to \langle i,(f_i)(b)\rangle$. On the one hand, this morphism is projected to f as required. On the other hand, $\langle f,g\rangle=\langle f,g\rangle;\langle f,id_{(f_i)(b)}\rangle$ as required. The uniqueness property results from the fact that the second component of the cartesian morphism is an identity and the projection over $\mathbf{I}$ is faithful.

The fact that the fibration is split is trivially checked.

This result allows us to derive properties of flattened indexed categories as corollaries of the general results that we proved about fibrations. A useful result is the following:

6.4.8 Proposition

Consider an indexed category $\mathbf{I}^{op} \to \mathbf{CAT}$. If $\mathbf{I}$ is complete, (i) is complete for every $i:\mathbf{I}$, and every translation functor (f) preserves limits, then $\mathbf{FLAT}()$ is complete.

Examples of indexed-categories, their properties, and some of their uses in Computing are given in the next section.

6.5 Institutions

I am proud to say that I grew up into Category Theory when trying to learn the concept of Institution put forward by Goguen and Burstall in

the early 80s! However, and although the fascination is still there, the reasons for including this section in this book are more than sentimental. The Theory of Institutions is a brilliant piece of "engineering mathematics", both in form and content. It is the result of a process of abstraction that has allowed computer scientists to classify and relate the specification formalisms that they have been developing or working with, as well as construct new formalisms in a systematic and integrated way. That is to say, it is a piece of Science, one that Computing can proudly boast.

Like every abstraction, institutions have their limitations. But, like in every other branch of Science, these limitations have inspired many other computer scientists to develop ramifications that have further enlightened the path of those of us who are in the business of systematising and supporting the process of software development. Hence, our own account of the basics of institutions is left here both for exemplifying some of the constructions that we developed so far in the book, and as a source of inspiration for the reader.

Some readers, and hopefully users of this book, will complain that this topic comes too late: for instance, it does not require the material that we introduced so far in this section. While this is clearly true, the reason is, like for all the other examples, that we deliberately placed Institutions where we think they "belong". The point, once again, is that the purpose of this section is not to promote institutions but, rather, illustrate the constructions that we have been defining with the best examples we know. The community has been longing for a book on the Theory of Institutions but this section is no substitute for it.

We start by defining not the original notion of institution as it appears in [61], but a variant that we developed in [45] called π-institution. This is not for self-promotion but because it will allow us to capitalise on some material already introduced in previous sections. We defined in section 3.6 the notion of closure system as consisting of a pair $<L,c>$ where L is a set and $c:L \to L$ is a total function satisfying the properties of a closure operator – reflexivity, idempotence and monotonicity. The idea is that the elements of L are the well-formed formulae that result from a specific choice of vocabulary symbols and a given grammar. The closure operator captures the notion of consequence of the logic at stake, instantiated to that particular language.

The idea of (π-)institutions is to add to this abstract view of Logic the notion that the notion of consequence is independent of the specific choice of vocabulary symbols that determine the language L. More precisely, it makes explicit the distinction between logical and non-logical symbols in a language: whereas the former (usually called the *connectives* of the logic) are captured by the grammar, the latter are given by (typed) sets. The notion of consequence is then constrained

to be invariant under changes of non-logical symbols in order to make it depend only on the connectives. That is to say, we group together several of these closure systems that we regard as being different instantiations of the same logic through different choices of non-logical symbols, and take that set as defining that logic.

Basically, this is what we did in section 3.5 when defining linear temporal logic: we introduced the notion of signature as a means of modelling the instantiations of the general structures of temporal logic, and made explicit the grammar that defines the connectives. For every specific signature, we obtain a closure system whose language is determined by the application of the grammar of temporal logic to the signature, and whose closure operator is obtained through a notion of consequence derived from the traditional Kripke semantics of temporal logic. The idea that all these closure systems are related in a way that makes them instantiations of the same logic can be captured by the following property:

6.5.1 Proposition

The mapping ***prop*** defined in 3.5.2 and the translations induced by signature morphisms as defined in 3.5.7 extend to a functor ***ltl***:***SET*** → ***CLOS***.

proof

The crux of the proof is in showing that signature morphisms map to morphisms of closure systems. But this is a direct consequence of the presentation lemma 3.5.10. The reader is invited to fill-in the details.

Through the functor that maps closure systems to their underlying languages, ***ltl*** maps each signature to the set of well-formed formulae over that signature. The fact that we have a functor means that changes of vocabulary induce corresponding translations at the level of the languages, thus capturing the idea of "uniformity" that we tend to associate with a grammar. The same applies to the closure operator. The fact that changes of vocabulary induce morphisms between the corresponding closure systems captures the flavour of uniformity and continuity that we associate with a logic. Naturally, we can make these "flavours" concrete by making the grammar explicit rather than implicit. It all depends on the level of abstraction at which one wants to work. The reader interested in these aspects should consult [62].

6.5.2 Definition – π-institution

A π-*institution* consists of a pair <***SIGN***,***clos***> where ***SIGN*** is a category (of signatures) and ***clos***:***SIGN*** → ***CLOS*** is a functor.

6.5.3 Remark

An equivalent definition of a π-institution, given in [45], consists of

- a category ***SIGN***
- a functor $\textbf{\textit{gram}}:\textbf{\textit{SIGN}} \to \textbf{\textit{SET}}$
- for every $\Sigma:\textbf{\textit{SIGN}}$, a relation $\vdash_\Sigma : 2^{\textbf{\textit{gram}}(\Sigma)} \times \textbf{\textit{gram}}(\Sigma)$ satisfying the following properties:
  - for every $p \in \textbf{\textit{gram}}(\Sigma)$, $p \vdash_\Sigma p$
  - for every $p \in \textbf{\textit{gram}}(\Sigma)$ and $\Gamma_1, \Gamma_2 \subseteq \textbf{\textit{gram}}(\Sigma)$, if $\Gamma_1 \subseteq \Gamma_2$ and $\Gamma_1 \vdash_\Sigma p$ then $\Gamma_2 \vdash_\Sigma p$
  - for every $p \in \textbf{\textit{gram}}(S)$ and $F_1, F_2 \subseteq \textbf{\textit{gram}}(S)$, if $F_1 \vdash_S p$ and $F_2 \vdash_S p'$ for every $p' \in F_1$, then $F_2 \vdash_S p$
  - for every $\sigma:\Sigma \to \Sigma'$, $p \in \textbf{\textit{gram}}(\Sigma)$ and $\Gamma \subseteq \textbf{\textit{gram}}(\Sigma)$, $\Gamma \vdash_\Sigma p$ implies $\textbf{\textit{gram}}(\sigma)(\Gamma) \vdash_{\Sigma'} \textbf{\textit{gram}}(\sigma)(p)$.

Notice that the functor ***gram*** is the composition of ***clos*** with the forgetful functor that maps closure systems to the underlying languages. The closure operator itself is derived from the consequence relation as follows: for every $\Gamma \subseteq \textbf{\textit{gram}}(\Sigma)$, $c_\Sigma(\Gamma) = \{p \in \textbf{\textit{gram}}(\Sigma) : \Gamma \vdash_\Sigma p\}$. On the other hand, every closure operator defines a consequence relation: $\Gamma \vdash_\Sigma p$ iff $p \in c_\Sigma(\Gamma)$.

The definition given in [45] further requires the consequence relation to be compact, a property that we will not need for the constructions that we are going to present.

6.5.4 Remarks

1. Given the equivalence between the two definitions, we shall often use the consequence relation and the grammar function without notifying the reader.
2. Because the notation can become quite cumbersome, we shall often omit the reference to the grammar functor when applied to morphisms and write $\sigma(p)$ instead of $\textbf{\textit{gram}}(\sigma)(p)$ like we did for temporal logic in section 3.5.

When defining a π-institution, the consequence relation or the closure operator can be presented in many different ways. One of the possible ways is the one we adopted in the definition of linear temporal logic: by providing a notion of model and a satisfaction relation. This is what institutions, as defined by Goguen and Burstall [61], consist of.

6.5.5 Definition – institution

An *institution* is a quadruple $\langle \textbf{\textit{SIGN}}, \textbf{\textit{gram}}, \textbf{\textit{mod}}, \models \rangle$ where

- ***SIGN*** is a category
- $\textbf{\textit{gram}}:\textbf{\textit{SIGN}} \to \textbf{\textit{SET}}$ is a functor
- $\textbf{\textit{mod}}:\textbf{\textit{SIGN}}^{op} \to \textbf{\textit{CAT}}$ is a functor

- for every Σ:***SIGN***, $\models_\Sigma$:***mod***(Σ)$\to$***gram***(Σ) satisfies for every morphism $\sigma:\Sigma\to\Sigma'$, $p\in$***gram***(Σ) and $M'\in$***mod***(Σ'), ***mod***(σ)(M') $\models_\Sigma p$ iff $M'\models_{\Sigma'}$***gram***(σ)(p).

The functor ***mod*** provides, for every signature, the category of models that can be used to interpret the language defined by ***gram*** over that signature. Signature morphisms induce translations between these classes of models in the opposite direction. The idea is that, as seen in 3.5.9 for linear temporal logic, sentences in the language of the source signature can be interpreted in a model for the target signature through the interpretation of their translations. The condition that is required on the satisfaction relation, which is usually called the *satisfaction condition*, states precisely this property – that satisfaction is invariant under change of notation.

When reasoning about institutions, we normally use the simplified notation for the ***gram*** functor that we mentioned for π-institution. We also adopt a similar simplification for the ***mod*** functor: for every morphism $\sigma:\Sigma\to\Sigma'$ and $M'\in$***mod***(Σ'), we normally write $M'|_\sigma$ instead of ***mod***(σ)(M'). For every signature Σ, $\Gamma\subseteq$***gram***(Σ), and $M\in$***mod***(Σ), we usually write $M\models_\Sigma\Gamma$ meaning that $M\models_\Sigma p$ for every $p\in\Gamma$.

6.5.6 Proposition

Every institution <***SIGN***,***gram***,***mod***,$\models$> presents the π-institution <***SIGN***,***gram***,$\vdash$> where, for every signature Σ, $p\in$***gram***(Σ) and $\Gamma\subseteq$***gram***(Σ), $\Gamma\vdash_\Sigma p$ iff, for every $M\in$***mod***(Σ), $M\models_\Sigma\Gamma$ implies $M\models_\Sigma p$.

proof

The proof that a π-institution is obtained in this way offers no difficulties and is left as an exercise.

Another way of presenting the closure operator or consequence relation of a π-institution is through the notion of proof, choosing an inference system for the logic. See [87] for such proof-theoretic presentations and their integration in a more comprehensive categorical account of logical systems that the author calls "General Logics".

6.5.7 Remark – models defined via a split (co)fibration

The model functor of an institution is defined in 6.5.5 directly as an indexed category, signatures providing the indexes. As seen in 6.1.20, indexed categories can be presented as split (co)fibrations, which is how, in many cases, the model-theory of some institutions is more naturally defined. That is to say, it is often more intuitive, not to say "practical", to provide directly either a split fibration ***sign***: ***MODL***$\to$***SIGN*** or a split cofibration ***sign***: ***MODL***$\to$***SIGN***^{op} for some category ***MODL*** of models and take ***ind(sign)*** as the model functor of the required institution. This happens when we already have a notion of model that corre-

sponds to the domain of interpretation intended for a given specification formalism, and we want to use it as the "semantics functor" of the institution.

An example can be given through linear temporal logic.

6.5.8 Example – linear temporal logic over *PROC*

Temporal signatures were interpreted, in 3.5.3, over infinite sequences of sets of atomic propositions corresponding to synchronisation sets of actions. Yet, from the point of view of modelling concurrent systems, such infinite sequences do not represent abstractions of full behaviour; they usually represent one single behaviour among many that may also be observed on a given system. Hence, from the point of view of providing a domain of interpretation for a formalism supporting concurrent system specification, sets of such trajectories are more meaningful in the sense that they can be taken to represent *full* behaviours. Put in another way, it we take single sequences as models of full behaviour, we are restricting the expressive power of the formalism to a much smaller class of systems: those that are deterministic and run in a deterministic environment.

We have already seen in section 6.3 how such a trace-based model of process behaviour can be organised in the functor-structured category ***spa(tra)*** that we called ***PROC***, where ***tra***:$\boldsymbol{SET_{\square}} \to$ ***SET*** is the functor that generates and translates between sets of traces. Hence, it seems intuitive that we investigate the use of this category for providing the models of an institution of linear temporal logic. However, the split cofibration defined by ***spa(tra)*** is over $\boldsymbol{SET_{\square}}$, not over $\boldsymbol{SET^{op}}$ as required for the temporal signatures. Indeed, processes were defined over arbitrary alphabets of actions whereas, in the case of temporal logic, propositions are interpreted over traces of synchronisation sets. Hence, some adaptation is required. There are two equivalent ways in which this adaptation can be performed.

The first approach is to work with the subcategory of ***spa(tra)*** that involves only alphabets of synchronisation sets. We have seen that powersets can be regarded as pointed sets, the empty set providing the distinguished element. More precisely, we saw in 3.3.2 that, by choosing the morphisms $2^B \to 2^A$ to be the inverses of the functions $A \to B$, we define a subcategory of $\boldsymbol{SET_{\square}}$ that we called ***POWER***. Therefore, we can particularise the ***spa*** construction that yields processes to the functor ***ptra***: ***POWER*** $\to$ ***SET*** that yields the set of infinite traces taken over sets of actions. Finally, there is a straightforward (and, as we shall see in the next chapter, powerful) way of relating $\boldsymbol{SET^{op}}$ to ***POWER***: through the contravariant functor $\boldsymbol{2^-}$ that maps every set A to its powerset 2^A and every function $A \to B$ to the map $2^B \to 2^A$ that computes inverse images according to that function. Hence, we can particularise ***tra*** to ***ftra***: ***SE-***

$\mathbf{T}^{op} \to \mathbf{SET}=2^{-};tra$ and use the split cofibration that ***spa(ftra)*** defines over $\mathbf{SET}^{op}$ to define, as detailed in 6.1.20(2), the model functor $ind(spa(ftra)):\mathbf{SET}^{op} \to \mathbf{CAT}$ of an institution of linear temporal logic: the one that associates every temporal signature (set of actions) with the category of processes whose alphabet consists of the subsets of the signature (synchronisation sets of actions).

An alternative approach is to restrict not the (co)fibration that generates the indexed category but the indexed category generated by the original (co)fibration, i.e. to work with the composition $2^{-};ind(spa(tra))$. This is simpler because it does not require the definition of a new (co)fibration. For the same reasons, this approach is less "satisfying" because it is less "structural" in the sense that it does not characterise explicitly the class of models that are chosen. In the case of linear temporal logic, both approaches lead to the same indexed category. Indeed, taking the indexed category $\mathbf{SET}_{\Sigma} \to \mathbf{CAT}$ generated by ***spa(tra)***, we can compose it with the contravariant powerset functor $\mathbf{SET}^{op} \to \mathbf{SET}_{\Sigma}$ to define the required model functor $\mathbf{SET}^{op} \to \mathbf{CAT}$.

It remains to define the satisfaction relation of this institution. Given that models over a signature Σ consist of sets of traces $P \subseteq (2^{\Sigma})^{\omega}$, satisfaction of a temporal proposition by a process is defined as satisfaction by all the traces of the process as defined in 3.5.3. The satisfaction condition results from the properties proved in 3.5.9. Indeed, the reader should check that the notion of reduct defined therein coincides with the functor between fibres defined by the cofibration (6.1.16)

It is easy to see that the institution defined over this indexed category yields the same π-institution as the one that can be defined directly by applying definition 6.5.2 to the functor defined in 6.5.1 (which is based on a semantic interpretation over single sequences as given in 3.5.3). This is because the ***PROC***-based institution and the one that uses single trajectories as in 3.5.3 give rise to the same consequence relation. However, we shall see that they satisfy different structural properties, which shows that, "forgetting" the model-theory to retain just the consequence relations, π-institutions are, indeed, more abstract than institutions..

6.5.9 Remark – when a split (co)fibration is not available

The availability of a split cofibration for the definition of the model functor is very useful because it allows us to work with reduct functors that operate on the chosen semantic models and, therefore, compare the structures that models and theories provide for specification. However, it may happen that we have a notion of semantic model in mind that can still be captured by a functor $sign: \mathbf{MODL} \to \mathbf{SIGN}^{op}$ but one that is not a cofibration. The problem here is the definition of a suitable translation between models.

This problem can be solved by working not directly over the fibres defined by the functor but with what we could call "generalised" models: those that are provided by structured morphisms as defined in 6.2.2. That is to say, the idea is, for every signature Σ, to work with models of the form $\langle \Sigma : \Sigma \to \mathbf{sign}(M), M \rangle$ so that $\mathbf{mod}(\Sigma)$ is the category $\Sigma/\mathbf{sign}$, making $\mathbf{mod}$ be the functor $\_/\mathbf{sign} : \mathbf{SIGN}^{op} \to \mathbf{CAT}$ defined on morphisms $\sigma : \Sigma \to \Sigma'$ by the functor $\sigma' / \mathbf{sign} : \Sigma'/\mathbf{sign} \to \Sigma/\mathbf{sign}$ that maps $\langle \sigma' : \Sigma' \to \mathbf{sign}(M'), M' \rangle$ to $\langle \sigma ; \sigma' : \Sigma \to \mathbf{sign}(M'), M' \rangle$. (The reader is encouraged to fill-in the way the functor works on morphisms between models.) In a way, what this construction does is compensate for the lack of (co)cartesian morphisms by providing an explicit "adaptor" that maps the signature to the language of the model. As a result, instead of translating directly between models, reducts operate "syntactically" at the level of these adaptors.

The reader may be wondering whether this is not just a contrived way of bringing structured morphisms into the discussion and a good example of "abstract non-sense": an accusation often made to categorical techniques when developed without a clear application in mind, in a self-justified process of "structure for structure"… Certainly not in this case (nor in any other case in this book): these generalised models are quite common in Modal Logic (e.g. [66])! Indeed, models for a modal logic, normally called frames or *Kripke structures*, can be regarded as concrete categories over a base category of possible worlds. The adaptors that we defined correspond to interpretation functions that, for each atomic proposition, return the set of possible worlds in which the proposition is true. A generalised model, consisting of a Kripke frame and an interpretation function, is what in Modal Logic is usually called a model.

In the case of linear temporal logic, this means that interpretation structures are no longer traces $\Sigma \to (2^{\Sigma})^{\omega}$ as defined in 3.5.3 – where we mentioned that they are *canonical* Kripke structures for linear, discrete, propositional logic [114] – but triples $\langle W, \Sigma \to W^{\omega}, v : \Sigma \to 2^{W} \rangle$ that correspond to the more traditional notion of models: W is a set of possible worlds (events), the pair $\langle W, \Sigma \to W^{\omega} \rangle$ defines a linear frame (Kripke structure), and the function v returns the set of worlds (events) in which each atomic proposition (action) holds (occurs).

In the case that interests us, ***MODL*** is the original ***spa(tra)*** but ***sign*** is not the original cofibration but its composition with the powerset functor $\mathbf{SET}_{\omega} \to \mathbf{SET}^{op}$ (in fact, as we shall see in 7.3.6, the left adjoint of the powerset functor used in 6.5.8). This means that reducts are not operated by the cofibration and, hence, we loose some of the structural properties of processes. More details on these constructions can be found in [32,33].

In Logic, models are considered to be the "duals" of theories. In institutions, this duality can be explored and made explicit in several ways. We can start by realising that the theories of the underlying closure systems can be organised in a category that provides itself a split (co)fibration over the category of signatures. Indeed, we have already proved in 6.1 that the category $\boldsymbol{THEO_{LTL}}$ of linear temporal theories, as defined in 3.5.4, is both a split fibration and a split cofibration over ***SET***. The extension to any π-institution is trivial:

6.5.10 Proposition – the (co)indexed-category of theories

1. Every π-institution $\langle \boldsymbol{SIGN},\boldsymbol{clos}\rangle$ defines an extension $\boldsymbol{theo}:\boldsymbol{SIGN}\to\boldsymbol{CAT}$ of ***clos*** by mapping every signature Σ to the category $\boldsymbol{THEO}_{\boldsymbol{clos}(\Sigma)}$ of the theories over $\boldsymbol{clos}(\Sigma)$, and every signature morphism $\sigma:\Sigma\to\Sigma'$ to the functor $\boldsymbol{theo}(\sigma):\boldsymbol{THEO}_{\boldsymbol{clos}(\Sigma)}\to\boldsymbol{THEO}_{\boldsymbol{clos}(\Sigma')}$ defined by $\boldsymbol{theo}(\sigma)(\Gamma)=c'(\boldsymbol{clos}(\sigma)(\Gamma))$.
2. Every π-institution $\langle \boldsymbol{SIGN},\boldsymbol{clos}\rangle$ defines a contravariant functor $\boldsymbol{theo}^{-1}:\boldsymbol{SIGN}^{op}\to\boldsymbol{CAT}$ by mapping every signature Σ to the category $\boldsymbol{THEO}_{\boldsymbol{clos}(\Sigma)}$ of the theories over $\boldsymbol{clos}(\Sigma)$, and every signature morphism $\sigma:\Sigma'\to\Sigma$ to the functor $\boldsymbol{theo}^{-1}(\sigma):\boldsymbol{THEO}_{\boldsymbol{clos}(\Sigma)}\to\boldsymbol{THEO}_{\boldsymbol{clos}(\Sigma')}$ defined by $\boldsymbol{theo}^{-1}(\sigma)(\Gamma)=\boldsymbol{clos}(\sigma)^{-1}(\Gamma)$.

proof

Notice that, because the category of theories over a closure system is, in fact, a pre-order, the functors $\boldsymbol{theo}(\sigma)$ and $\boldsymbol{theo}^{-1}(\sigma)$ do not have to be defined on morphisms. However, we are required to prove:

1. Given theories $\Gamma\le\Delta$ in $\boldsymbol{THEO}_{\boldsymbol{clos}(\Sigma)}$, $\boldsymbol{theo}(\sigma)(\Gamma)\le\boldsymbol{theo}(\sigma)(\Delta)$. This holds because ***clos*** is a functor. More precisely, given $\Gamma\vdash\Delta$ in $\boldsymbol{THEO}_{\boldsymbol{clos}(\Sigma)}$, we have $\boldsymbol{clos}(\sigma)(\Gamma)\subseteq\boldsymbol{clos}(\sigma)(\Delta)$ because $\boldsymbol{clos}(\sigma)$ is a normal function between the languages of the two closure systems, which implies $c'(\boldsymbol{clos}(\sigma)(\Gamma))\vdash c'(\boldsymbol{clos}(\sigma)(\Delta))$ because closure operators are monotonic with respect to set inclusion.
2. Given theories $\Gamma\le\Delta$ in $\boldsymbol{THEO}_{\boldsymbol{clos}(\Sigma)}$, $\boldsymbol{theo}^{-1}(\sigma)(\Gamma)\le\boldsymbol{theo}^{-1}(\sigma)(\Delta)$. Given $\Gamma\vdash\Delta$, we have $\boldsymbol{clos}(\sigma)^{-1}(\Gamma)\subseteq\boldsymbol{clos}(\sigma)^{-1}(\Delta)$ because $\boldsymbol{clos}(\sigma)$ is a normal function between the languages of the two closure systems and the inverse image of a closed set is itself closed.

The reader is invited to complete the proof as an exercise.

The (co)flattening of ***theo*** as a (co)indexed-category provides us with a split (co)fibration $\boldsymbol{THEO}_{\langle \boldsymbol{SIGN},\boldsymbol{clos}\rangle}\to\boldsymbol{SIGN}$ where $\boldsymbol{THEO}_{\langle \boldsymbol{SIGN},\boldsymbol{clos}\rangle}$, the flattened category, consists of pairs $\langle\Sigma,\Gamma\rangle$ where Σ is a signature and Γ is a closed set of sentences of $\boldsymbol{gram}(\Sigma)$, i.e. Γ is a theory of $\boldsymbol{clos}(\Sigma)$. A morphism $\sigma:\langle\Sigma,\Gamma\rangle\to\langle\Sigma',\Gamma'\rangle$ is a signature morphism $\sigma:\Sigma\to\Sigma'$ such that $c_{\Sigma'}(\Gamma(\sigma))\vdash\Gamma'$. This is the way theories have been originally defined in (π-)institutions. Notice that it is indifferent to flatten the indexed or the co-indexed category: the categories obtained are dual. This is because $c_{\Sigma'}(\Gamma(\sigma))\vdash\Gamma'$ holds iff $\Gamma\vdash\Gamma'(\sigma)$.

This construction can be generalised to presentations and strict presentations:

6.5.11 Definition – theories/presentations in a π-institution

Consider a π-institution <***SIGN***,***clos***>. We define the following categories:

1. $\boldsymbol{THEO}_{\langle SIGN,clos\rangle}$, the category of theories, has for objects the pairs $\langle\Sigma,\Phi\rangle$ where Σ is a signature and Φ is a closed set of sentences of $\boldsymbol{gram}(\Sigma)$. A morphism $\sigma:\langle\Sigma,\Phi\rangle\to\langle\Sigma',\Phi'\rangle$ is a signature morphism $\sigma:\Sigma\to\Sigma'$ such that $\sigma(\Phi)\subseteq\Phi'$.
2. $\boldsymbol{PRES}_{\langle SIGN,clos\rangle}$, the category of theory presentations, has for objects the pairs $\langle\Sigma,\Phi\rangle$ where Σ is a signature and $\Phi\subseteq\boldsymbol{gram}(\Sigma)$. A morphism $\sigma:\langle\Sigma,\Phi\rangle\to\langle\Sigma',\Phi'\rangle$ is a signature morphism $\sigma:\Sigma\to\Sigma'$ such that $\sigma(c_\Sigma(\Phi))\subseteq c_{\Sigma'}(\Phi')$.
3. $\boldsymbol{SPRES}_{\langle SIGN,clos\rangle}$, the category of strict theory presentations, has for objects the pairs $\langle\Sigma,\Phi\rangle$ where Σ is a signature and $\Phi\subseteq\boldsymbol{gram}(\Sigma)$. A morphism $\sigma:\langle\Sigma,\Phi\rangle\to\langle\Sigma',\Phi'\rangle$ is a signature morphism $\sigma:\Sigma\to\Sigma'$ such that $\sigma(\Phi)\subseteq\Phi'$.

These categories satisfy the properties that we have already proved for their linear temporal logic instantiations:

6.5.12 Proposition

Consider a π-institution <***SIGN***,***clos***>.

1. The categories $\boldsymbol{PRES}_{\langle SIGN,clos\rangle}$, $\boldsymbol{SPRES}_{\langle SIGN,clos\rangle}$ and $\boldsymbol{THEO}_{\langle SIGN,clos\rangle}$ define split fibrations and split cofibrations through the functor ***sign*** that projects their objects and morphisms to the corresponding signature components. Given a signature morphism $\sigma:\Sigma\to\Sigma'$,

	cartesian morphism for $\langle\Sigma',\Phi'\rangle$	**cocartesian morphism for** $\langle\Sigma,\Phi\rangle$
PRES	$\sigma:\langle\Sigma,\sigma^{-1}(c(\Phi))\rangle\to\langle\Sigma',\Phi'\rangle$	$\sigma:\langle\Sigma,\Phi\rangle\to\langle\Sigma',\sigma(\Phi)\rangle$
SPRES	$\sigma:\langle\Sigma,\sigma^{-1}(\Phi)\rangle\to\langle\Sigma',\Phi'\rangle$	$\sigma:\langle\Sigma,\Phi\rangle\to\langle\Sigma',\sigma(\Phi)\rangle$
THEO	$\sigma:\langle\Sigma,\sigma^{-1}(\Phi)\rangle\to\langle\Sigma',\Phi'\rangle$	$\sigma:\langle\Sigma,\Phi\rangle\to\langle\Sigma',c(\sigma(\Phi))\rangle$

2. All these (co)fibrations are fibre-(co)complete. The following procedure calculates limits and colimits of a diagram I with $I=\langle\Sigma_i,\Phi_i\rangle$ in these categories:
   - Calculate the limit $\sigma:\Sigma\to I$ or colimit $\sigma:I\to\Sigma$ of the underlying diagram of signatures.
   - Lift the result by computing the ***SIGN***-component according to the following rules:

	limit	colimit
PRES	$\Sigma:\langle\Sigma,\Sigma_{i\in I}\Sigma_i^{-1}(c(\Sigma_i))\rangle\in\Sigma$	$\Sigma:\Sigma\in\langle\Sigma,\Sigma_{i\in I}\Sigma_i(\Sigma_i)\rangle$
SPRES	$\Sigma:\langle\Sigma,\Sigma_{i\in I}\Sigma_i^{-1}(\Sigma_i)\rangle\in\Sigma$	$\Sigma:\Sigma\in\langle\Sigma,\Sigma_{i\in I}\Sigma_i(\Sigma_i)\rangle$
THEO	$\Sigma:\langle\Sigma,\Sigma_{i\in I}\Sigma_i^{-1}(\Sigma_i)\rangle\in\Sigma$	$\Sigma:\Sigma\in\langle\Sigma,c_\Sigma(\Sigma_{i\in I}\Sigma_i(\Sigma_i))\rangle$

3. If the category ***SIGN*** is (co)complete, so are ***PRES***$_{\langle SIGN,clos\rangle}$, ***SPRES***$_{\langle SIGN,clos\rangle}$ and ***THEO***$_{\langle SIGN,clos\rangle}$.

This symmetry between theories and models, as captured through indexed-categories, can be used to provide every π-institution with a canonical notion of model that makes it an institution:

6.5.13 Proposition – institution presented by a π-institution

Every π-institution $\langle SIGN,gram,\vdash\rangle$ defines the institution $\langle SIGN,gram,theo^{-1},\models\rangle$ where, for every signature Σ, $p\in gram(\Sigma)$ and $\Gamma\subseteq theo^{-1}(\Sigma)$, $\Gamma\models_\Sigma p$ iff $p\in\Gamma$.

proof

We just have to prove that the satisfaction condition holds. Let $\Sigma:SIGN$, $\models_\Sigma:mod(\Sigma)\subseteq gram(\Sigma)$ satisfies for every morphism $\sigma:\Sigma\to\Sigma'$, $p\in gram(\Sigma)$ and $\Gamma'\subseteq theo^{-1}(\Sigma')$, $theo^{-1}(\sigma)(\Gamma')\models_\Sigma p$ iff $\sigma^{-1}(\Gamma')\models_\Sigma p$ iff $p\in\sigma^{-1}(\Gamma')$ iff $\sigma(p)\in\Gamma'$ iff $\Gamma'\models_{\Sigma'}\sigma(p)$.

What this results says is that, whereas having a model-theory that corresponds to an intended domain of interpretation for the language of a logic may be a convenient way of defining a π-institution, once we are given a π-institution there is no much point in searching for a model-theory for it: its theories are up to the job at no conceptual expense.

Notice that the consequence relation induced by this notion of satisfaction is $\Gamma\vdash_\Sigma p$ iff, for every $\Gamma\subseteq theo^{-1}(\Sigma)$, $\Gamma\supseteq\Gamma$ implies $p\in\Gamma$, which is equivalent to $p\in c(\Gamma)$. That is to say, we recover the π-institution from which we started. Hence, institutions do give us a more concrete level of abstraction at which properties of specifications can be discussed.

Duality between models and theories can also be explored by lifting some of the signature-based constructions to theory-based ones:

6.5.14 Proposition

Let $\langle SIGN,gram,mod,\models\rangle$ be an institution.

1. The model functor $mod:SIGN^{op}\to CAT$ can be extended to $tmod:THEO^{op}\to CAT$ by assigning to every theory $\langle\Sigma,\Gamma\rangle$ the full subcategory of $mod(\Sigma)$ that consists of the models that satisfy Γ.
2. When the model functor is generated by a cofibration $sign: MODL\to SIGN^{op}$, we can lift it to $theo: MODL\to THEO^{op}$ by associating with every model the theory that consists of all the sentences that are true for that model.

proof

We do have to prove that a functor is, indeed, defined in 1. This is because, given a theory morphism $\varphi:\langle\Sigma,\Gamma\rangle \to \langle\Sigma',\Gamma'\rangle$ and a model M' of $\langle\Sigma',\Gamma'\rangle$, $M'|_\varphi$ satisfies Γ. The reader is invited to workout the proof of the second property.

6.5.15 Remark – theories over generalised models

We saw that functors $\mathbf{sign}: \mathbf{MODL} \to \mathbf{SIGN}^{op}$ give rise to model functors of the form $\_\Box\mathbf{sign}:\mathbf{SIGN}^{op} \to \mathbf{CAT}$ that are typical of Modal Logic. In an institution defined over such a model functor, the notions of theory and the constructions that we have presented around them for arbitrary (π-)institutions still apply. For instance, $\_\Box\mathbf{sign}:\mathbf{SIGN}^{op} \to \mathbf{CAT}$ still extends to $\mathbf{tmod}:\mathbf{THEO}^{op} \to \mathbf{CAT}$ as defined in 6.5.14. However, in this particular case, it is interesting to check if, or when, $\mathbf{sign}: \mathbf{MODL} \to \mathbf{SIGN}^{op}$ can lift to $\mathbf{theo}: \mathbf{MODL} \to \mathbf{THEO}^{op}$ so that $\mathbf{tmod}$ is, in fact, $\_\Box\mathbf{theo}$.

The intuition for this interest is that we should be able to associate with every model M a canonical specification $\mathbf{theo}(M)$ so that every morphism $T \to \mathbf{theo}(M)$ identifies T as a specification of M and, dually, M as a realisation of T. In this case, the models of T correspond to all possible refinements of T into "programs", something that we already discussed in 6.2.1. Identifying $\mathbf{MODL}$ with (the behaviours of) programs (or processes), this corresponds to the idea that programs can themselves be regarded as specifications, and that the models of a specification can be identified with the programs which, in some sense, "complete" the specification.

Although it is always possible to define a mapping that associates a theory $\mathbf{theo}(M)$ with every model M – the obvious candidate assigns to $\mathbf{theo}(M)$ the signature $\mathbf{sign}(M)$ and all the sentences that are true in it $\{p \in \mathbf{gram}(\mathbf{sign}(M)): \langle id,M\rangle \models p\}$ – it is not always possible to extend it to a functor that lifts $\mathbf{sign}$, i.e. such that $\mathbf{theo}(h:M \to M')=\mathbf{sign}(h)$. In order to motivate why this is so, assume that we do have a functor $\mathbf{theo}: \mathbf{MODL} \to \mathbf{THEO}^{op}$ and consider a morphism $h:M' \to M$. Because $\mathbf{theo}$ is a functor, we have $\mathbf{theo}(h):\mathbf{theo}(M) \to \mathbf{theo}(M')$. Hence, if M is a model of a theory T, i.e. if we have a morphism $\varphi:T \to \mathbf{theo}(M)$, then M' is also a model of T through the morphism $\varphi;\mathbf{theo}(h)$. In particular this implies that, for every signature Σ and $p \in \mathbf{gram}(\Sigma)$, if $\langle\Sigma,M\rangle \models_\Sigma p$ then $\langle\Sigma;\mathbf{sign}(h),M'\rangle \models_\Sigma p$. This property, however, is not universal: only some institutions (logics) satisfy it.

There are two important aspects about this property. The first is that, as proved in [33], it guarantees that $\mathbf{sign}: \mathbf{MODL} \to \mathbf{SIGN}^{op}$ lifts to $\mathbf{theo}: \mathbf{MODL} \to \mathbf{THEO}^{op}$ as defined, and that $\mathbf{tmod}$ is, in fact, $\_\Box\mathbf{theo}$. The second is that, once again, it is not an artificial "fabrication": it captures what, in Modal Logic, is known as "the p-property" [66]. That is to say, there are well known principles of Modal Logic that can be captured in

the categorical framework that we have defined, meaning that we are, indeed, addressing structural properties of Logic as we have known them. Once again, a more detailed discussion of the p-property and its impact in specification can be found in [32,33].

6.5.16 Exercise

Check that the institution of linear temporal logic as defined in 6.5.9 over general Kripke structures satisfies the p-property.

The reader will have noticed that, in an institution, the model functor is defined over ***CAT*** instead of ***SET***. This is because, some times, models come equipped with a non-trivial notion of morphism and, as we have argued in 6.5.9, it is not always possible to reflect in the logic the structure that they induce on models.

However, so far, the constructions that we have discussed do not depend on the notion of morphism between models and, hence, do not reflect the structure of the interpretation domain that has been chosen for the specification formalisms. We are now going to illustrate a situation in which such structures are of interest.

6.5.17 Definition – initial/terminal semantics

We say that an institution $<$***SIGN,gram,mod,*** $\models>$ has *initial* (resp. *terminal*) *semantics* provided that, for every theory $<\square,\square>$, the category ***tmod***$(<\square,\square>)$ has an initial (resp. terminal) object.

A non-trivial (but simple) example of an institution with terminal semantics is linear temporal logic.

6.5.18 Example – terminal semantics of linear temporal logic

The institution of linear temporal logic as defined in 6.5.8 over the cofibration ***spa(ftra)*** has terminal semantics. This is because, given any theory $<\square,\square>$, the category ***tmod***$(<\square,\square>)$ consists of sets of traces ordered by inclusion and is closed under intersection.

Notice that linear temporal logic interpreted over single sequences does not have terminal semantics: the (discrete) category that represents the set of all the sequences that satisfy a given theory does not have a sequence that represents the whole set unless the set is singular.

6.5.19 Exercise

Check that the institution of linear temporal logic as defined in 6.5.9 over general Kripke structures has terminal semantics. Does it coincide with the one obtained in 6.5.18?

Properties such as having an initial or terminal semantics can be used for differentiating between different institutions that present the same π-institution. It is interesting to note that, what we have called the "ca-

nonical" institution for a given π-institution, the one that uses theories as models, has both initial and terminal semantics: the initial model of every theory is the theory itself and the terminal model is the inconsistent theory, i.e. the full language. That is to say, we do not extract much information from the structure of models, which was only to be expected because we do not extract any information from the model-theory itself.

This example can also be used to transmit a certain "moral" message, namely that the pursuit for an institution with "good" properties like the ones above is, most of the times, a trivial one in the sense that it is usually possible to engineer a model-theory that satisfies one's requirements. As already said, the real purpose of these properties is to measure the relationship that the institution provides between specifications and the domain as abstracted through the models. Hence, one needs to determine the nature of the models relative to the domain of interpretation and not to the properties, otherwise what we get is "Abstract Nonsense",

7 Adjunctions

7.1 The social life of functors

Yes, even functors are entitled to have their own social lives... And they can be quite rich too. In this book, we will remain at the level of what is basic and indispensable for covering this last topic of our introduction to Category Theory: adjunctions. For that purpose, we will just provide a short introduction to what are normally accepted to be "the" morphisms between functors: natural transformations.

The best way of understanding what natural transformations consist of, and can be used for, is to look at functors as views that one has from one category into another and formulate the properties that characterise the "preservation" of such views.

7.1.1 Example – two views of Eiffel class specifications

We have already seen how one can look into Eiffel classes from the point of view of temporal logic specification: this is the view that is provided through the functor ***spec*** that we defined in 5.1.3. This functor accounts both for the pre/post conditions of methods and the class invariants. However, one is often interested in a more higher-level view of the behaviour of object classes that is concerned with the global properties that can be observed from their interfaces, typically their functions and routines. For this particular view, the actual specification of the functionality of the methods is of little relevance; they account for "how" these global properties are achieved rather than just "what" they are. Therefore, it makes sense that we define another functor ***obsv:*** ***CLASS_SPEC*** $\to$ ***PRES***$_{FOLTL}$ that offers the more abstract point of view. Formally, these observable properties can be defined as sentences that involve only routines and functions. Hence, given an Eiffel class specification $e=<\Sigma,P,I>$ we define

$$\textbf{\textit{obsv}}(e)=<\Sigma,\{\varphi \in LTL(fun(\Sigma) \cup rou(\Sigma)) \mid P \vdash_{\Sigma} \varphi\}>$$

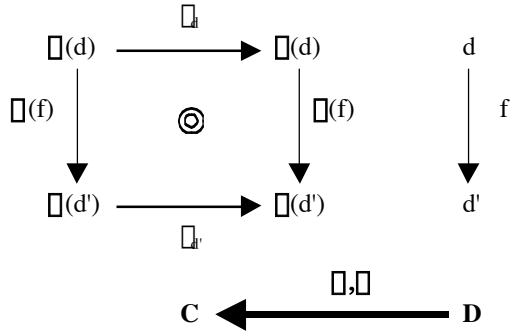
where Φ is the set of axioms of ***spec***(e). The reader is invited to check that this mapping defines a functor, i.e. that class inheritance induces an interpretation between the corresponding global properties.

Given two such views of class specifications, how can we relate them?

Clearly, such a relationship has to be established on the basis of morphisms that, for each class specification e, relate ***obsv***(e) and ***spec***(e). By definition, observable properties are derivable from the full class properties, so there is a morphism of presentations between each ***obsv***(e) and ***spec***(e). However, in order to respect the structure that morphisms (inheritance between class specification) induce on both views, the way observable properties relate through inheritance must be "the same" as the full specifications relate between them.

7.1.2 Definition – natural transformation

Given two functors Φ:***D***$\rightarrow$ ***C*** and Ψ:***D***$\rightarrow$ ***C***, a natural transformation η from Φ to Ψ, denoted by $\eta:\Phi \Rightarrow \Psi$ or $\eta:\Phi \Rightarrow \Psi$, is a function that assigns to each object d of ***D*** a morphism $\eta_d:\Phi(d)\rightarrow \Psi(d)$ of ***C*** such that, for every morphism $f:d\rightarrow d'$ of ***D***, the following square commutes, in which case we say that η_d is *natural in d* or that the *naturality condition* holds:



7.1.3 Exercise

Workout the example in full by defining and proving the properties of the natural transformation.

The most obvious example of a natural transformation is, naturally, the identity:

7.1.4 Definition – identity

Given a functor Φ:***D***$\rightarrow$ ***D*** the identity natural transformation id_Φ assigns to each object d of ***D*** the identity morphism $id_{\Phi(d)}$.

In the rest of this section, we are going to analyse some of the mechanisms and properties that are available for using natural transformations when reasoning about how functors relate. The main results that we need concern the way natural transformations compose and the mechanisms we have to act on them.

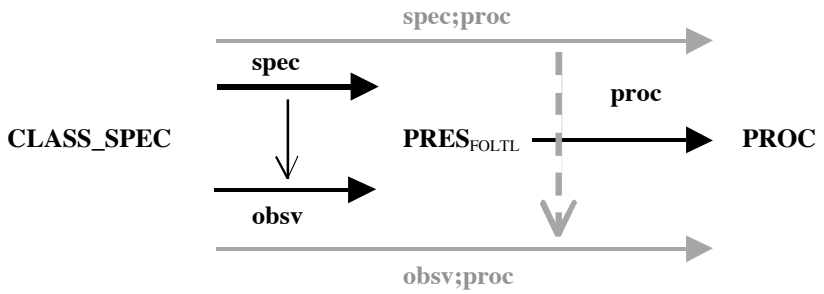
First, we can spell out what the dual of a natural transformation is:

7.1.5 Definition – dual of a natural transformation

Consider two functors $F:D \to C$ and $G:D \to C$, and a natural transformation $\alpha: F \Rightarrow G$. We define $\alpha^{op}: G^{op} \to F^{op}$ by $\alpha^{op}_d = \alpha_d$.

Notice that a morphism $\alpha^{op}(d): G^{op}(d) \to F^{op}(d)$ in C^{op} corresponds exactly to a morphism $\alpha(d): F(d) \to G(d)$ in C.

A useful class of operations on natural transformations is induced by functors into the sources, or from the targets, of the categories involved. For instance, supposing that we have a way (functor) to relate the domain of a viewpoint to another one (say between $PRES_{FOLTL}$ and $PROC$ as shown further down – 7.3.14), it makes sense to compose it with a natural transformation to provide a new one that extends the former to the second domain, e.g. to provide a mapping between the processes that capture the observable and the full view of object behaviour.



7.1.6 Definition – external composition

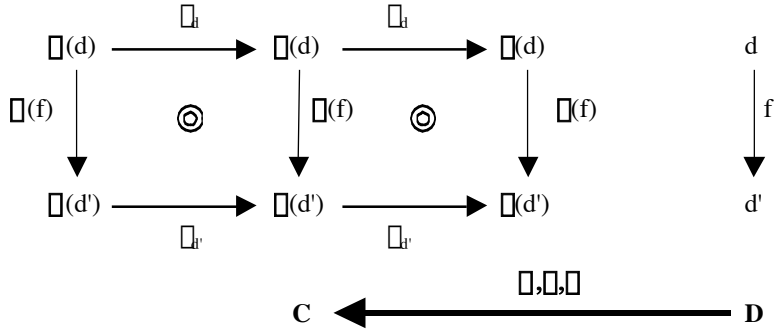
Consider two functors $F:D \to C$ and $G:D \to C$, and a natural transformation $\alpha: F \Rightarrow G$.

1. Given $H:E \to D$ we define $\alpha;H: F;H \to G;H$ by $(\alpha;H)_e = \alpha_{H(e)}$.
2. Given $K:C \to B$ we define $K;\alpha: K;F \to K;G$ by $(K;\alpha)_d = K(\alpha_d)$.

These are "external" operations on a given natural transformation. An "internal" law can also be defined that allow us to compose simpler views into more complex ones in the sense that they bridge over sequences of viewpoints:

7.1.7 Definition – internal composition

Consider three functors $F,G,H:D \to C$ and natural transformations $\alpha: F \Rightarrow G$ and $\beta: G \Rightarrow H$. The composition $\beta \circ \alpha: F \Rightarrow H$ is defined by $(\beta;\alpha)_d = \beta_d;\alpha_d$.



7.1.8 Exercise

Prove that the composition of natural transformations is well defined, is associative and that the identities are, indeed, units for the composition law.

7.1.9 Definition – natural isomorphism

A natural transformations $\alpha: F \Rightarrow G$ is said to be a *natural isomorphism* provided that each α_d is an isomorphism, in which case F and G are said to be *naturally isomorphic*, denoted by $F \cong G$.

7.1.10 Definition – equivalence of categories

Tho categories C and D are *equivalent* when they admit functors $F:D \to C$ and $G:C \to D$ such that $F;G \cong id_D$ and $G;F \cong id_C$.

Notice that equivalence between categories is a weaker notion than isomorphism (5.1.7). In particular, an equivalence does not operate up to "equality" but up to "isomorphism": for instance, given any object d of D, $F(G(d))$ does not need to be d but just isomorphic to d. Hence, to mark the fact, we do not use the term "inverse" for qualifiying each of these functors with respect to the other but "pseudo-inverse". This is the terminology used, for instance, in [12].

For instance, for any institution, ***PRES*** and ***THEO*** are usually not isomorphic (the same theory may admit many presentations), but they are equivalent (all the presentations that define the same theory are isomorphic). This example shows that a category may be equivalent to one of its strict subcategories.

7.1.11 Exercise

What about ***SPRES***?

Another example of an equivalence concerns two categories about which we have already highlighted many relationships:

7.1.12 Example – equivalence between *PAR* and $SET_{\square}$

The mappings

- $-_{\square}$ that removes the designated element from pointed sets and transforms morphisms into partial functions
- $+_{\square}$ that adds a new element to each set and completes partial functions by using the new element where they were undefined

define two functors whose compositions are naturally isomorphic to the identity.

Indeed, both categories are, basically, the "same" in the sense that one just makes explicit the partiality by presenting the designated element. In many applications to Computing, namely in the modelling of system behaviour as we have been illustrating with processes and specifications (theories), one tends to switch between one category and the other depending on whether we wish to attribute a meaning to the designated element, like for processes where it models steps performed by the environment, or be less "bureaucratic" (more pragmatic) and keep it just implicit. In fact, this is what we did in the graphical representations of the examples of universal constructions on processes: to simplify the notation, we omitted the designated element from the alphabets and represented the morphisms as partial functions. The "old" duality between "syntax" and "semantics" also tends to play a role: for semantic domains, like processes, it is useful to handle the designated element; but for syntax, like that of the parallel design language CommUnity that we study in chapter 8, it is often more "practical" to work instead with partial functions. The equivalence then tells us how we can switch between the two views.

Notice that the two categories are not isomorphic: the new element that is used to complete a set is not necessarily the one that was forgotten when that set is obtained from a pointed one. In fact, it is interesting for the reader to come up with a "real" definition of the functor that adds new elements to sets to make pointed sets: which elements does it add? Is there a "canonical" way of performing this completion?

The fact that we do not have an isomorphism is more significant for the "social life" of the categories themselves than for the structures that they endow internally on their objects (which is basically the same). We give an example of what we mean by this in 7.3.13.

7.2 Reflective functors

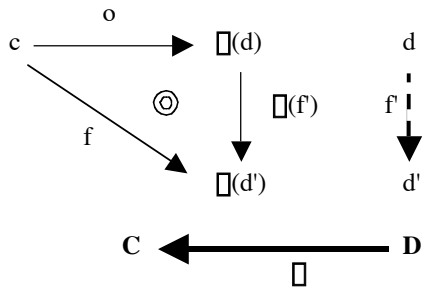
Keeping the promise of writing for the community of software scientists and practitioners, who are not necessarily as mathematically mature as most of the other books on Category Theory assume, we shall abstain from the traditional introduction to adjunctions, such as the construction of free monoïds and other mathematical structures, or Galois connections. Instead, and besides giving examples closer to system development, we will follow the method adopted in previous chapters, and use concepts introduced about subcategories to motivate the definition of "similar" properties of functors, based on the fact that the inclusion of a subcategory in another one defines a functor.

The generalisation from subcategories into functors that interests us for adjunctions concerns reflections and co-reflections. You may wish to revisit 3.3.8–3.3.12 where we motivated (co)reflections as defining a specialised class of "secretaries" through which all the interactions can be factorised. The generalisation consists, once again, in replacing the inclusion by a functor; in a way, our secretaries become "interpreters", i.e. some kind of "adjuncts" through which all the communication with the "other side of the functor" is handled.

7.2.1 Definition – reflection

Let $\Box: \boldsymbol{D} \Box \boldsymbol{C}$ be a functor.

1. Let c be a $\boldsymbol{C}$-object. A $\Box$-reflection for c is a $\boldsymbol{C}$-morphism $o: c \Box \Box(d)$ for some $\boldsymbol{D}$-object d such that, for any $\boldsymbol{C}$-morphism $f: c \Box \Box(d')$ where d' is a $\boldsymbol{D}$-object, there is a unique $\boldsymbol{D}$-morphism $f': d \Box d'$ such that $f = o; \Box(f')$ i.e. the $\boldsymbol{C}$-diagram commutes:



2. The functor $\Box$ is said to be *reflective* iff every $\boldsymbol{C}$-object admits a $\Box$-reflection. We tend to denote functors that are reflective with the special arrow $\bullet\!\!\rightarrow$.

That is to say, given an object c of $\mathbf{C}$, we are looking for the "best" object of $\mathbf{D}$ that can handle its relationships "across the border", i.e. with other objects of $\mathbf{D}$ through the functor F. The morphism o can be seen as the protocol that c needs to have with its "interpreter", or the "distance" that remains to be bridged in $\mathbf{D}$.

Notice that F-reflections are F-structured morphisms in the sense of 6.2.2. The following proposition provides a useful characterisation of reflections:

7.2.2 Proposition

1. Given a functor $F:\mathbf{D}\to\mathbf{C}$ and a $\mathbf{C}$-object c, the F-reflections for c are the initial objects of the category c/F.
2. A functor $F:\mathbf{D}\to\mathbf{C}$ is reflective iff, for every $\mathbf{C}$-object c, the category c/F has initial objects.

7.2.3 Exercise

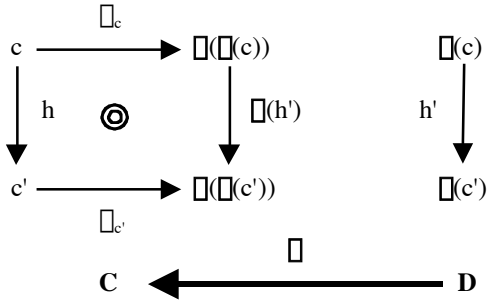
Prove 7.2.2 and conclude that F-reflections for an object c are essentially unique, i.e. two F-reflections for c are isomorphic and, if $f:c\to F(d)$ is a F-reflection for c and $h:d\to d'$ is an isomorphism, then $f;F(h)$ is also a F-reflection for c.

The notion of reflector for the inclusion functor defined by a reflective subcategory put forward in 5.1.13 can also be generalised to an arbitrary reflective functor.

7.2.4 Definition/proposition – reflector

Let $F:\mathbf{D}\to\mathbf{C}$ be a reflective functor. We define a functor $G:\mathbf{C}\to\mathbf{D}$ as follows:

- every $\mathbf{C}$-object c has a F-reflection arrow $\eta_c:c\to FG(d)$. We define $G(c)=d$
- consider now a morphism $h:c\to c'$. The composition $h;\eta_{c'}$ is such that the definition of F-reflection arrow for c guarantees the existence and uniqueness of a morphism $h':G(c)\to G(c')$ such that $h;\eta_{c'}=\eta_c;FG(h')$. We define $G(h)=h'$.



This functor is called a *reflector for* Φ.

proof

The proof is trivially generalised from the one given in 5.1.13 and is left as an exercise.

7.2.5 Definition/proposition – reflection unit

Let $\Phi: D \to C$ be a reflective functor and $\Psi: C \to D$ a reflector. The definition of Ψ provides directly a natural transformation $id_C \Rightarrow \Phi;\Psi$. We call it the *unit* of the reflection.

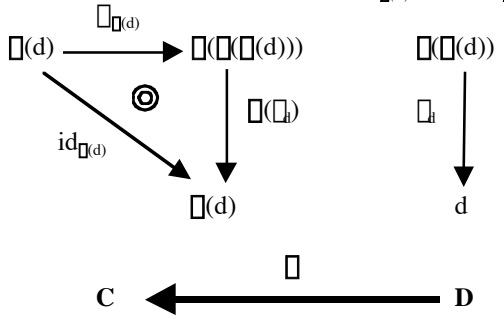
By duality, we obtain the notion of *co-reflective* functor and *co-reflector* for a co-reflective functor, generalising what was defined for co-reflective subcategories. We tend to denote functors that are co-reflective with the special arrow $\leftarrow\!\!\bullet$.

7.2.6 Proposition

Let $\Phi: D \to C$ be a reflective functor. Every reflector $\Psi: C \to D$ for Φ is co-reflective and admits Φ as a co-reflector. Moreover, given any D-object d, its Φ-co-reflection $\varepsilon_d: \Phi(\Psi(d)) \to d$ satisfies $\eta_{\Phi(d)};\Phi(\varepsilon_d) = id_{\Phi(d)}$

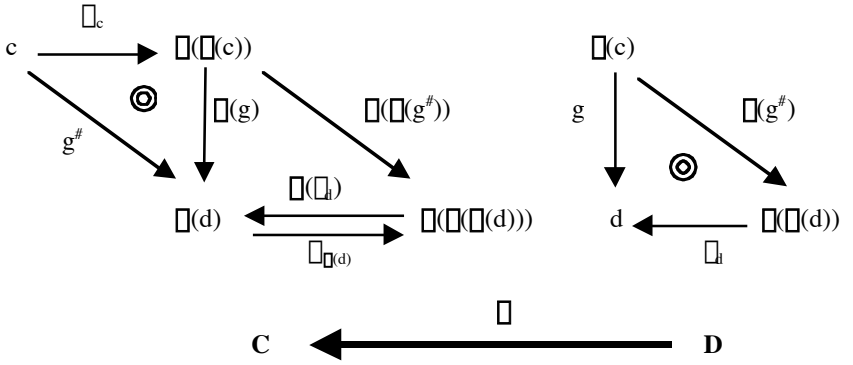
proof

Let d be a D-object. The universal properties of $\eta_{\Phi(d)}$ ensure the existence and uniqueness of a morphism $\varepsilon_d: \Phi(\Psi(d)) \to d$ such that $\eta_{\Phi(d)};\Phi(\varepsilon_d) = id_{\Phi(d)}$.



It is easy to see that ε_d is, indeed, a Φ-co-reflection for d. Let $g: \Phi(c) \to d$ be a D-morphism. We are going to prove that $g^{\#} = \eta_c;\Psi(g): c \to \Psi(d)$ satisfies $\Phi(g^{\#});\varepsilon_d = g$. Because there is only one morphism $h: \Psi(c) \to d$ such that $\eta_c;\Phi(h) = g^{\#}$, we are going to prove that $\Phi(g^{\#});\varepsilon_d$ satisfies that equation:

$\eta_c;\Phi(\Psi(g^{\#});\varepsilon_d)$
$= \eta_c;\Phi(\Psi(g^{\#}));\Phi(\varepsilon_d)$
$= g^{\#};\eta_{\Phi(d)};\Phi(\varepsilon_d)$ properties of natural transformations
$= g^{\#}$ properties of η_d



Moreover, $g^{\#}$ is the only morphism $g': c \to U(d)$ that satisfies $g=U(g');\varepsilon_d$ as the equality implies $\eta_c;U(g)=\eta_c;U(U(g'));U(\varepsilon_d)=g';\eta_{U(d)};U(\varepsilon_d)=g'$.

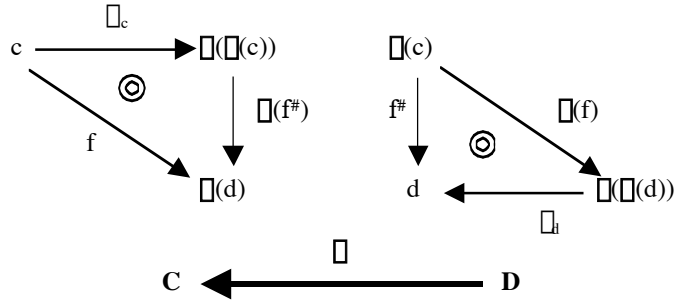
7.2.7 Corollary

Consider a reflective functor $U:D \to C$ and its reflector $F:C \to D$.

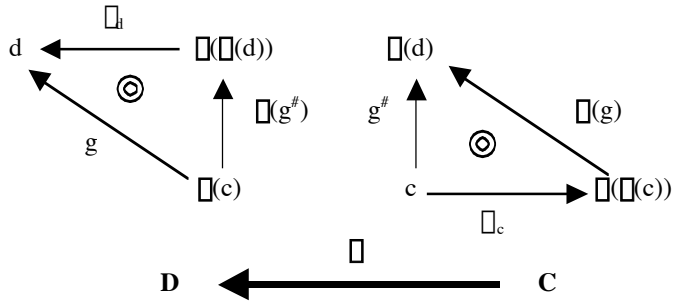
1. From proposition 7.2.6 we derive a natural transformation $\varepsilon;\varepsilon \Rightarrow id_D$ that we call the *co-unit* of the reflection. The two natural transformations (unit and co-unit) satisfy

$$\eta_{F(c)};F(\eta_c);F(\varepsilon_{F(c)}) = \eta_{F(c)};id_{F(c)} = id_{F(c)}$$
$$\eta_{U(d)};U(\varepsilon_d);\eta_{U(d)} = \eta_{U(d)};id_{U(d)} = \eta_{U(d)}$$

2. Every morphism $f:c \to U(d)$ can be mapped to $f^{\#}=F(f);\varepsilon_d:F(c) \to d$ which is the unique morphism $F(c) \to d$ that makes the ***C***-triangle commute.



and every morphism $g:F(c) \to d$ can be mapped to $g^{\#}=\eta_c;U(g):c \to U(d)$ which is the unique morphism $c \to U(d)$ that makes the ***D***-triangle commute.



These mappings define a bijection that is "natural" in ***C*** and ***D*** in the sense that it satisfies, for every $h:c' \to c$ and $k:d \to d'$,

$$F(h);f^{\#};k=(h;f;G(k))^{\#} \text{ and } h;g^{\#};G(k)=(F(h);g;k)^{\#}$$

proof

We leave it as an exercise. Notice that, for instance,

$$f^{\#}=\eta_c;G(f^{\#})=\eta_c;G(F(f));\varepsilon_{d'}=\eta_c;G(F(f));G(\varepsilon_{d'})=f;\varepsilon_{F(d)};G(\varepsilon_{d'})=f$$

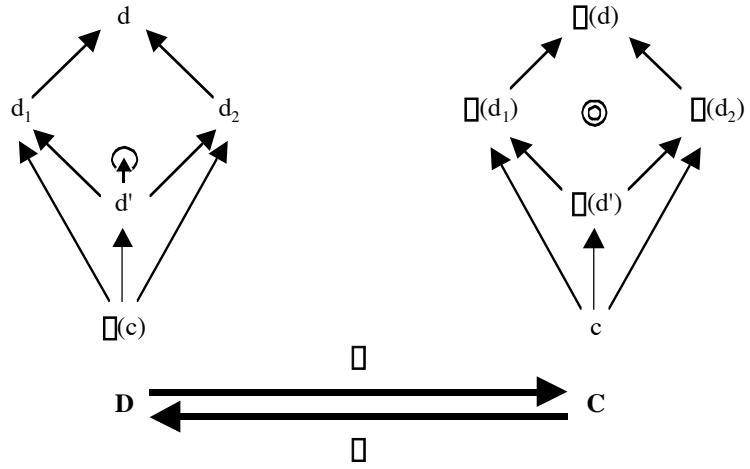
Some useful properties of reflective functors are:

7.2.8 Proposition

1. Reflective functors compose, i.e. if $F:E \to D$ and $G:D \to C$ are reflective then so is $F;G:E \to C$.
2. Reflective functors preserve limits.

proof

Left as an exercise. We just give a brief illustration of 2 for the case of pullbacks.



We start with a pullback diagram in ***D*** that we translate to ***C***. If we now consider a commutative cone $<c \to F(d_i)>$, we can lift it back to ***D*** through the reflec-

tion as a commutative cone $<\Gamma(c)\Box d_i>$. From the properties of the pullback, we are given a morphism $\Gamma(c)\Box d$ of commutative cones that translates, once again, to $\boldsymbol{C}$ as a morphism of commutative cones. Uniqueness can be easily checked.

The characterisation of reflective functors provided through 7.2.2 allows us to give examples among some of the constructions analysed in chapter 6:

7.2.9 Corollary

1. Let $\Box:\boldsymbol{C}\Box \boldsymbol{SET}$ be a functor. The functor $\Box: \boldsymbol{spa}(\Box)\Box \boldsymbol{C}$ that forgets the $\boldsymbol{SET}$-component of each object (6.3.3) is both reflective and co-reflective. The $\Box$-reflection of any $\boldsymbol{C}$-object c is $id_c:c\Box \Box(<c,\emptyset>)$ and its co-reflection is $id_c: \Box(<c,\Box(\mathbf{c})>)\Box c$.
2. Consider an indexed category $\Box:\boldsymbol{I}^{op}\Box \boldsymbol{CAT}$. The functor $\boldsymbol{fib}(\Box):\boldsymbol{FLAT}(\Box)\Box \boldsymbol{I}$ that projects objects and morphisms to their $\boldsymbol{I}$-components (6.4.7)is reflective iff, for every index i, $\Box(i)$ has an initial object $0_{\Box(i)}$, and is co-reflective iff, for every index i, $\Box(i)$ has a terminal object $1_{\Box(i)}$. The $\boldsymbol{fib}(\Box)$-reflection of any index i, when it exists, is $id_i:i\Box \boldsymbol{fib}(\Box)(<i,0_{\Box(i)}>)$, and its $\boldsymbol{fib}(\Box)$-co-reflection, when it exists, is $id_i:\boldsymbol{fib}(\Box)(<i,1_{\Box(i)}>)\Box i$.

Another obvious example of (co)reflective functors concerns, of course, (co)reflective subcategories:

7.2.10 Corollary

1. If $\boldsymbol{D}$ is a reflective subcategory of a category $\boldsymbol{C}$, then the inclusion functor is reflective.
2. If $\boldsymbol{D}$ is a co-reflective subcategory of a category $\boldsymbol{C}$, then the inclusion functor is co-reflective.

We can also generalise the results (3.3.10) that relate full (co)reflective subcategories with properties of the (co)unit:

7.2.11 Proposition

Consider a reflective functor $\Box:\boldsymbol{D}\Box \boldsymbol{C}$ and let $\Box$ be its co-unit.

1. $\Box$ is faithful iff, for every $\boldsymbol{D}$-object d, $\Box_d$ is epi.
2. $\Box$ is full and faithful iff, for every $\boldsymbol{D}$-object d, $\Box_d$ is an isomorphism.

7.2.12 Exercise

Complete the constructions and proofs of 7.2.6 and 7.2.7 to get acquainted with these newly acquired tools[10].

[10] And do have fun exploring the symmetries revealed through the diagrams, sketching the odd duality… The more mathematically mature readers may wish to identify Galois connections in the process.

7.3 Adjunctions

The reader already acquainted with Category Theory will have noticed that we are not only following a different path to the topic, as already justified even if it turns out not to be that different from [1], but also departing from the standard terminology (if one really exists) for adjunctions. The reason is that the terminology that we are introducing is a natural continuation of the one we used for subcategories (and this one is standard, or at least it complies with [79], which comes more or less to the same). What we have called a $\mathcal{F}$*-reflection for* d is called in [1] a $\mathcal{F}$*-universal arrow for* d (or with domain d), and a reflective functor is called therein an *adjoint*. The prefix *co* is used in [1] exactly in the same way so that a $\mathcal{F}$-co-reflection is a $\mathcal{F}$-co-universal arrow and a co-reflective functor is co-adjoint.

Although we prefer the terminolgy that we introduced in the previous sections, there are also good reasons for using the terminology introduced in [1]: adjoints and co-adjoints arise in *adjunctions*. In this section, we are going to introduce the standard terminology on adjunctions (i.e. [79]) because it is *really* standard. Adjunctions are an intrinsic part of the vocabulary of Category Theory. What is hardly standard is the way to approach and define the notion of adjunction. This is where, as authors, we can allow ourselves a little illusion of originality.

7.3.1 Definition – adjunction

An *adjunction* from a category $\boldsymbol{C}$ to another category $\boldsymbol{D}$ consists of

- two functors $\mathcal{F}:\boldsymbol{D}\to\boldsymbol{C}$ and $\mathcal{G}:\boldsymbol{C}\to\boldsymbol{D}$
- two natural transformations $\boldsymbol{id}_{\boldsymbol{C}}\Rightarrow\mathcal{F};\mathcal{G}$ and $\mathcal{G};\mathcal{F}\Rightarrow\boldsymbol{id}_{\boldsymbol{D}}$ satisfying

$$\mathcal{F}\Rightarrow\mathcal{F};\mathcal{G};\mathcal{F}\Rightarrow\mathcal{F}\quad=\quad\mathcal{F}\Rightarrow\mathcal{F}$$

$$\mathcal{G}\Rightarrow\mathcal{G};\mathcal{F};\mathcal{G}\Rightarrow\mathcal{G}\quad=\quad\mathcal{G}\Rightarrow\mathcal{G}$$

We use the notation $\mathcal{F}\dashv\mathcal{G}$ for such an adjunction. Given an adjunction $\mathcal{F}\overset{\eta}{\underset{\varepsilon}{\dashv}}\mathcal{G}$

- $\mathcal{G}$ can be called: the *right adjoint*, the *adjoint*, the *forgetful functor*
- $\mathcal{F}$ can be called: the *left adjoint*, the *co-adjoint*, the *free functor*
- η is called the *unit*
- ε is called the *co-unit*.

In what concerns the terminology, the left/right classification is quite widespread; (co)adjoints are used in [1] as already mentioned Classifying the functors as forgetful/free can be every helpful when the roles that they play is obvious. This is precisely the case of the adjunctions

that result from reflective subcategories, functor-structured categories and indexed categories as illustrated below: the (right) adjoint usually "forgets" part of the structure of objects that the left/co-adjoint is able to freely generate the additional structure.

An immediate example is:

7.3.2 Proposition

Every equivalence defines two adjunctions.

It is useful to state explicitly a result about duality:

7.3.3 Proposition

For every adjunction $F \dashv G$, $F^{op} \dashv G^{op}$ is also an adjunction.

And also about composition:

7.3.4 Proposition

Given functors $F:\mathbf{D} \to \mathbf{C}$, $G:\mathbf{C} \to \mathbf{D}$, $H:\mathbf{E} \to \mathbf{D}$, $K:\mathbf{D} \to \mathbf{E}$, if F is a left adjoint of G and H is a left adjoint of K then $F;H$ is a left adjoint of $K;G$.

There are many alternative ways of characterising adjunctions. It can even be hard to find two books that adopt the same characterisation as the defining one. However, they all involve, in some way or the other, but not arbitrarily, the properties analysed in 7.2.7. For instance,

7.3.5 Proposition

An adjunction from a category $\mathbf{C}$ to another category $\mathbf{D}$ can be obtained from

- two functors $F:\mathbf{D} \to \mathbf{C}$ and $G:\mathbf{C} \to \mathbf{D}$
- a bijection between morphisms $c \to G(d)$ and $F(c) \to d$ that is natural in $\mathbf{C}$ and $\mathbf{D}$.

proof

Much of the proof has been sketched in 7.2.7. We leave it as an exercise to complete it.

This characterisation is useful for the following example in particular:

7.3.6 Proposition

The power-set functor $\mathbf{2}^{-}:\mathbf{SET}^{op} \to \mathbf{SET}_{\bullet}$ that maps every set to its powerset as a pointed set, the empty set being the designated element, and every function to its inverse image, defines an adjunction from $\mathbf{SET}_{\bullet}$ to $\mathbf{SET}^{op}$. Its left adjoint computes power sets of proper elements (i.e. excluding the designated element) and inverse images.

proof

The reader is invited to carry out the proof as it is very instructive if not challenging due to the fact that it operates on a contravariant functor! The natural bijection is defined by associating functions of the form $f:A \to 2^B$ with $g:B \to 2^A$ by $a \mapsto g(b)$ iff $b \mapsto f(a)$.

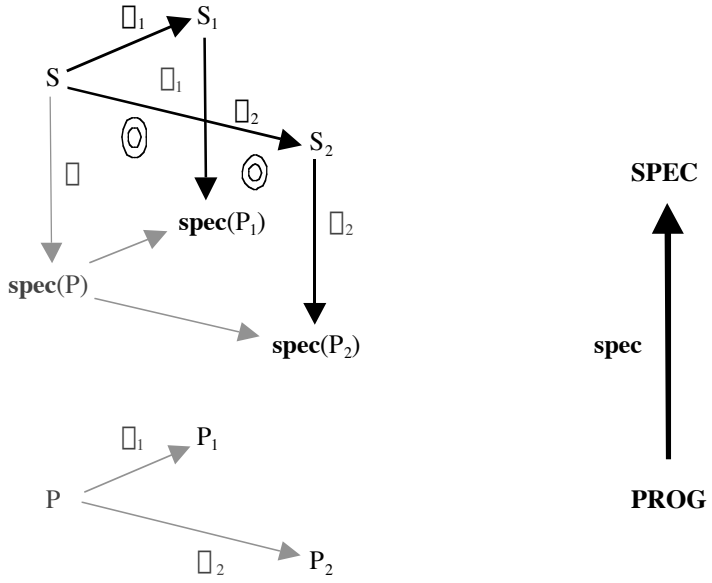
An adjunction is a very strong relationship between two categories: it allows us to give canonical approximations of objects in one domain with respect to the structural properties that are captured in the other domain. Examples of the use of adjunctions abound, even in the particular aspects of Computing Science that interest us in this book. One that we have worked out and presented in [36] concerns the synthesis of programs from specifications:

7.3.7 Example – synthesis of programs from specifications

We have already mentioned (5.1.3) how the satisfaction relationship between programs and specifications may be captured, in certain circumstances, by a functor ***spec***: $\boldsymbol{PROG} \to \boldsymbol{SPEC}$ that maps every program to the maximal set of properties that it satisfies. In such situations, we have called every morphism $S \to \boldsymbol{spec}(P)$ a possible realisation of the specification S by the program P (6.2.1). In this context, we have illustrated how systems can be evolved by interconnecting components that make new required properties to emerge (6.1.24). We have also showed how the process of assembling a system from smaller components, including the interconnection of new components, can be addressed in a compositional way by addressing realisations and not just individual specifications or programs (6.2.4).

We are now interested in incorporating into the picture the ability to synthesise programs from specifications as a means of supporting the process of compositional evolution that we have been addressing and according to which the addition of a component to a system should not require the recalculation (or the re-synthesis) of the whole system but only of the new component and its interconnections to the previous system. To illustrate our purpose, consider once again the vending machine (3.5.6). In (6.1.24), we developed the specification of a regulator and showed that, once interconnected with the vending machine, the new system did not allow arbitrary sales of cigars but required the insertion of a special token before a cigar can be selected. Assuming that the original vending machine was implemented and running, and that we were in possession of a realisation of the regulator, possibly by having used the synthesis method of [85], we would like to be able to synthesise the interconnections between the two programs (the running vending machine and the realisation of the regulator) in order to obtain a realisation of the specification diagram.

In summary, given realisations of component specifications (either obtained through traditional transformational methods, or synthesised directly from the specifications, or reused from previous developments), we would like to be able to synthesise the interconnections between the programs in such a way that the program diagram realises the specification diagram. That is to say, given specifications S_1 and S_2 (newly) interconnected via morphisms $\sigma_1:S\to S_1$ and $\sigma_2:S\to S_2$, and realisations $<\sigma_1,P_1>$, $<\sigma_2,P_2>$ of S_1 and S_2, respectively, one would like to be able to synthesise a realisation $<\sigma,P>$ of S and morphisms $\sigma_i:P\to P_1$ and $\sigma_2:P\to P_2$ such that $\sigma;\mathbf{spec}(\sigma_i)=\sigma_i;\sigma_i$ (i=1,2).

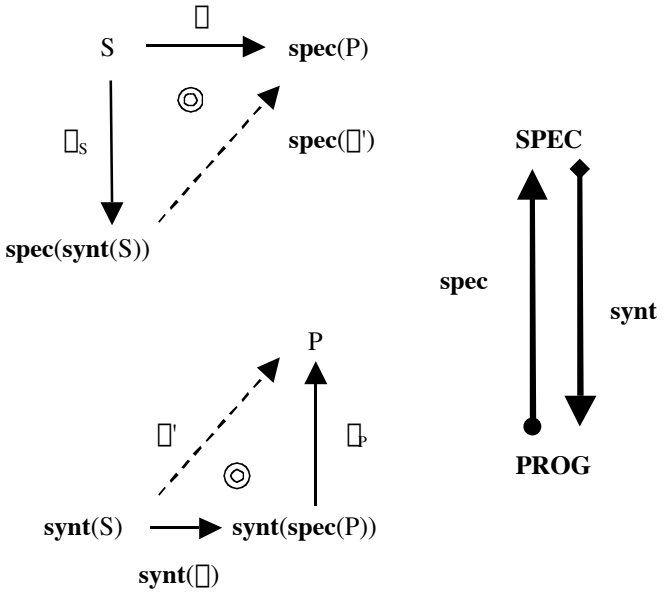


This general statement of what it means to synthesise interconnections makes it clear that it is both necessary to synthesise the middle program P and the morphisms σ_i that are required to interconnect the given programs. Because, in the general case, any object can be used in an interconnection, this suggests, rather obviously, that a functor ***synt***: ***SPEC*** $\to$ ***PROG*** is required that is somehow related to ***spec***. One possible such relationship is for ***spec*** to be the inverse (5.1.7) of ***synt***, but this is a rather strong property because it would require the two categories of programs and specifications to be isomorphic. Clearly, if this were to be the case, we could hardly claim that we were dealing with two different levels of abstraction. Hence, it makes sense to look for weaker properties of the relationship between the two functors.

It seems clear that, more than programs, synthesis must return realisations of the given specifications. That is to say, for every specification S, ***synt(S)*** must be provided together with a morphism

$\eta_S: S \to \mathbf{spec}(\mathbf{synt}(S))$ that establishes ***synt(S)*** as one of the possible realisations of *S*. Hence, η_S expresses a correctness criterion for ***synt***. Moreover, ***synt*** must respect interconnections in the following sense: given a specification diagram $\Delta: I \to \mathbf{SPEC}$, it is necessary that the program diagram $\Delta;\mathbf{synt}$ be a realisation of Δ through $(\eta_i: \Delta(i) \to \mathbf{spec}(\Delta(i)))_{i \in |I|}$ as defined in 6.2.4, i.e. we must have, for every $f: i \to j$ in ***I***, $\Delta(f);\eta_j = \eta_i;\mathbf{spec}(\Delta(f))$. But these are the ingredients that define a natural transformation. Hence, ***synt*** must be provided together with a natural transformation $\eta: \mathbf{1}_{\mathrm{SPEC}} \to \mathbf{synt};\mathbf{spec}$.

Consider now the synthesis of interconnections themselves. Given an interconnection of specifications $\sigma: S \to \mathbf{spec}(P)$ we should be able to synthesise $\sigma': \mathbf{synt}(S) \to P$ in such a way that the interconnection is respected, i.e. $\sigma = \eta_S;\mathbf{spec}(\sigma')$. This is equivalent to defining a (natural) bijection between the morphisms $S \to \mathbf{spec}(P)$ and the morphisms $\mathbf{synt}(S) \to P$. But this is, precisely, the property that characterises the existence of an adjunction between ***SPEC*** and ***PROG***. Hence, synthesis of interconnections can be characterised by the existence of a reflector (left adjoint) ***synt*** for ***spec***. Notice that 7.2.2 characterises the synthesis functor precisely in terms of the existence, for every specification, of a "minimal" realisation in the sense that all other programs that implement the specification simulate it.



Notice that the co-unit of the adjunction $\eta_P: \mathbf{synt}(\mathbf{spec}(P)) \to P$ is not necessarily an isomorphism because ***spec(P)*** may not be powerful enough to fully characterise *P* (we cannot guarantee that the specifica-

tion domain is expressive enough to capture the semantics of P in full). Hence, we are not even in the presence of an equivalence.

The direction of the co-unit reflects the fact that if we synthesise from the strongest specification of a program P, we obtain a program that cannot be stronger than P. Hence, the morphism ε_P provides a sort of "universal adaptor" between the program synthesised from ***spec(P)*** and P itself.

Although weaker than the existence of an inverse or a pseudo-inverse, the existence of a left adjoint to the functor ***spec***: ***PROG*** $\to$ ***SPEC*** is quite a strong property. This is not surprising because the ability to synthesise any specification is itself, in intuitive terms, a very strong property. In the literature, examples of synthesis of finite state automata from temporal logic specifications can be found, both from propositional linear temporal logic as above [85] and from branching time logic [30]. However, their generalisation to a full systems view is difficult. We shall see in section 7.5 that we can go a longer way in the context of formalisms that separate "Coordination" from "Computation".

Another important property that results from the properties of reflective functors is the following:

7.3.8 Proposition – adjunctions and reflections

1. Every reflective functor defines an adjunction (in which it plays the role of right adjoint).
2. In every adjunction $\Phi \dashv \Psi$, Ψ is reflective with reflector $\Phi\Psi$, and Φ is co-reflective with co-reflector $\Phi\Psi$.

This result allows us to derive from 7.2.9 two interesting adjunctions:

7.3.9 Corollary

1. Let Σ:***C*** $\to$ ***SET*** be a functor. The functor Σ***: spa***(Σ)$\to$ ***C*** has for left adjoint the functor that maps each ***C***-object c to $\langle c,\emptyset\rangle$, and for right adjoint the functor that maps each ***C***-object c to $\langle c,\Sigma\mathbf{(c)}\rangle$.
2. The functor ***alph*** that maps ***PROC*** to $\mathbf{SET}_{\Sigma}$ by forgetting behaviours has both a left and right adjoint. The left adjoint maps each alphabet A_Σ to the process $\langle A_\Sigma,\emptyset\rangle$ and the right adjoint to $\langle A_\Sigma,$***tra***$(A_\Sigma)\rangle$.

7.3.10 Corollary

In any π-institution, the functor ***sign***:***THEO*** $\to$ ***SIGN*** that maps theories to their underlying signatures has both a left and right adjoint. The left adjoint maps each signature to the theory $\langle\Sigma,c_\Sigma(\phi)\rangle$ and the right adjoint maps it to $\langle\Sigma,$***gram***$(\Sigma)\rangle$.

Basically, both results tell us how to map back and forth between processes and alphabets, and between theories and signatures.

7.3.11 Exercise

Workout direct proofs for both 7.3.9 and 7.3.10, and interpret the meaning of the natural transformations. Check how far 7.3.10 extends to presentations and strict presentations.

From 7.2.10 and 7.3.8 we get another obvious class of adjunctions:

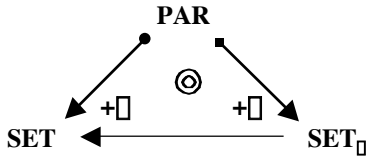
7.3.12 Corollary

1. Every reflective subcategory defines an adjunction in which the inclusion functor is the right adjoint.
2. Every co-reflective subcategory defines an adjunction in which the inclusion is the left adjoint.

7.3.13 Example – adjunctions between *SET*, *PAR* and $SET_{\square}$

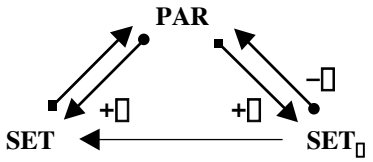
The fact that, as seen in 3.3.11, ***SET*** is a co-reflective subcategory of ***PAR*** tells us that the inclusion has a right-adjoint. As also seen in 3.3.11, this right adjoint (call it $+\square$) is the one that performs the traditional "elevation" of partial into total functions by extending sets with an "undefined" element, or "bottom", that serves as image for the elements in which the partial functions are undefined. This construction may well remind the reader of one of the functors over which the equivalence between ***PAR*** and $SET_{\square}$ was built in 7.1.12: the one that bears the same name. That is to say, we have the same kind of construction – the "elevation" – being performed over two different categories. Are they related?

There is a "natural" way in which every pointed set can be viewed as a (normal) set: just forget the "added structure", i.e. the fact that it has a designated element. Note that this does not mean "through away the designated element", which is what the functor $-\square$ (the pseudo-inverse of $+\square$ in the equivalence) does. Going back to 3.2.1, we are mapping pointed sets $<A,\square_A>$ to the underlying set A and morphisms between two pointed sets $f:<A,\square_A>\square<B,\square_B>$ to the corresponding total function $f:A\square B$. This mapping defines $SET_{\square}$ as a concrete category over ***SET***. The two elevations of ***PAR*** are related by this forgetful functor: the elevation to ***SET*** is simply the result of forgetting that there is a designated element in the elevation to $SET_{\square}$, which is captured by the following commutative diagram:

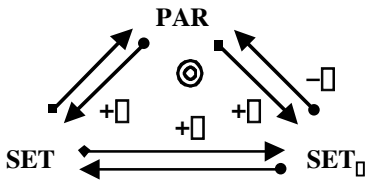


Notice that the elevation to $SET_{\square}$ is explicitly recorded into a structure that is added to sets whereas the elevation to ***SET*** is merely a repre-

sentation or encoding. This difference is well captured in the fact that it gives rise to an equivalence in the first case but "just" a reflective functor in the second. The elevation from ***PAR*** to $\boldsymbol{SET}_{\square}$ is also reflective but the fact that it is a co-reflection is more interesting: if we complete the diagram with the adjunctions that we have already built



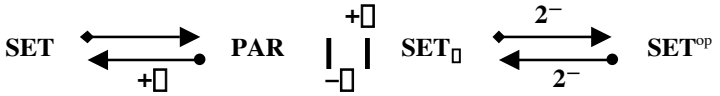
we can see that the functor that forgets the designated elements admits a right adjoint that, again, performs another kind of elevation, this time superposing a designated element to every set. This is just the elevation of partial functions being performed on total ones as a particular case.



Notice that we obtain a commutative diagram of adjunctions (i.e. of reflective and co-reflective functors), but not of the functors in general: for instance, the diagram of $+\square$ is, clearly, not a commutative one!

It is also important to point out, in the sequel of the remarks made in 7.1.12, that because ***PAR*** and $\boldsymbol{SET}_{\square}$ are equivalent, we tend to look at them as being "the same", but they may bear quite different relationships to other categories like, in this case, ***SET***: for instance, the "elevations" go in opposite directions, one from ***PAR***, the other into $\boldsymbol{SET}_{\square}$; one is reflective and the other co-reflective. What is more interesting is that these are "technical" differences: conceptually, both ***PAR*** and $\boldsymbol{SET}_{\square}$ provide a co-reflective representation for "normal" sets; the representations are different because, in spite of being equivalent, the two categories offer different structures and, hence, require different encodings of what is, essentially, the same kind of relationship.

This example also shows how diagrams of adjunctions can be useful to understand how different domains relate to each other; they provide a kind of "roadmap" or "classification scheme" that is essential for being able to "navigate" among the different structures that one tends to find in the literature. For instance, we can enrich the previous diagram of adjunctions with the one that we obtained in 7.3.6:



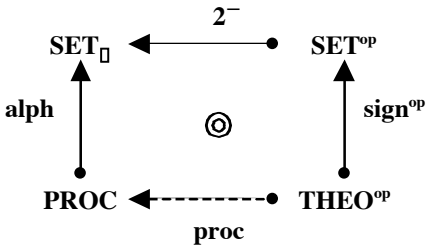
The composition of these adjunctions gives us the well known adjunction between ***SET*** and $\boldsymbol{SET}^{op}$ performed by the powerset functor.

This kind of roadmap was used in [99,113] for formalising relationships between models of concurrency like transition systems, synchronisation trees, event structures, etc, in what constitutes one of the most striking examples of the expressive power of adjunctions. Each such model is endowed with a notion of morphism that captures a form of simulation as a behaviour-preserving mapping. Typical operations of process calculi are captured as universal constructions as exemplified in 4.3.8 for ***PROC***. Reflections and co-reflections[11] are used for expressing the way one model is embedded in another: one of the functors in the adjunction embeds the more abstract model in the other, while the other functor abstracts away from some aspect of the representation.

Instead of reproducing an example of such uses of adjunctions, which on its own would hardly capture the richness of the classification that is developed for different kinds of concurrency models in [99,113], we present a related kind of application: a duality between process models and specifications. More precisely, a duality between ***PROC*** and linear temporal logic specifications given by $\boldsymbol{THEO}_{LTL}$ as presented in [32] to show how both semantics domains – theories and models – can be made part of the same roadmap.

7.3.14 Example – processes vs specifications

We start by recalling that ***PROC*** is concrete over $\boldsymbol{SET}_{\square}$ through the forgetful functor ***alph*** and that ***THEO*** is concrete over ***SET*** through the forgetful functor ***sign***. In fact, we proved in 7.3.9 and 7.3.10 that these functors are both reflective and co-reflective. Moreover, we have shown in 7.3.6 that the contravariant powerset functor $\boldsymbol{2^{-}}:\boldsymbol{SET}^{op}\square\boldsymbol{SET}_{\square}$ is reflective. We are now going to show that $\boldsymbol{2^{-}}$ can be lifted to $\boldsymbol{proc}:\boldsymbol{THEO}^{op}\square\boldsymbol{PROC}$ as a reflective functor that makes the diagram (of reflective functors) below commute.



[11] Further terminological confusion arises with respect to [113] where a co-reflection is an adjunction in which the reflective functor (the adjoint) is a full embedding, i.e. the straight generalisation of a full reflective subcategory.

Let $<\Sigma,\Phi>$ be a theory. The required commutativity of the diagram fixes the choice of the alphabet for ***proc***$(<\Sigma,\Phi>)$*:* the powerset 2^{Σ} considered as a pointed set. We are going to choose for its behaviour the set $\{\Box\Box(2^{\Sigma})^{\Sigma} \mid \Box \models \Box\}$, i.e. the least deterministic process that satisfies the properties given by the theory. It is not difficult to prove that we do obtain a functor.

If we consider now a process $<A_{\Sigma},\Phi>$, the category $<A_{\Sigma},\Phi>/$***proc*** consists of the specifications that are validated by the behaviours in Φ after a suitable translation (what in 6.5.9 we called generalised models). This category has an initial object: the set of all sentences in the language of 2^{A} that are validated by the behaviours of Φ translated by the unit of the powerset adjunction. It is the largest, not the smallest, because we are working with a contravariant functor. Hence, ***proc*** is indeed reflective, its reflector assigning to $<A_{\Sigma},\Phi>$ the theory $<2^{A},\{p \mid \Phi;\Box_{A} \models p\}>$.

Notice that because reflective functors preserve limits (7.2.8), colimits of specifications are mapped by ***proc*** to limits of the corresponding processes. This is again a form of *compositionality*. It says that composition of specifications as given by colimits of configuration diagrams captures parallel composition of the corresponding processes taking into account the interactions given by the diagram. In other words, compositionality expresses precisely the view that "conjunction as composition" [116] applies to development steps, not just specifications, or designs.

The reader is encouraged to consult [32,33] for a wider discussion of the relationships between these two categories, namely in the context of what are called categorical institutions in [87].

7.3.15 Exercises

1. Workout the full proof of 7.3.14.
2. Relate this result to 6.5.18.
3. Since, in the diagram of 7.3.14, ***sign***^{op} and $\boldsymbol{2}^{-}$ are reflective and ***alph*** is co-reflective, why didn't we take the composition of ***sign***^{op} with $\boldsymbol{2}^{-}$ and the co-reflector of ***alph*** to obtain an adjunction from ***PROC*** to ***THEO***^{op}?
4. What kind of generalisation into institutions can we hope for?

We would now like to extend this ability of adjunctions to relate different semantic models of concurrency, to different specification formalisms as captured by the categories of theories defined by institutions. For that purpose, we present in the next section a summary of the results published in [5].

7.4 Adjunctions in institutions

We start with an example in order to motivate the structures that are involved in mapping between specification formalisms as captured by institutions.

The typical temporal logics that have been used for the specification of reactive systems are based on linear time, of which the one we have been working with is only an example. However, sometimes, a branching time logic may be more justified. For instance, it is well known that verification techniques over branching time logic can be more effective. The expressive power of branching time logic can also be useful, especially in relation to progress properties related to required nondeterminism, like responsiveness. So, we would like to have ways of mapping between specifications in these different logics that support translations back and forth between them, through which one can take advantage of the best features of each.

As an example, consider the specification of a "user-friendly" vending machine that, once it accepts a coin, will make a cake and a cigar available. Notice that this is not a property of the vending machine as specified in 3.5.6; the specification given therein allows behaviours in which, for instance, after a coin is accepted, no cakes and no cigars are delivered! One could think that adding a property like *(coin* ***F****(cake cigar))* would solve the problem but it is easy to see that this requirement does not capture the availability of a cake or a cigar for the customer to choose; it requires that, in every behaviour, the acceptance of a coin is followed by the delivery of a cigar or a cake. Hence, it admits as an implementation a machine that only delivers cigars! Moreover, it forces the customer to take either the cake or the cigar among the options that are given, which does not make sense when that activity is not initiated by the machine. All this is because the trace-based semantics is not expressive enough to model choice; for that purpose, branching structures are required.

A logic in which such properties can be easily expressed is the branching time logic *CTL**. This logic is said to be branching because operators are provided that quantify over the possible future behaviours from the current state.

7.4.1 Definition – CTL* as an institution

The branching temporal logic institution *CTL** is defined as follows:

- Its category of signatures is ***SET***

- We define two classes of propositions (the state propositions Φ_s and the path propositions Φ_p) for a temporal signature Σ

$$\Phi_s ::= a \mid \neg\Phi_s \mid \Phi_s \wedge \Phi_s \mid A\Phi_p$$
$$\Phi_p ::= \Phi_s \mid \neg\Phi_p \mid \Phi_p \wedge \Phi_p \mid \Phi_p U \Phi_p$$

- The set of branching temporal propositions $gram_{CTL^*}(\Sigma)$ is the set of state propositions
- Every signature morphism $f:\Sigma \to \Sigma'$ induces the translation $gram(f):gram_{CTL^*}(\Sigma) \to gram_{CTL^*}(\Sigma')$ defined as follows:

$$gram(f)(\Phi_s) ::= f(a) \mid \neg gram(f)(\Phi_s) \mid gram(f)(\Phi_s) \wedge gram(f)(\Phi_s) \mid A\, gram(f)(\Phi_p)$$
$$gram(f)(\Phi_p) ::= gram(f)(\Phi_s) \mid \neg gram(f)(\Phi_p) \mid gram(f)(\Phi_p) \wedge gram(f)(\Phi_p) \mid gram(f)(\Phi_p) U gram(f)(\Phi_p)$$

- The model functor mod_{CTL^*} is defined as follows:
  - For every signature Σ, a branching Σ-model is a triple $<W,R,V>$ with R a total relation on W and $V:\Sigma \to 2^W$.
  - Given a branching model $M=<W,R,V>$ we denote by $L(M)$ the set of all infinite sequences $\sigma \in 2^\omega$ of the form $\sigma;V^{-1}$ where $\sigma:\omega \to W$ is such that $\sigma(i)R\sigma(i+1)$ for every $i \in \omega$. We denote by $L(M,s)$ the subset of $L(M)$ that is generated by the sequences $\sigma_s:\omega \to W$ such that $\sigma_s(0)=s$, i.e. $L(M,s)$ contains the subset of paths that begin at state s.
  - Let $M_1=<W_1,R_1,V_1>$ and $M_2=<W_2,R_2,V_2>$ be Σ-models. A (p-)morphism from M_1 to M_2 is a map $f:W_1 \to W_2$ such that:
    1. sR_1t implies $f(s)R_2f(t)$,
    2. $f(s)R_2u$ implies the existence of $t \in W_1$ such that sR_1t and $f(t)=u$,
    3. $s \in V_1(a)$ iff $f(s) \in V_2(a)$.
  - Let $f:\Sigma_1 \to \Sigma_2$ be a signature morphism. If $<W_2,R_2,V_2>$ is a Σ_2-model, then $<W_2,R_2,f;V_2>$ is a Σ_1-model called the *f-reduct* of $<W_2,R_2,V_2>$.
- The satisfaction relation is defined as follows: the truth of a Σ-proposition Φ in $M=<W,R,V>$ at state $s \in W$ (which we write $(M,s) \models_\Sigma \Phi$) is inductively defined as for LTL except for the operator A for which

$$(M,s) \models_\Sigma A\Phi \text{ iff, for every } \sigma_s \in L(M,s),\ \sigma_s \models^0_\Sigma \Phi.$$

- The branching temporal proposition Φ is said to be true in M, which we denote by $M \models_\Sigma \Phi$, iff $(M,s) \models_\Sigma \Phi$ at every state s *of* W.

Notice how the new operator, A, quantifies over all possible paths that start from the current state.

The reader is invited to check that the satisfaction condition holds, i.e. prove the following property:

7.4.2 Proposition – satisfaction condition

Let $f:\Sigma_1 \to \Sigma_2$ be a signature morphism. For every $M \in |\mathbf{mod}_{CTL^*}(\Sigma_2)|$ and $\varphi \in \mathbf{gram}_{CTL^*}(\Sigma_1)$, $M \models_{\Sigma_2} \mathbf{gram}(f)(\varphi)$ iff $\mathbf{mod}_{CTL^*}(f)(m) \models_{\Sigma_1} \varphi$.

As done in previous chapters, we will often use f instead of $\mathbf{gram}(f)$.

7.4.3 Corollary

*CTL** as defined in 7.4.1 is an institution.

As an example of a specification in CTL*, consider the user-friendly vending machine

```
specification  user-friendly vending machine is
signature      coin, cake, cigar
axioms         beg   A(¬cake∨¬cigar)
                   ∧ A(coin   (¬cake∨¬cigar)Wcoin)
               coin    A((¬coin)W(cake cigar))
               coin   (EXcake ∨EXcigar)
               (cake cigar)    A(¬cake∨¬cigar)Wcoin
               cake   (¬cigar)
```

The operator ***E*** is the dual of ***A***: it expresses the existence of a path from the current state in which the given property holds. Notice the use of the conjunction in *(coin (**EX***cake ∧ **EX***cigar))*; it requires the machine to give the customer the choice; hence, for instance, if the machine runs out of cakes, it may not accept coins even if cigars are still available.

It is clear from the definition of *CTL** that this logic "incorporates" *LTL* in the sense that it can express at least as much as *LTL*. A theory in *LTL* expresses properties about all possible behaviours of a system taken as infinite sequences of states. In *CTL**, this quantification can be made explicit through the operator ***A***. Hence, it should be straightforward to map a theory of *LTL* to a theory of *CTL** by qualifying every linear proposition with ***A***.

This syntactic transformation between the two languages respects the translations defined by signature morphisms, i.e. is "natural" on signatures. Indeed, it is captured by a natural transformation:

7.4.4 Definition/proposition

The family of functions $\alpha_\Sigma:\mathbf{gram}_{LTL}(\Sigma) \to \mathbf{gram}_{CTL^*}(\Sigma)$ defined by $\alpha_\Sigma(\varphi)=\mathbf{A}\varphi$ is a natural transformation from $\mathbf{gram}_{LTL}$ to $\mathbf{gram}_{CTL}$.

There is also a way in which this translation can be claimed to be "correct". Every branching structure gives rise to a linear one in a natural way:

7.4.5 Definition/proposition

Let $\Box_{\Box}$ map every branching model $M=<W,R,V>$ to the linear model $L(M)$ defined in 7.4.1. Given any branching $\Box$-models $M=<W,R,V>$ and $M'=<W',R',V'>$, and a p-morphism $f:M\Box M'$, let $\Box_{\Box}(f)$ be the inclusion $L(M)\Box L(M')$ that is induced by the properties of the morphism. The mapping $\Box_{\Box}$ is a functor $\boldsymbol{mod}_{CTL^*}(\Box)\Box \boldsymbol{mod}_{LTL}(\Box)$. The family $<\Box_{\Box}>_{\Box\Box|SIGN|}$ defines a natural transformation $\boldsymbol{mod}_{CTL^*}\Box \boldsymbol{mod}_{LTL}$.

That is to say, we generate from every branching structure M the linear structure $L(M)$ that consists of all possible paths (runs) through M.

The syntactic and semantic transformations agree in the following sense:

7.4.6 Definition/proposition

If $M=<W,R,V>$ is a branching $\Box$-model and $\Box\Box\boldsymbol{gram}_{LTL}(\Box)$, then $M\models_{\Box}\boldsymbol{A}\Box$ iff $\Box_{\Box}(M)\models_{\Box}\Box$.

This relationship between the syntax and semantics of the given institutions allows us to define the intended functor between the corresponding categories of theories:

7.4.7 Definition/proposition

The mapping $\mathcal{T}:\boldsymbol{THEO}_{LTL}\Box \boldsymbol{THEO}_{CTL^*}$ defined by $\mathcal{T}(<\Box,\Box>)=<\Box,c(\boldsymbol{A}\Box)>$ is a functor.

By $\boldsymbol{A}\Box$ we are denoting the set $\{\boldsymbol{A}\Box\Box\Box\Box\Box\}$ and by c the closure operator of CTL^*.

This functor allows us to translate any specification (theory) in LTL to a specification in CTL^*. This translation is "canonical" in the following sense:

7.4.8 Proposition

The functor $\mathcal{T}:\boldsymbol{THEO}_{LTL}\Box \boldsymbol{THEO}_{CTL^*}$ is co-reflective, that is, has a right adjoint.

It is not difficult to "guess" the nature of the right adjoint. Because $\mathcal{T}$ computes direct images through $\Box$, its right adjoint computes inverse images, i.e. the adjunction is given by a generalisation to closure operators of the well known Galois connection between direct and inverse images of sets. The unit of the adjunction is given by the inclusion $\Box\Box\Box_{\Box}^{-1}(c(\Box_{\Box}(\Box)))$.

7.4.9 Proposition

The mapping $\mathcal{U}:\boldsymbol{THEO}_{CTL^*}\Box \boldsymbol{THEO}_{LTL}$ given by $\mathcal{U}(<\Box,\Box>)=<\Box,\Box_{\Box}^{-1}(\Box)>$ is a co-reflector (right adjoint) of $\mathcal{T}$.

proof (of 7.4.7 and 7.4.8)

The functor $\mathcal{U}$ is a co-reflector of $\mathcal{T}$ iff for every theory $<\Sigma,\Gamma>$ of LTL, the pair $(<\Sigma,\Gamma_\Sigma^{-1}(c(\Gamma_\Sigma(\Gamma)))>,id_\Sigma)$ is a reflection. Consider $f:<\Sigma,\Gamma>\to<\Sigma',\Gamma'^{-1}(\Gamma')>$. We have to prove that there is a unique *CTL** morphism $f':<\Sigma,c(\Gamma_\Sigma(\Gamma))>\to<\Sigma',\Gamma'>$ such that $id_\Sigma;f'=f$. Unicity is automatically guaranteed by this equation. All that remains is the proof that f is a theory morphism $<\Sigma,c(\Gamma_\Sigma(\Gamma))>\to<\Sigma',\Gamma'>$ in *CTL**. Let $\Gamma_\Sigma(\Gamma)\vdash_\Sigma\varphi$. We have to prove that $f(\varphi)\in\Gamma'$. Because $\Gamma_\Sigma(\Gamma)\vdash_\Sigma\varphi$ we have $f(\Gamma_\Sigma(\Gamma))\vdash_{\Sigma'}f(\varphi)$ (a consequence of the satisfaction condition of *CTL**). But $f(\Gamma_\Sigma(\Gamma))=\Gamma_\Sigma(f(\Gamma))$ because Γ is a natural transformation. Hence, $f(\varphi)\in\Gamma_\Sigma c(f(\Gamma))$. On the other hand, $f(\varphi)\in\Gamma'^{-1}(\Gamma')$ because f was taken as a theory morphism $<\Sigma,\Gamma>\to<\Sigma',\Gamma'^{-1}(\Gamma')>$. Hence, $f(\varphi)\in\Gamma'$.

The co-reflector "forgets" the branching nature of time in *CTL** by retaining only those propositions φ for which $\mathbf{A}\varphi$ is a theorem in *CTL**, i.e. it retains those truths that hold for every possible path.

The existence of the adjunction means that, in order to prove that a *CTL**-theory *BT* provides an interpretation (refinement) of an *LTL*-theory *LT*, it is equivalent to prove $LT\subseteq\mathcal{U}(BT)$ or $\mathcal{T}(LT)\subseteq BT$. For practical purposes, the inclusion $\mathcal{T}(LT)\subseteq BT$ is easier to prove because it can be lifted to presentations. Indeed, if we take the category $\mathbf{PRES}_{LTL}$ of the theory presentations of *LTL*, the adjunction between presentations and theories (3.6.4) allows us to extend the adjunction between $\mathbf{THEO}_{CTL^*}$ and $\mathbf{THEO}_{LTL}$ to one between $\mathbf{THEO}_{CTL^*}$ and $\mathbf{PRES}_{LTL}$. Hence, the inclusion $\mathcal{T}(LT)\subseteq BT$ can be proved at the level of a presentation of *LT*. The converse, however, does not hold because, although there is also an adjunction between $\mathbf{THEO}_{CTL^*}$ and $\mathbf{PRES}_{CTL^*}$, the right adjoint does not go in the same direction as $\mathcal{U}$.

The actual relationship between $\mathbf{THEO}_{CTL^*}$ and $\mathbf{THEO}_{LTL}$ is stronger than what we proved. The proof above showed us that every *LTL*-theory $<\Sigma,\Gamma>$ is included in $<\Sigma,\Gamma_\Sigma^{-1}(c(\Gamma_\Sigma(\Gamma)))>$ but, in fact, they are equal. That is, when translated back from its image in *CTL**, an *LTL*-theory does not gain any theorems. This result can be proved by noticing that every linear structure can be generated by a branching one, i.e. the natural transformation Γ consists of surjective mappings, which gives us the "faithfulness" of the right-adjoint (7.2.11). We will generalise this result below but it is important to realise that this means that the translation from *LTL* to *CTL** is "conservative", i.e. the representation of *LTL* in *CTL** is faithful.

Notice that, in the proof above, no use was made of the syntactic transformation itself. Only the fact that Γ is a natural transformation was used, which indicates that the relationship between *LTL* and *CTL** can be generalised to other institutions.

In order to perform the generalisation, let us first analyse what in the example above can be cast directly in categorical terms. The basic ingredients in our example were:

- a natural transformation $\alpha: \mathbf{gram}_{LTL} \Rightarrow \mathbf{gram}_{CTL^*}$;
- a natural transformation $\beta: \mathbf{mod}_{CTL^*} \Rightarrow \mathbf{mod}_{LTL}$;
- the invariance condition $M \models_S a_S(f)$ iff $b_S(M) \models_S f$.

These are exactly the ingredients found in institution morphisms [61] and institution maps [87].

7.4.10 Definition – institution morphism

Let $\mathcal{I}=\langle \mathbf{SIGN}, \mathbf{gram}, \mathbf{mod}, \models \rangle$ and $\mathcal{I}'=\langle \mathbf{SIGN}', \mathbf{gram}', \mathbf{mod}', \models' \rangle$ be institutions. An institution morphism $\mu: \mathcal{I} \to \mathcal{I}'$ is a triple $\langle \alpha, \beta, \gamma \rangle$ where:

- $\alpha: \mathbf{SIGN} \to \mathbf{SIGN}'$ is a functor
- $\beta: \alpha; \mathbf{gram}' \Rightarrow \mathbf{gram}$ is a natural transformation
- $\gamma: \mathbf{mod} \Rightarrow \alpha; \mathbf{mod}'$ is a natural transformation

such that the following property (the invariance condition) holds for any signature $\Sigma \in |\mathbf{SIGN}|$, $m \in |\mathbf{mod}(\Sigma)|$ and $\varphi \in \mathbf{gram}'(\alpha(\Sigma))$: $m \models_\Sigma \beta_\Sigma(\varphi)$ iff $\gamma_\Sigma(m) \models'_{\alpha(\Sigma)} \varphi$.

7.4.11 Definition – institution map

Let $\mathcal{I}=\langle \mathbf{SIGN}, \mathbf{gram}, \mathbf{mod}, \models \rangle$ and $\mathcal{I}'=\langle \mathbf{SIGN}', \mathbf{gram}', \mathbf{mod}', \models' \rangle$ be institutions. An institution map $\mu: \mathcal{I} \to \mathcal{I}'$ is a triple $\langle \alpha, \beta, \gamma \rangle$ where:

- $\alpha: \mathbf{SIGN} \to \mathbf{SIGN}'$ is a functor
- $\beta: \mathbf{gram} \Rightarrow \alpha; \mathbf{gram}'$ is a natural transformation
- $\gamma: \alpha; \mathbf{mod}' \Rightarrow \mathbf{mod}$ is a natural transformation

such that the following property (the invariance condition) holds for any signature $\Sigma \in |\mathbf{SIGN}|$, $m' \in |\mathbf{mod}(\alpha(\Sigma))|$ and $\varphi \in \mathbf{gram}(\Sigma)$: $\gamma_\Sigma(m') \models_\Sigma \varphi$ iff $m \models'_{\alpha(\Sigma)} \beta_\Sigma(\varphi)$.

7.4.12 Proposition

Through 7.4.4, 7.4.5 and 7.4.6 we have defined both a map $LTL \to CTL^*$ and a morphism $CTL^* \to LTL$.

Indeed, the fact that the relationship between the two institutions is based on the identity functor between their categories of signatures blurs the difference between both concepts (morphism and map).

The existence of the two functors $\mathcal{T}$ and $\mathcal{V}$ between $\mathbf{THEO}_{LTL}$ and $\mathbf{THEO}_{CTL^*}$ is also a consequence of the existence of a map and a morphism between the two institutions:

7.4.13 Proposition

Let $\mu=\langle \alpha, \beta, \gamma \rangle: \mathcal{I} \to \mathcal{I}'$ be an institution map. The functor α can be extended to a functor $\mathbf{THEO}_\mathcal{I} \to \mathbf{THEO}_{\mathcal{I}'}$ by establishing $\alpha(\langle \Sigma, \Phi \rangle) = \langle \alpha(\Sigma), c(\beta_\Sigma(\Phi)) \rangle$.

7.4.14 Proposition

Let $\Phi=<\Phi,\Phi',\Phi>: \Phi \to \Phi'$ be an institution morphism. The functor Φ can be extended to a functor $\boldsymbol{THEO}_{\Phi}: \boldsymbol{THEO}_{\Phi} \to \boldsymbol{THEO}_{\Phi'}$ by establishing $\Phi(<\Sigma,\Gamma>)=<\Phi(\Sigma),\Phi'^{-1}_{\Phi}(\Gamma)>$.

We can now generalise the results on the adjunction between the categories of theories of two institutions:

7.4.15 Proposition

Let $\Phi=<\boldsymbol{SIGN},\boldsymbol{gram},\boldsymbol{mod},\models>$ and $\Phi'=<\boldsymbol{SIGN}',\boldsymbol{gram}',\boldsymbol{mod}',\models'>$ be institutions, $\Phi=<\Phi,\Phi,\Phi>: \Phi \to \Phi$ an institution map and $<\Phi,\Phi',\Phi>:\Phi \to \Phi$ a morphism such that Φ is a right adjoint of Φ, and, for every $\Phi \in |\boldsymbol{SIGN}|$, $\Phi_{\Phi}=\boldsymbol{gram}(\Phi_{\Phi});\Phi'_{\Phi(\Phi)}$ where Φ is the unit of the adjunction. Then,

1. The functor $\mathcal{V}:\boldsymbol{THEO}_{\Phi} \to \boldsymbol{THEO}_{\Phi}$ induced by the morphism $<\Phi,\Phi',\Phi>$, is a right adjoint of the functor $\boldsymbol{THEO}_{\Phi} \to \boldsymbol{THEO}_{\Phi'}$ induced by the map $<\Phi,\Phi,\Phi>$.
2. If each component of Φ is surjective, i.e. if the institution morphism is sound in the sense of [61], then the units Φ_{Φ}, as theory morphisms, are conservative.

proof:

this is a direct generalisation of the proof of 7.4.9.

That is to say, adjunctions on signatures can be lifted to adjunctions of theories provided that the left adjoint is associated with a map and the right adjoint with a morphism of institutions. A compatibility result is required, $\Phi_{\Phi}=\boldsymbol{gram}(\Phi_{\Phi});\Phi'_{\Phi(\Phi)}$, to make sure that both the map and the morphism make, essentially, the same translations. Notice that the invariance condition relating Φ and Φ, automatically generates a similar property for Φ. The result on "conservative" representations of one formalism into another is also important: basically, it says that no new theorems arise when a theory is translated from one formalism to another.

This result shows that there is a very strong relationship between institution morphisms and maps, as suggested by the fact that they make use of essentially the same transformations between languages and models. The difference between them, which is evident in the directions taken by the transformations vis-à-vis the functor between the categories of signatures, can be explained more easily when we see that they correspond to the two directions of an adjunction. Notice that the map takes the direction of the left adjoint while the morphism takes the direction of the right adjoint. These directions are very much consistent with the accepted view of maps as providing representations and morphisms projections of one institution into another.

In fact, the result above is more general in that the existence of a map (resp. morphism) and a right (resp. left) adjoint for the signature func-

tor guarantees the existence of a morphism (resp. map) in the other direction that generates a right (resp. left) adjoint to the theory functor:

7.4.16 Proposition

Let $\mathcal{I}$=<***SIGN***,***gram***,***mod***, $\models$> and $\mathcal{I}'$=<***SIGN***',***gram***',***mod***', $\models'$> be institutions,

1. if σ=<Σ,Σ',σ>: $\mathcal{I}$ → $\mathcal{I}'$ is a map such that the functor σ has a right adjoint ρ, then
   a) the triple <Σ,η',ε'> where η' is the natural transformation defined by η'_{Σ}=$\Sigma_{\sigma(\Sigma)}$;***gram***'(η_{Σ}) and ε' is the natural transformation defined by $\varepsilon'_{\Sigma'}$=***mod***'($\varepsilon_{\Sigma'}$);$\Sigma_{\sigma(\Sigma')}$,is an institution morphism $\mathcal{I}$ → $\mathcal{I}'$.
   b) the functor $\mathcal{T}$: ***THEO***$_{\mathcal{I}}$ → ***THEO***$_{\mathcal{I}'}$ induced by the map has a right adjoint – the functor $\mathcal{U}$:***THEO***$_{I'}$ → ***THEO***$_{I}$ induced by the morphism, i.e. $\mathcal{U}$(<Σ',Φ'>)=<Σ(Σ'),$\Sigma_{\sigma(\Sigma')}^{-1}$($\rho_{\Sigma'}^{-1}$($\Phi'$))>.
2. if <Σ,η',ε'>:$\mathcal{I}$ → $\mathcal{I}'$ is a morphism such that the functor σ has a left adjoint ρ, then
   a) the triple <Σ,η,ε> where η is the natural transformation defined by η_{Σ}=***gram***(η_{Σ});$\eta'_{\sigma(\Sigma)}$ and ε is the natural transformation defined by ε_{Σ}=$\Sigma_{\sigma(\Sigma)}$;***mod***(ε_{Σ}) is an institution map from $\mathcal{I}'$ → $\mathcal{I}$.
   b) the functor $\mathcal{U}$: ***THEO***$_{\mathcal{I}'}$ → ***THEO***$_{\mathcal{I}}$ induced by the morphism has a left adjoint – the functor $\mathcal{T}$:***THEO***$_{\mathcal{I}}$ → ***THEO***$_{\mathcal{I}'}$ induced by the map, $\mathcal{T}$(<Σ,Φ>)=<σ(Σ),$\eta'_{\sigma(\Sigma)}$c(ε_{σ}(Φ))>.

That is to say, provided that there is an adjunction between the categories of signatures of two institutions, maps and morphisms between them can be defined, interchangeably, that provide adjunctions for the functor between the corresponding categories of theories.

7.5 Coordinated categories

In this last section of the last chapter of Part Two and, hence, what could have been the closing paragraphs of an Introduction to Category Theory for Software Scientists and Practitioners, we address one of the topics that have been closest to the hearts of the research team that has been working on CommUnity, i.e. the subject of part three: the formalisation of the separation of concerns that is known as "Coordination". This is both a justification for stopping here – the reader will not need any more categorical "ammunition" to attack part three – and having gone this far – the kind of application discussed in part three is intrinsically related to this topic and, even if a quicker route could have been taken, everybody knows that motorways are not the best ways for getting to know a region.

An introduction to this subject has already been given in section 5.2 as part of the motivation for studying the behaviour of functors in relation to universal constructions; the reader is invited to read it (once again) as well as, if possible, what I consider to be the best introduction to "Coordination": Arbab's gem "What Do You Mean, Coordination?" [3]. The central idea of this research area is to investigate the extent up to which a given formalism can separate between the mechanisms that coordinate the interactions that are responsible for emergent behaviour from the description of what in systems is responsible for the computations that ensure the functionalities of the services that individual system components provide.

For instance, object-oriented systems do not go a long way in supporting that separation. Because interactions in object-oriented approaches are based on *identities* [72], in the sense that, through clientship, objects interact by invoking specific methods of specific objects (instances) to get something specific done, the resulting systems are too rigid to support the levels of agility required by the "just-in-time" binding mechanisms of (web) services; any change on the collaborations that an object maintains with other objects needs to be performed at the level of the code that implements that object and, possibly, of the objects with which the new collaborations are established. That is to say, as beautifully put in [100], feature calling is, for interconnections, what assembly language represents for computations. On the contrary, interactions in a service-oriented approach should be based only on the description of what is required, thus decoupling the "what one wants to be done" from the "who does it". In the context of the "societal metaphor" that we have been using in the book, it is interesting to note that this shift from "object" to "service"-oriented interactions mirrors what has been happening already in human society: more and more, business relationships are being established in terms of acquisition of services (e.g. 1000 Watts of lighting for your office) instead of products (10 lamps of 100 Watts each for the office).

Our introduction to section 5.2 has disclosed most of the "secrets" of the mathematical characterisation that we started to develop in [35] as a systematic study of the nature and properties of the separation between "Computation" and "Coordination" concerns. Now that the reader has more categorical background, we can revisit the motivation that has been already delivered. Notice that we shall systematically work with co-limits just to fix a direction of the "component-of" relationship and use it consistently. However, those that are more accustomed to limits can simply switch the direction of the arrow, i.e. work in the opposite category. Summarising:

- We model this separation by a forgetful functor $\boldsymbol{int}:\boldsymbol{SYS}\to\boldsymbol{INT}$ where the category $\boldsymbol{SYS}$ stands for the representations (models, behaviours,

specifications, programs, …) of the components out of which systems can be put together, and the category ***INT*** captures the "interfaces" through which interconnections between system components can be established;

- The functor ***int*** should be faithful (5.1.7) so that morphisms in ***SYS*** (the "component-of" relationship) do not induce more relationships between components than those that can be captured through their underlying interfaces. That is to say, by taking into consideration the computational part, we should not get additional observational power over the external behaviour of systems. Using the terminology that we introduced in the previous chapter, ***SYS*** is concrete over ***INT***.
- Because we use diagrams for modelling configurations of complex systems and colimits to obtain emergent behaviour, ***int*** should lift colimits (5.2.1): when we interconnect system components in a (configuration) diagram, any colimit of the underlying diagram of interfaces establishes an interface for which a computational part exists that captures the joint behaviour of the interconnected components as given by the colimit of the original diagram. We have already mentioned that this property expresses (non)-interference between computation and coordination: on the one hand, the computations assigned to the components cannot interfere with the viability (in the sense of the existence of a colimit) of the underlying configuration of interfaces; on the other hand, the computations assigned to the components cannot interfere in the calculation of the interface of the resulting system. For instance, we saw in 6.1.22 that split fibre-(co)complete (co)fibrations lift limits.
- It is also clear that ***int*** should preserve colimits (5.2.1): every interconnection of system components should be an interconnection of the underlying interfaces, i.e. computations should not make a configuration of system components "viable", in the sense that it admits a colimit, when the underlying configuration of interfaces is not. This is another form of the required "non-interference". Given that ***int*** is faithful, this means that all colimits in ***SYS*** are concrete (6.1.9).

Lifting and preservation of colimits imply that any colimit in ***SYS*** can be computed by first translating the diagram to ***INT***, then computing the colimit in ***INT***, and finally lifting the result back to ***SYS***, a situation that we have already encountered for ***PROC*** through the functor ***alph*** and for the category of theories (or presentations) ***THEO*** of any (π-)institutions through the functor ***sign***. In the case of processes, this means that the set of behaviours does not interfere with the interconnections; and in the case of theories, that interconnections are established just by name bindings.

Both examples allow us to illustrate another intuitive property of the separation that is not captured by those mentioned so far. Consider, for instance, processes. Taking pullbacks as the most basic form of interconnection, we can notice that the "middle" process through which we express the interconnection is "always" idle, i.e. has all possible behaviours. Indeed, the set of behaviours that is present in the middle process does not interfere neither in the interconnection, which is expressed at the level of the alphabets, nor in the calculation of the set of behaviours of the resulting process, which is defined through the intersection of the inverse images of the sets of behaviours of the other two component processes. Hence, there is a sort of "canonical" middle processes: those that are idle. Notice that their duals, the empty processes, do not make good middle processes because they do not admit any incoming morphisms…

The same happens with theories and theory presentations: the middle object in a pushout is "always" empty (or the closure of the empty set of axioms) because the theorems that result from the pushout are computed from the pushout of the signatures and the theorems of the other two components. This seems to be saying that the middle objects that we use for interconnecting components, be it for pullbacks or pushouts, are, essentially, interfaces, which makes all the sense from the point of view of the separation of "Coordination" from "Computation". How can we express this property in categorical terms?

Basically, and taking the colimit approach as exemplified by, for instance, theories, what we want is to be able to assign to every interface C:***INT*** a component $s(C)$:***SYS*** such that, for every morphism $f:C \to \mathbf{int}(S)$, there is a morphism $g:s(C) \to S$ such that $\mathbf{int}(g)=f$. That is to say, we want every interface C to have a "realisation" as a system component $s(C)$ in the sense that, using C to interconnect a component S, which is achieved through a morphism $f:C \to \mathbf{int}(S)$, is tantamount to using $s(C)$ through any $g:s(C) \to S$ such that $\mathbf{int}(g)=f$. Notice that, because ***int*** is faithful, there is only one such g, which means that f and g are, essentially, the same. That is, sources of morphisms in diagrams in ***SYS*** are, essentially, interfaces. We would use the dual property to characterise what happens with processes.

Such a realisation is called a *discrete lift* in [1], and a functor ***int*** for which every object C:***INT*** admits a discrete lift is said to have *discrete structures*.

7.5.1 Definition – discrete lifts/structures

Given a concrete category U:$\mathbf{D} \to \mathbf{C}$, a *discrete lift* for c:$\mathbf{C}$ is a $\mathbf{D}$-object d such that $U(d)=c$ and, for every morphism $f:c \to U(d')$, there is a morphism $g:d \to d'$ such that $U(g)=f$. The functor (concrete category) is

said to have *discrete structures* whenever every ***C***-objects admits a discrete lift.

The dual notion is called *indiscrete lift* and the functor (concrete category) is said to have *indiscrete structures*.

When ***int*** lifts and preserves colimits, this property allows us to replace every "middle" object in a configuration diagram by the discrete lift of the underlying interface: both diagrams will have the same colimits. For all "practical" purposes, this means that we can use more economical representations for configuration diagrams by showing only the interfaces of the middle objects that interconnect components.

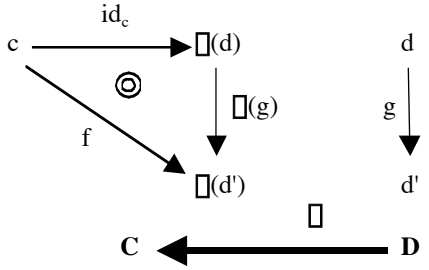
It is easy to see that the indiscrete lift of a process alphabet A_Σ is $\langle A_\Sigma, \mathbf{tra}(A_\Sigma)\rangle$ and the discrete lift of a signature Σ is the theory $\langle \Sigma, c_\Sigma(\phi)\rangle$. They also admit their dual versions – i.e. signatures have indiscrete lifts (inconsistent theories) and alphabets have discrete lifts (empty processes) – but these are not the ones that interest us for system configuration: they disable rather that enable interaction!

The more attentive reader is probably having a feeling of *déjà vu*… Indeed, these (in)discrete lifts are the objects involved in the (co)reflections that define the corresponding forgetful functors as (co)reflectors (see 7.3.9 and 7.3.10):

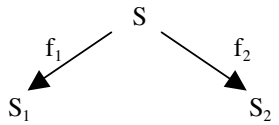
7.5.2 Proposition

Every concrete category $\Sigma:\mathbf{D}\to\mathbf{C}$ that has discrete structures is reflective, the reflections (i.e. the components of the unit) being identities.

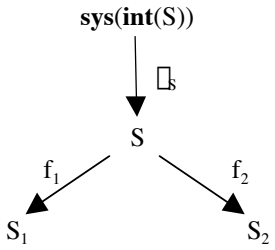
The proof of this result is immediate once one transcribes the definition of discrete lifts to diagrams:



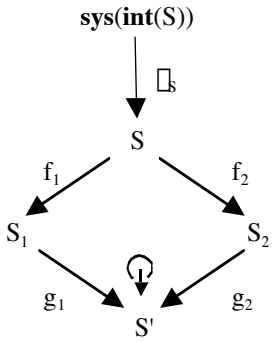
Notice that, Σ being faithful, the co-reflections are epis (7.2.11). Actually, this is the property that allows us to replace the middle objects that perform interconnections by the discrete lifts of their underlying interfaces. Indeed, denoting by ***sys*** the reflector of ***int***, every diagram



extends to



Both diagrams admit the same pushouts because, η_S being epi,



$\eta_S;f_1;g_1=\eta_S;f_2;g_2$ implies $f_1;g_1=f_2;g_2$.

7.5.3 Exercise

Prove that both diagrams have, indeed, the same pushouts.

He have now all the ingredients for our proposed characterisation of the formalisms that separate "Coordination" from "Computation":

7.5.4 Definition – coordinated category

A concrete category (faithful functor) $U:\mathbf{D}\to\mathbf{C}$, is said to be *coordinated* when:

- U lifts colimits
- U has discrete structures

In these circumstances, we also say that $\mathbf{D}$ is coordinated over $\mathbf{C}$ (via the functor U).

We have omitted the requirement on the preservation of colimits. This is because:

7.5.5 Exercise

Prove that coordinated functors preserve colimits.

As examples, we have already seen that theories and theory presentations of any (π)institution constitute a concrete category that is coordinated over their signatures, and that the (dual of) ***PROC*** is coordinated over (the dual of) the category of alphabets. In Part Three of the book, we will see an example related to architectural description languages, the language CommUnity. We end this section with a "genuine" example: a simplified version of the language Gamma [11], which is based on the chemical reaction paradigm [16].

Before that, we would like to point out that the properties that characterise ***SYS*** as being coordinated over ***INT*** make ***SYS*** "almost" topological over ***INT***. To be topological [1], ***int*** would have to lift colimits uniquely, which would make the concrete category amnestic (6.1.4). As far as the algebraic properties of the underlying formalism are concerned, this is not a problem because every concrete category can be modified to produce an amnestic, concretely equivalent version. However, and although ***PROC*** is indeed amnestic, ***PRES***, for instance, is not and neither is CommUnity. This is the "closest" characterisation we have to a "classical" mathematical structure: topological categories abound in Mathematics and other areas of Computer Science. In the areas related to Software Engineering, namely those in which one welcomes, or cannot avoid, "user intervention", one tends to work "up to isomorphism" more than "up to equality". In the case of the lifting of colimits, this means that there can be room for choosing between different, but isomorphic, system representations, for instance, alternative presentations of the same theory: one tends not to care whether a given conjunction ends up represented as $a \wedge b$ or $b \wedge a$.

7.5.6 Definition – Gamma programs

A Gamma program P consists of:

- a signature $\Sigma=<S,\Omega,\Pi>$, where S is a set of sorts, Ω is a set of operation symbols and Π is a set of relation symbols, representing the data types that the program uses;
- a set of reactions, each of which is of the form:

  $$R \equiv X,\ t_1,\ ...,\ t_n \rightarrow t'_1,\ ...,\ t'_m \Leftarrow c$$

  where
  1. X is a set (of variables); each variable is typed by a data sort in S;
  2. $t_1,\ ...,\ t_n \rightarrow t'_1,\ ...,\ t'_m$ is the action of the reaction – a pair of sets of terms over X;
  3. c is the reaction condition – a proposition over X.

An example of a Gamma program is the following producer of burgers and salads from, respectively, meat and vegetables:

```
PROD: sorts        meat, veg, burger, salad
      ops          vprod: veg → salad, mprod: meat → burger
      reactions    m:meat, m → mprod(m)
                   v:veg, v → vprod(v)
```

The parallel composition of Gamma programs, as defined in [11], is a program consisting of all the reactions of the component programs. Its behaviour is obtained by executing the reactions of the component programs in any order, possibly in parallel. This leads us to the following notion of morphism.

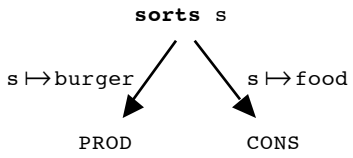
7.5.7 Definition – morphisms of Gamma programs

A morphism σ between Gamma programs P_1 and P_2 is a morphism between the underlying data signatures such that $\sigma(P_1) \subseteq P_2$, i.e., P_2 has more reactions than P_1.

In order to illustrate system configuration in Gamma, let us consider that we want to interconnect the producer with the following consumer:

```
CONS: sorts        food, waste
      ops          cons: food → waste
      reactions    f:food, f → cons(f)
```

The interconnection of the two programs is based on the identification of the food the consumer consumes, that is, the interconnection is established between their data types. For instance, the coordination of the producer and the consumer based on meat is given by the following interconnection:



Gamma is, indeed, coordinated over the category of data types:

- the forgetful functor *dt* from Gamma programs to data types is faithful;
- given any diagram in the category Gamma, a colimit $\sigma_i:(dt(P_i) \Rightarrow \sigma)i:\mathbf{I}$ of the corresponding diagram in the category of data types is lifted to the following colimit of programs $\sigma_i:(P_i \Rightarrow \langle \sigma, \sigma\sigma_j(R_j)\rangle)i:\mathbf{I}$;
- the discrete lift of a data type is the program with the empty set of reactions.

7.5.8 Exercise

Workout the full characterisation of the category of Gamma programs and prove that it is indeed coordinated over the data types.

PART THREE – applications

8 Community

8.1 A language for program design

CommUnity is a language similar to Unity [19] and Interacting Processes [47]. It was initially developed [44] to show how "programs" could fit into Goguen's categorical approach to General Systems Theory. Since then, the language and the design framework have been extended to provide a formal platform for testing ideas and experimenting techniques for the architectural design of open, reactive, and reconfigurable systems.

One of the extensions that we have made to CommUnity since its original definition concerns the support for higher levels of design. At its earlier stages, the architecture of the system is normally given in terms of components that are not necessarily programs but abstractions of programs – called *designs* – that can be *refined* into programs in later steps of the development process. Designs may also account for components of the real-world with which the software components will be interconnected. Typically, such abstractions derive from requirements that have been specified in some logic or other mathematical models of the behaviour of real-world components.

The goal of supporting abstraction is not only to address a typical stepwise approach to software *construction*, but also the definition of an architectural design layer that is close enough to the application domain for the evolution of the system to be driven directly as a reflection of the changes that occur in the domain. An important part of this evolution may consist of changes in the nature of components, with real-world components being replaced or controlled by software components, or software components being reprogrammed in another language.

The support for abstraction in CommUnity is twofold. On the one hand, designs account for what is usually called *underspecification*, i.e. they are structures that do not denote unique programs but collections of programs. On the other hand, designs can be defined over a collection of data types that do not correspond necessarily to those that will

be available in the final implementation platform. Therefore, there are two refinement procedures that have to be accounted for in CommUnity. On the one hand, the removal of underspecification from designs in order to define programs over the layer of abstraction defined by the data types that have been used. On the other hand, the reification of the data types in order to bring programs into the target implementation environment.

The choice of data types determines, essentially, the nature of the elementary computations that can be performed locally by the components, which are abstracted as operations on data elements. Such elementary computations also determine the granularity of the services that components can provide and, hence, the granularity of the interconnections that can be established at a given layer of abstraction. Nevertheless, data refinement is more concerned with the computational aspects of systems than with the coordination mechanisms that are responsible for interactions among system components. Because the support that Category Theory can provide to the specification of abstract data types is already well established and available in the literature, even through books [8,28,29,75,94], we shall not address this aspect of CommUnity in depth but, rather, concentrate on the broader architectural aspects, giving more emphasis to refinement of designs for a fixed choice of data types and omitting any discussion on data refinement.

Given this, we shall assume a fixed collection of data types. In order to remain independent of any specific language for the definition of these data types, we take them in the form of a first-order algebraic specification. That is to say, we assume a data signature $<S,\Sigma>$, where S is a set (of sorts) and Σ is a $S^* \times S$-indexed family of sets (of operations), to be given together with a collection Φ of first-order sentences specifying the functionality of the operations.

A CommUnity design for a component over such a data type specification is of the form

```
design P is
out    out(V)
in     in(V)
prv    prv(V)
do
[]_{g ∈ sh(Σ)}        g[D(g)]: L(g), U(g) ⇝ R(g)
[]_{g ∈ prv(Σ)} prv   g[D(g)]: L(g), U(g) ⇝ R(g)
```

where

- V is a set (of *communication channels*). A communication channel (or, simply, channel) can be declared as *input*, *output* or *private*. Each channel v is typed with a sort $sort(v) \in S$ that reflects the nature of the data that is exchanged through it.

Input channels are used for reading data from the environment of the component. The component has no control on the values that are made available in such channels. Moreover, reading a value from an input channel does not "consume" it: the value remains available until the environment decides to replace it.

Output and private channels are controlled locally by the component, i.e. the values that, at any given moment, are available on these channels cannot be modified by the environment. Output channels allow the environment to read data produced by the component. Private channels support internal activity that does not involve the environment in any way. We use *loc(V)* to denote the union *prv(V)*$\cup$*out(V)*, i.e. the set of local channels.

In some of the earlier papers on CommUnity, we have named the elements of *V variables* or *attributes*. The change from *variables* to *channels* aims at reinforcing the idea that the elements of *V* correspond to means that components have to communicate rather than "store" data. This is consistent with the "black-box" view of components that we intend to model, which should hide the representation of the state of components and provide only means for it to be observed.

Channels cater for asyncronous communication between components in the sense that reading and writing into a channel are independent operations: a value that is written on a channel will remain there, regardless of how many times it is read, until it is overwritten.

- $\Box$ is a set (of *action names*). The named actions can be either *private* or *shared* (for simplicity, we only declare which actions are private).

  Private actions represent internal computations in the sense that their execution is uniquely under the control of the component.

  Shared actions are used for synchronous interactions between the component and the environment, meaning that their execution is also under the control of the environment.

  The significance of naming actions will become obvious below; the idea is to provide points of *rendez-vous* at which components can synchronise, for instance as a means of ensuring that the right values are being exchanged through the channels.
- For each action name *g*, the following attributes are defined:
  - *D(g)* is a subset of *loc(V)* consisting of the local channels into which executions of the action can write. This is what is sometimes called the *write frame* of *g*. For simplicity, we will omit the explicit reference to the write frame when *R(g)* is a conditional multiple assignment (see below), in which case *D(g)* can be inferred from the assignments. Given a local channel *v*, we will denote by *D(v)* the set of actions *g* such that *v*$\in$*D(g)*, i.e. the actions that write into *v*.

- *L(g)* and *U(g)* are two conditions such that *U(g)* *L(g)*. These conditions establish an interval in which the enabling condition of any guarded command that implements *g* must lie. The condition *L(g)* is a lower bound for enabledness in the sense that it is implied by the enabling condition. Therefore, its negation establishes a *blocking* condition. On the other hand, *U(g)* is an upper bound in the sense that it implies the enabling condition, therefore establishing a *progress* condition. Hence, the enabling condition is fully determined only if *L(g)* and *U(g)* are equivalent, in which case we write only one condition.
- *R(g)* is a condition on *V* and *D(g)'* where by *D(g)'* we denote the set of primed local channels from the write frame of *g*. As usual, primed channels account for references to the values that the channels display after the execution of the action. These conditions are usually a conjunction of implications of the form *pre* *pos* where *pre* does not involve primed channels. They correspond to pre/post-condition specifications in the sense of Hoare. When *R(g)* is such that the primed version of each local channel in the write frame of *g* is fully determined, we obtain a conditional multiple assignment, in which case we use the notation that is normally found in programming languages. When the write frame *D(g)* is empty, *R(g)* is tautological, which we denote by *skip*.

Notice that CommUnity supports several mechanisms for underspecification – actions may be underspecified in the sense that their enabling conditions may not be fully determined (subject to refinement by reducing the interval established by *L* and *U*) and their effects on the variables may also be undetermined.

When, for every $g \square \square$, *L(g)* and *U(g)* coincide, and the relation *R(g)* defines a conditional multiple assignment, then the design is called a *program* and the traditional notation for guarded commands is used. Notice that a program with a non-empty set of input channels is *open* in the sense that its execution is only meaningful in the context of a configuration in which these inputs have been connected with local outputs of other components. The notion of configuration, and the execution of an open program in a given configuration, will be discussed further below. The behaviour of a closed program is as follows. At each execution step, one of the actions whose enabling condition holds of the current state is selected, and its assignments are executed atomically. Furthermore, private actions that are infinitely often enabled are guaranteed to be selected infinitely often. See [76] for a model-theoretic semantics of CommUnity.

Designs can be parameterised by data elements (sorts and operations) indicated after the name of the component (see an example below). These parameters are instantiated at configuration time, i.e. when a spe-

cific component needs to be included in the configuration of the system being built, or as part of the reconfiguration of an existing system.

As an example, consider the following parameterised design:

```
design buffer [t:sort, bound:nat] is
in     i:t
out    o:t
prv    rd: bool, b: list(t)
do     put: |b|<bound → b:=b.i
[] prv next: |b|>0 ∧ rd → o:=head(b) ‖ b:=tail(b) ‖ rd:=true
[]     get: rd → rd:=false
```

The parameters of this design consist of the sort *t* of data elements that the buffer can handle and the capacity *bound* of the buffer. The buffer itself is defined over a list with elements of *t*. As already discussed, we are assuming that the data type *list* is available through an algebraic specification that includes the traditional operations such as |_| returning the current size of the list, *head(_)* returning the first element of the list, *tail(_)* returning the list after the first element, and _._ for appending an element to the end of the list.

This design is actually a (parameterised) program and the traditional notation of guarded commands was used accordingly. Notice in particular that the reference to the write frame of the actions was omitted: it can be inferred from the multiple assignments that they perform. As already mentioned, because we are dealing with multiple assignments, the traditional notation involving the symbol := was used instead of the logical language over channels and their primed versions. In the case above, this corresponds to:

$R(put): \Box i' = b.i$
$R(next): \Box o' = head(b) \wedge b' = tail(b) \wedge rd'$
$R(get): \Box \neg rd'$

This program models a buffer with a limited capacity and a FIFO discipline. It can store, through the action *put*, messages of sort *t* received from the environment through the input channel *i*, as long as there is space for them. The buffer can also discard stored messages, making them available to the environment through the output channel *o* and the action *next*. Naturally, this activity is possible only when there are messages in store and the current message in *o* has already been read by the environment (which is modelled by the action *get* and the private channel *rd*).

In order to illustrate the ability of CommUnity to support higher-level component design, we present below the design of a typical sender of messages.

```
design sender[t:sort] is
out    o:t
prv    rd: bool
do     prod[o,rd]: ¬rd,false ⇒ rd'
[]     send[rd]: rd,false ⇒ ¬rd'
```

In this design, we are primarily concerned with the interaction between the sender and its environment, ignoring details of internal computations such as the production of messages. This is why the output channel *o* is included in the write frame of *prod* but *R(prod)* does not place any constraint on how it is updated. Notice that the component *sender* cannot produce another message before the previous one has been processed: after producing a message, the sender expects an acknowledgement (modelled through the execution of *send*) to produce a new message.

In order to leave unspecified when and how many messages the *sender* will send and in which situations it will produce a new message, the progress conditions of *prod* and *send* are false. Furthermore, the discipline of production is also left completely unspecified: the action *prod* includes the output channel *o* in its write frame but the design does not commit to any specific way of updating the values in this channel.

From a mathematical point of view, (instantiated) CommUnity designs are structures defined as follows.

8.1.1 Definition – signatures and designs

A *signature* in CommUnity is a tuple $<V,\Gamma,tv,ta,D>$ where

- V is an S-indexed family of mutually disjoint finite sets,
- Γ is a finite set,
- $tv:\bigcup_{s\in S} V_s \to \{out,in,prv\}$ is a total function,
- $ta:\Gamma \to \{sh,prv\}$ is a total function,
- $D:\Gamma \to 2^{loc(V)}$ is a total function.

A *design* in CommUnity is a pair $<\Sigma,\Delta>$ where $\Sigma=<V,\Gamma,tv,ta,D>$ is a signature and Δ, the body of the design, is a tuple $<R,L,U>$ where:

- R assigns to every action $g\in\Gamma$, a proposition over $V\cup D(g)'$,
- L and U assign a proposition over V to every action $g\in\Gamma$.

The reader who is familiar with parallel program design languages or earlier versions of CommUnity will have probably noticed the absence of initialisation conditions. The reason they were not included in CommUnity designs is because they are part of the configuration language of CommUnity, not the parallel program design language. That is to say, we take initialisation conditions as part of the mechanisms that relate to the building and management of configurations out of designs, not of the construction of designs themselves.

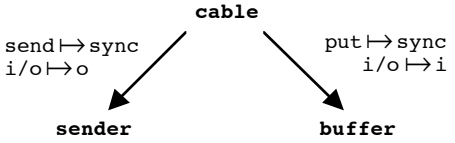
8.2 Interconnecting designs

So far, we have presented the primitives for the design of individual components, which are another variation on guarded commands, albeit with some “twists” of originality such as the use of an interval as a specification for the enabling conditions of commands. The main distinguishing features of CommUnity are those that concern design "in the large", i.e. the ability to design large systems from simpler components.

The model of interaction between components in CommUnity is based on action synchronisation and the interconnection of input channels of a component with output channels of other components. These are standard means of interconnecting software components. What distinguishes CommUnity from other parallel program design languages is the fact that such interactions between components have to be made explicit by providing the corresponding name bindings. Indeed, parallel program design languages normally leave such interactions implicit by relying on the use of the same names in different components. In CommUnity, names are local to designs. This means that the use of the same name in different designs is treated as being purely accidental, and, hence, expresses no relationship between the components.

In CommUnity, name bindings are established as relationships between the signatures of the corresponding components, matching channels and actions of these components. These bindings are made explicit in configurations. A configuration determines a diagram containing nodes labelled with the components that are part of the configuration. Name bindings are represented as additional nodes representing the actual interactions, and edges labelled with the projections that map each interaction to the signatures of the corresponding components.

For instance, a configuration in which the messages from a *sender* component are sent through a bounded buffer is defined the following diagram:



The node labelled *cable* is the representation of the set of bindings. It stands for the following design

```
design cable[t:sort] is
in     i/o:t
do     sync:true,false  skip
```

By using the word "cable" we mean to suggest analogies with the use of physical cables for interconnecting mechanical or electrical components. Because, as we have seen, channels and action names are typed and classified in different categories, not every pair of names is a valid binding. To express the rules that determine valid bindings, it is convenient to define *cable* as a component itself (just like electric cables are made of coloured wires so that we know what should be connected to what). Hence, in the case above, *cable* consists of an input channel *i/o* to model the medium through which data is to be transmitted between the sender and the buffer, and a shared action *sync* for the two components to synchronise in order to transmit the data. Because, as we have already mentioned, names in CommUnity are local, the identities of the shared input channel and the shared action in *cable* are not relevant: they are just placeholders for the projections to define the relevant bindings.

The bindings themselves are established through the labels of the edges of the diagram. In the case above, the input channel of *cable* is mapped to the output channel *o* of *sender* and to the input channel *i* of *buffer*. This establishes an i/o-interconnection between *sender* and *buffer*. On the other hand, the actions *send* of *sender* and *put* of *buffer* are mapped to the shared action of *cable*. This defines that *sender* and *buffer* must synchronise each time either of them wants to perform the corresponding action. The fact that the mappings on action names and on channels go in opposite directions will be discussed below.

Notice that *sync* does not perform any activity: it just provides the place for the "rendez-vous" between the sender and the buffer to take place. This is in analogy with cables that are completely neutral, i.e. they do not interfere with the computations that are going on in the components. Hence, cables are, essentially, signatures. This observation will be formalised later on in the section.

The arrows that we are using to define interconnections between components are also mathematical objects: they are examples of signature morphisms.

8.2.1 Definition – signature morphisms

A morphism $\sigma:\Sigma_1 \to \Sigma_2$ of signatures $\Sigma_1=\langle V_1,\Box_1,tv_1,ta_1,D_1\rangle$ and $\Sigma_2=\langle V_2,\Box_2,tv_2,ta_2,D_2\rangle$ is a pair $\langle\sigma_{ch},\sigma_{ac}\rangle$ where

- $\sigma_{ch}: V_1 \to V_2$ is a total function satisfying:
  1. $sort_2(\sigma_{ch}(v))=sort_1(v)$ for every $v \in V_1$
  2. $\sigma_{ch}(o) \in out(V_2)$ for every $o \in out(V_1)$

3. $\sigma_{ch}(i) \in out(V_2) \cup in(V_2)$ for every $i \in in(V_1)$
4. $\sigma_{ch}(p) \in prv(V_2)$ for every $p \in prv(V_1)$

- $\sigma_{ac} : \Gamma_2 \rightharpoonup \Gamma_1$ is a partial mapping satisfying for every $g \in \Gamma_2$ s.t. $\sigma_{ac}(g)$ is defined:

5. if $g \in sh(\Gamma_2)$ then $\sigma_{ac}(g) \in sh(\Gamma_1)$
6. if $g \in prv(\Gamma_2)$ then $\sigma_{ac}(g) \in prv(\Gamma_1)$
7. $\sigma_{ch}(D_1(\sigma_{ac}(g))) = D_2(g)$
8. σ_{ac} is total on $D_2(\sigma_{ch}(v))$ and $\sigma_{ac}(D_2(\sigma_{ch}(v))) = D_1(v)$ for every $v \in loc(V_1)$

Signature morphisms represent more than the projections that arise from name bindings as illustrated above. A morphism σ from Γ_1 to Γ_2 is intended to support the identification of a way in which a component with signature Γ_1 is embedded in a larger system with signature Γ_2. This justifies the various constructions and constraints in the definition.

The function σ_{ch} identifies for each channel of the component the corresponding channel of the system. The partial mapping σ_{ac} identifies the action of the component that is involved in each action of the system, if ever. The fact that the two mappings go in opposite directions is justified as follows. Actions of the system constitute synchronisation sets of actions of the components. Because not every component is necessarily involved in every action of the system, the action mapping is partial. On the other hand, because each action of the component may participate in more than one synchronisation set, but each synchronisation set cannot induce internal synchronisations within the components, the relationship between the actions of the system and the actions of every component is functional from the former to the latter. Hence, actions will be dealt with in the category ***PAR*** of partial functions. As seen in 7.1.12, this category is equivalent to the category that we used for modelling alphabets of processes, meaning that the intuitions that we developed on the way universal constructions capture composition can be used for CommUnity as well.

Input/output communication within the system is not modelled in the same way as action synchronisation. Synchronisation sets reflect parallel composition whereas with i/o-interconnections we wish to merge communication channels of the components. This means that, in the system, channels should be identified rather than paired. This is why mappings on channels and mappings on actions go in opposite directions. We will see that, as a result, the mathematical semantics of configuration diagrams induces fibred products of actions (synchronisation sets) and amalgamated sums of channels (equivalence classes of connected channels).

The constraints are concerned with typing. Sorts associated with channels have to be preserved but, in terms of their classification, input

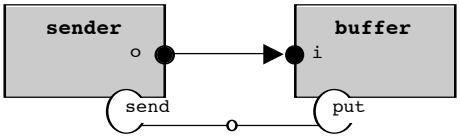
channels of a component may become output channels of the system in the sense that, as a default, they should remain open for communication with other components. In most languages for parallel design, the default is to hide the communication, what in CommUnity would correspond to classify the resulting channel as being private. In our opinion, closing/hiding the channel should not be a default but a design decision that should be performed explicitly. Hence, in CommUnity, mechanisms for internalising communication can be applied but they are not the default in a configuration. The last two conditions on write frames (7 and 8) imply that actions of the system in which a component is not involved cannot have local channels of the component in its write frame. That is, change within a component is completely encapsulated in the structure of actions defined for the component.

Given the ingredients out of which signatures are assembled, the proof of the following result is pure routine and left as an exercise:

8.2.2 Proposition – category of signatures

Signatures in CommUnity together with their morphisms constitute a category that we shall denote by ***c-SIGN***.

The notation can be simplified and made more friendly by adopting features that are typical of languages for configurable distributed systems like [89]. For instance, the interconnection defined before can be described as follows.



The notation should be self-explaining. Components are represented through boxes, their channels through bullets and their actions through circles. Normally, we only depict the actions and channels involved in the configuration. Hence, as discussed below, private actions and channels do not figure up.

Interconnections, i.e. name bindings, are still represented explicitly but, instead of being depicted as a component, the cable is now represented, perhaps more intuitively, in terms of arcs that connect channels and actions directly. The direction of the arcs is from output to input channels. Configurations in this notation are easily translated into categorical diagrams by transforming the interconnections into channels and morphisms, something which, again, we shall abstain from formalising here.

So far, we have explained how interconnections between components can be established at the level of the signatures of their designs. It re-

mains to explain how the corresponding designs are interconnected, i.e. what is the semantics of the configuration diagram once designs are taken into account. For that purpose, we need to extend the notion of morphism from signatures to designs.

8.2.3 Definition/proposition – design morphisms

A morphism $\Phi:P_1 \to P_2$ of designs $P_1=<\Sigma_1,\Delta_1>$ and $P_2=<\Sigma_2,\Delta_2>$, consists of a signature morphism $\sigma:\Sigma_1 \to \Sigma_2$ such that, for every $g \in \Sigma_2$ for which $\Phi_{ac}(g)$ is defined:

1. $\Gamma \vdash R_2(g) \Rightarrow R_1(\Phi_{ac}(g))$
2. $\Gamma \vdash L_2(g) \Rightarrow L_1(\Phi_{ac}(g))$
3. $\Gamma \vdash U_2(g) \Rightarrow U_1(\Phi_{ac}(g))$

where Γ is the axiomatisation of the data type specification, $\vdash$ denotes validity in the first-order sense, and σ is the extension of σ to the language of expressions and conditions as for institutions 6.5.3. We will normally simplify the notation and overload the use of σ in place of σ (6.5.4). Designs and their morphisms constitute a category ***c-DSGN***. This category is concrete over ***c-SIGN*** through the obvious forgetful functor.

A morphism $\Phi:P_1 \to P_2$ identifies a way in which P_1 is "augmented" to become P_2 so that P_2 can be considered as having been obtained from P_1 through the superposition of additional behaviour, namely the interconnection of one or more components. The conditions on the actions require that the computations performed by the system reflect the interconnections established between its components. Condition 1 reflects the fact that the effects of the actions of the components can only be preserved or made more deterministic in the system. This is because the other components in the system cannot interfere with the transformations that the actions of a given component make on its state, except possibly by removing some of the underspecification present in the component design.

Conditions 2 and 3 allow the bounds that the component design specifies for the enabling of the action to be strengthened but not weakened. Strengthening of the lower bound reflects the fact that all the components that participate in the execution of a joint action have to give their permission for the action to occur. On the other hand, it is clear that progress for a joint action can only be guaranteed when all the designs of the components involved can locally guarantee so.

The notion of morphism that we have just defined captures what in the literature on parallel program design is called "superposition" or "superimposition" [19,47,71]. See [44] for the categorical formalisation of different notions of superposition and their algebraic properties.

The semantics of configurations is given by a categorical construction: the colimit of the underlying diagrams. As we have already explained in chapter 4, taking the colimit of a diagram collapses the configuration into an object by internalising all the interconnections, thus delivering a design for the system as a whole. Furthermore, the colimit provides a morphism ι_i from each component design P_i in the configuration into the new design (that of the system) – the edge of the co-cone at P_i (see 4.4.1). Each such morphism is essential for identifying the corresponding component within the system because the construction of the new design typically requires that the features of the components be renamed in order to account for the interconnections.

Again, given the nature of the "ingredients", it is not difficult to understand how colimits of designs work: because channels are handled through total functions, colimits amalgamate channels (4.3.2); and because actions are handled as partial functions in the opposite direction, i.e. in the dual category, colimits operate on actions as limits and compute fibred products (4.3.8). For instance, in the case of actions, the colimit represents every synchronisation set $\{g_1,\ldots,g_n\}$ of actions of the components, as defined through the interconnections, by a single action $g_1\|\ldots\|g_n$ whose occurrence captures the joint execution of the actions in the set (recall 4.3.8 and 4.4.11). Because limits perform conjunctions of logical conditions (4.2.6), the transformations performed by a joint action are specified by the conjunction of the specifications of the local effects of each of the synchronised actions:

$$R(g_1\|\ldots\|g_n)=\iota_1(R(g_1))\sqcap\ldots\sqcap\iota_n(R(g_n))$$

where the ι_i are the morphisms that connect the components to the system (the edges of the co-cone). The bounds on the guards of joint actions are also obtained through the conjunctions of the bounds specified by the components, i.e.

$$L(g_1\|\ldots\|g_n)=\iota_1(L(g_1))\sqcap\ldots\sqcap\iota_n(L(g_n))$$
$$U(g_1\|\ldots\|g_n)=\iota_1(U(g_1))\sqcap\ldots\sqcap\iota_n(U(g_n))$$

Finally, because morphisms require inclusions of write frames, the write frame of a joint action is given by the union of the (translations of) the write frames of the component actions. This way of computing colimits derives from the strong algebraic properties of the category of designs. More precisely:

8.2.4 Proposition

The forgetful functor ***c-sign*** that maps CommUnity designs to the corresponding signatures defines ***c-DSGN*** as a category coordinated over ***c-SIGN*** (7.5.4). We shall call a *cable* the discrete lift of a signature: given a signature Σ, the corresponding cable ***dsgn***(Σ) has Σ for signature

and, for every action g, $R(g)$, $L(g)$ and $U(g)$ are all *true*, what we normally denote by *skip*.

Summarising, colimits in CommUnity capture a generalised notion of parallel composition in which the designer makes explicit what interconnections are used between components. Because the category of designs is coordinated over signatures, all interconnections can be performed through cables, i.e. they do not involve the computational part of components, only their "interfaces"– i/o communication through channels and rendez-vous through action synchronisation. We can see this operation as a generalisation of the notion of superimposition as defined in [47].

The colimit of the configuration, when it returns a closed program, can also be used for providing an operational semantics for the system thus configured: as explained in section 8.1, at each execution step, any action whose guard is true can be executed, with the guarantee that private actions that are infinitely often enabled are selected infinitely often. Because actions of the system are synchronisation sets of actions of the components, the evaluation of the guard of the chosen action can be performed in a distributed way by evaluating the guards of the component actions in the synchronisation set. According to the semantics that we have just given, the joint action will be executed iff all the local guards evaluate to *true*. The execution of the multiple assignment associated with the joint action can also be performed in a distributed way by executing each of the local assignments. What is important is that the atomicity of the execution is guaranteed, i.e. the next system step should only start when all local executions have completed, and the i/o-communications should be implemented so that every local input channel is instantiated with the correct value – that which holds of the local state before any execution starts (synchronicity).

Hence, the colimit of the configuration diagram should be seen as an abstraction of the actual distributed execution that is obtained by coordinating the local executions according to the interconnections, rather than the program that is going to be executed as a monolithic unit. The fact that the computational part, i.e. the one that is concerned with the execution of the actions on the state, can be separated from the coordination aspects is, therefore, an essential property for guaranteeing that the operational semantics is compositional on the structure of the system as given through its configuration diagram.

Not every diagram of designs reflects a meaningful configuration. For instance, it does not make sense to interconnect components by connecting two output channels because it will almost inevitably lead to conflicts between the actions that output to these channels. Such constraints are not "structural": they cannot be captured by morphisms between designs because they concern general interconnections, not just

the component-of relationship. That is to say, the constraints are not "technical" but, rather, "methodological".

8.2.5 Definition/proposition – well-formed configurations

Let ***chan*** be the forgetful functor from ***c-DSGN*** to ***SET*** that maps designs to their underlying sets of channels. A *configuration* is a finite diagram **dia**:$\boldsymbol{I} \to$ ***c-DSGN*** together with a subset J of $|\boldsymbol{I}|$ (the nodes that represent the components being interconnected) such that:

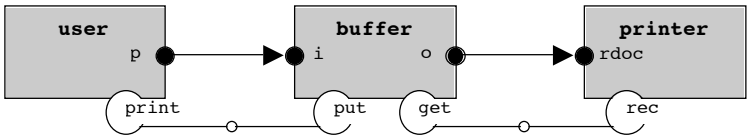
1. For every $f:i \to j$ in $\boldsymbol{I}$, either $i=j$ and $f=id_i$; or $j \in J$ and $i \in J$ and ***dia***(i) is a cable;
2. For every $i \in |\boldsymbol{I}| \setminus J$ s.t. ***dia***(i) is a cable, there exist distinct nodes $j,k \in |\boldsymbol{I}|$ with morphisms $f:i \to j$ and $g:i \to k$;
3. If $\{\lambda_i$:***chan***(***dia***$(i)) \to V: i \in |\boldsymbol{I}|\}$ is a colimit of ***dia***;***chan*** then, for every $v \in V$, there exists at most one $i \in |\boldsymbol{I}|$ s.t. $\lambda_i^{-1}(v) \cap out(V_{dia(i)}) \neq \varnothing$ and, for such i, $\lambda_i^{-1}(v) \cap out(V_{dia(i)})$ is a singleton.

We further say that a configuration ***dia*** is *well-formed* if for every $i \in |\boldsymbol{I}| \setminus J$ s.t. ***dia***(i) is a cable, ***dia***(i) has neither private actions nor private channels.

Condition 1 states that the elementary interconnections are established through cables. Condition 2 ensures that a configuration diagram does not include cables that are not used. Finally, condition 3 prevents the identification of output channels. The explicit reference to the subset J of components is necessary because the distinction between nodes that are being used as channels and as components is a pragmatic, not formal one: it is possible that, in a given configuration, a node is intended to represent a component but, because it is still totally underspecified, it is the discrete lift of a signature, i.e. what we have called a cable.

Well-formed configurations are such that private actions and channels are not involved in the interconnections, i.e. they support the intuitive semantics we gave in section 8.1 according to which private channels cannot be read by the environment and that the execution of shared actions is uniquely under the control of the component.

An example of a more complex configuration is given below. It models the interconnection between a user and a printer via a buffer.



The user produces files that it stores in the private channel w. It can then convert them either to postscript or pdf formats, after which it makes them available for printing in the output channel p.

```
design user is
out    p: ps+pdf
prv    s,t: bool, w: Lowtex
do     work[w,s,t]: ¬t,false □ t'
[]     pr_ps:□¬s□t,false □ p:=ps(w) ‖ s:=true
[]     pr_pdf:□¬s□t,false □ p:=pdf(w) ‖ s:=true
[]     print:□s □ s:=false ‖ t:=false
```

The printer copies the files it downloads from the input channel *rdoc* into the private channel *pdoc*, after which it prints them.

```
design printer is
in     rdoc: ps+pdf
prv    busy: bool, pdoc: ps+pdf
do     rec:□□busy □ pdoc:=rdoc ‖ busy:=true
[] prv end_print:□busy □ busy:=false
```

The configuration connects the user to the printer via a buffer as expected. The user "prints" by placing the file in the buffer: this is achieved through the synchronisation set *{print,put}* and the i/o-interconnection *{p,i}*. The printer downloads from the buffer the files that it prints: this is achieved through the synchronisation set *{get,rec}* and the i/o-interconnection *{o,rdoc}*.

The design of the system that results from the colimit of the configuration diagram contains two channels that account for the two i/o-interconnections *{p,i}* and *{o,rdoc}*, together with the private channels of the components. At the level of its actions, it generates the following shared actions (synchronisation sets):

print|put, get|rec

as required by the interconnections, and

work, pr_ps, pr_pdf,
work|get|rec, pr_ps|get|rec, pr_pdf|get|rec

which reflect the concurrent executions that respect the interconnections.

No other shared actions are possible because of the synchronisation requirements imposed on the components.

8.3 Refining designs

The notion of morphism defined in the previous section does not capture a refinement relation in the sense that it does not ensure that any implementation of the target provides an implementation for the source. For instance, it is easy to see that morphisms do not preserve the interval assigned to the guard of each action. Given that the aim of

the defined morphisms was to capture the relationship that exists between systems and their components, this is hardly surprising. The same holds in languages such as CSP [68]: in the failure or ready semantics, parallel composition does not induce refinement – $P \| Q$ is not necessarily a refinement of P – and refinement cannot always be expressed as the result of a parallel composition – P may refine Q and, yet, there may not exist a Q' such that P is $Q \| Q'$.

Because refinement is an important dimension in structuring software development, it is only natural that we investigate ways of supporting it in a categorical setting. This would be especially useful for analsying the way refinement and composition can work together. A notion of morphism can indeed be defined that captures a refinement relation for CommUnity designs.

8.3.1 Definition/proposition – refinement morphisms

A refinement morphism $\varphi: P_1 \to P_2$ of designs $P_1 = \langle \Sigma_1, \Gamma_1 \rangle$ and $P_2 = \langle \Sigma_2, \Gamma_2 \rangle$ is a pair $\langle \varphi_{ch}, \varphi_{ac} \rangle$ satisfying:

- $\varphi_{ch}: V_1 \to V_2$ is a total function satisfying, for every $v \in V_1$, $o \in out(V_1)$, $i \in in(V_1)$, $p \in prv(V_1)$:
  1. $sort_2(\varphi_{ch}(v)) = sort_1(v)$
  2. $\varphi_{ch}(o) \in out(V_2)$
  3. $\varphi_{ch}(i) \in in(V_2)$
  4. $\varphi_{ch}(p) \in prv(V_2)$
  5. $\varphi_{ch} \upharpoonright (out(V_1) \cup in(V_1))$ is injective
- $\varphi_{ac}: \Sigma_2 \to \Sigma_1$ is a partial mapping satisfying for every $g \in \Sigma_2$ s.t. $\varphi_{ac}(g)$ is defined:
  6. if $g \in sh(\Sigma_2)$ then $\varphi_{ac}(g) \in sh(\Sigma_1)$
  7. if $g \in prv(\Sigma_2)$ then $\varphi_{ac}(g) \in prv(\Sigma_1)$
  8. if $g \in sh(\Sigma_1)$ then $\varphi_{ac}^{-1}(g) \neq \emptyset$
  9. $\varphi_{ch}(D_1(\varphi_{ac}(g))) \subseteq D_2(g)$
  10. φ_{ac} is total on $D_2(\varphi_{ch}(v))$ and $\varphi_{ac}(D_2(\varphi_{ch}(v))) \subseteq D_1(v)$ for every $v \in loc(V_1)$
- for every $g \in \Sigma_2$ s.t. $\varphi_{ac}(g)$ is defined:
  11. $\Phi \vdash (R_2(g) \Rightarrow \varphi(R_1(\varphi_{ac}(g))))$
  12. $\Phi \vdash (L_2(g) \Rightarrow \varphi(L_1(\varphi_{ac}(g))))$
- for every $g_1 \in \Sigma_1$,
  13. $\Phi \vdash (\varphi(U_1(g_1)) \Leftarrow \square_{\varphi_{ac}(g_2)=g_1} U_2(g_2))$

We denote by Φ the axiomatisation of the data type specification, and by $\vdash$ the validity relation of first-order logic; φ is the extension of φ to the language of expressions and conditions.

Designs and their refinement morphisms constitute a category ***r-DSGN***. This category is concrete over ***c-SIGN*** through the functor ***r-sign*** that, like ***c-sign*,** projects designs to their signatures.

A refinement morphism identifies a way in which a design P_1 (its source) is refined by a more concrete design P_2 (its target). The function $\Box_{ch}$ identifies, for each input (resp. output) channel of P_1, the corresponding input (resp. output) channel of P_2. Notice that, contrarily to what happens with the component-of relationship as captured through design morphisms (8.2.3), refinement does not change the border between the system and its environment and, hence, input channels can no longer be mapped to output channels (3). This is also why the mapping is required to be injective on input and output channels (5): identifying channels is a configuration operation to be achieved through interconnections, not a refinement step.

As for design morphisms, refinement morphisms are required to preserve the sorts of channels (1). As discussed at the beginning of this chapter, data refinement is a dimension that, for simplicity, we are deliberately ignoring in the book. We should also point out that a more general notion of refinement can be given by mapping channels to terms defined over the language of the data types enriched by channels as constants. See [37] for details.

The mapping $\Box_{ac}$ identifies for each action g of P_1, the set $\Box_{ac}^{-1}(g)$ of actions of P_2 that implements g. This set is a "menu" of refinements that is made available for implementing action g; different choices can be made at different states to take advantage of the structures available at the more concrete design level. This menu can be empty for private actions, i.e. one may choose not to implement the private actions of the more abstract design: because private actions do not intervene in interconnections, what is important is that the overall behaviour of the component as made observable through shared actions and output channels be implemented. This is also why every shared action has to be implemented (8); again, such actions model interaction between the component and its environment, and refinement should not interfere with the border between them.

The actions for which $\Box_{ac}$ is left undefined (the new actions) and the channels which are not involved in $\Box_{ch}(V_1)$ (the new channels) introduce more detail in the description of the component. As for the "old actions", the interval defined by their blocking and progress conditions (in which the enabling condition of any implementation must lie) must be preserved or reduced (12 and 13). This is intuitive because refinement, pointing in the direction of implementations, should reduce underspecification. Hence, the lower bound cannot be weakened (12) and, contrarily to design morphisms, the upper bound cannot be strengthened (13). This is also the reason why the effects of the actions

of the more abstract design are required to be preserved or made more deterministic (11).

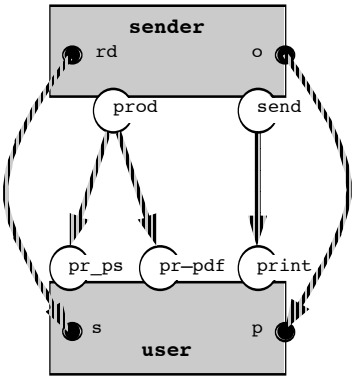
Notice that the forgetful functors ***r-sign*** and ***c-sign*** are essentially the same; they are only formally different because their sources are not the same category. Indeed, the only difference between design and refinement morphisms at the level of signatures is on the additional properties that refinement morphisms need to satisfy: 3, 5 and 8.

As an example, it is easy to see that *sender* is refined by *user* via the refinement morphism σ*:sender*σ *user* defined by

$\sigma_{ch}(o)=p$, $\sigma_{ch}(rd)=(s)$
$\sigma_{ac}(pr\_ps)=\sigma_{ac}(pr\_pdf)=prod$, $\sigma_{ac}(print)=send$

In *user*, the production of messages (to be sent) is modelled by any of the actions *pr_ps* and *pr_pdf*; the messages are made available in the output channel *p*. Notice that the production of messages, that was left unspecified in *sender,* is completely defined in *user*: it corresponds to the conversion of the files stored in *w* to ps or pdf formats.

In the simplified graphical notation that we have been using, refinement is represented through patterned arrows:



Notice that, for depicting refinement, all the actions and channels of the source should be represented, including private ones.

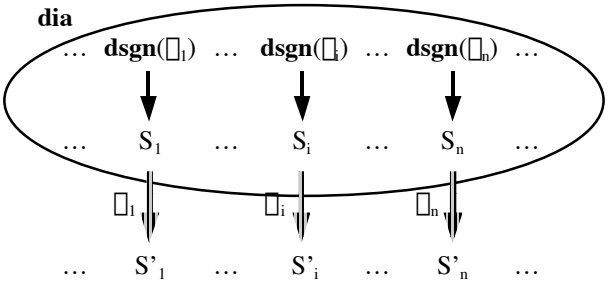
Summarising what we have built so far, we have two categories ***c-DSGN*** (8.2.3) and ***r-DSGN*** (8.3.1), both over the same notion of object – CommUnity design – but with different notions of morphism, i.e. capturing different aspects of their social lives: one tells us about their ability to relate with other designs at the same level of abstraction, and the other about the way they can be made more "concrete" by reducing underspecification. Furthermore, ***c-DSGN*** is coordinated over ***c-SIGN*** through the functor ***c-sign*** (8.2.4). As an exercise, the reader is invited to extend the study of the structural properties of ***r-DSGN***. We are now interested in the way interconnection relates to refinement.

The first important property relates to the requirement that refinement should not be based on the specificities of each particular design as far its ability to be interconnected to other designs is concerned. In other words, refinement morphisms should be such that designs that are isomorphic in ***c-DSGN*** refine, and are refined exactly by, the same designs. This is, indeed, the case:

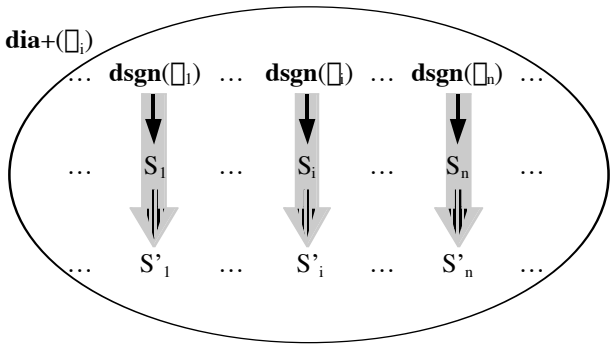
8.3.2 Proposition

Every isomorphism in ***c-DSGN*** defines an isomorphism in ***r-DSGN***.

Another crucial property is in the ability to refine a complex system from refinements of its individual components. Consider a well-formed configuration ***dia*** of a system with components $S_1,...,S_n$ and refinement morphisms $\sigma_i: S_i \to S'_i: i \in 1..n$,



By composing the morphisms σ_i with those in ***dia*** that originate in cables (designs of the form ***dsgn***(σ) where σ is a signature) and have the S_i as targets, we obtain a new diagram in ***c-DSGN*** – ***dia***$+(\sigma_i)$

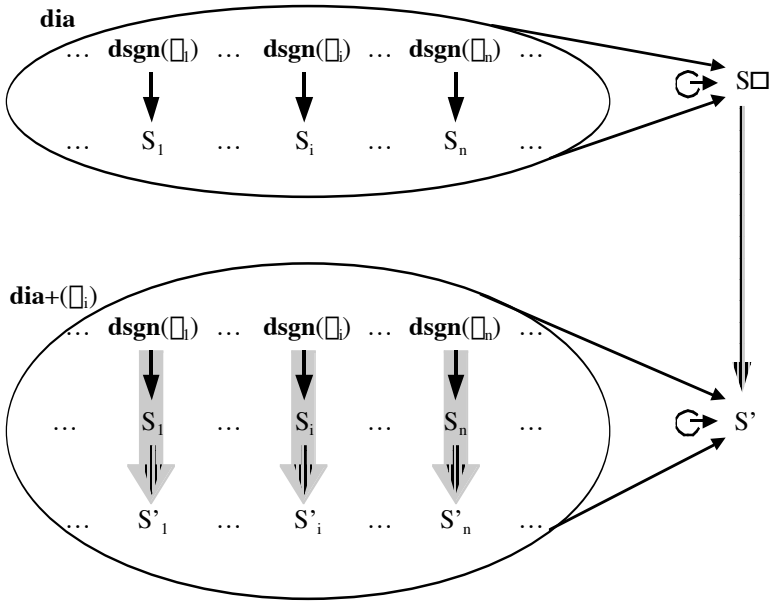


This composition is possible because, ***c-DSGN*** being coordinated over ***c-SIGN***, any morphism σ:***dsgn***$(\sigma) \to S_i$ "is" the signature morphism ***c-sign***$(\sigma): \to$ ***c-sign***(S_i); given now a refinement morphism $\sigma_i: S_i \to S'_i$, we can compose ***c-sign***(σ) with ***r-sign***(σ_i) to obtain a signature morphism: $\to$ ***c-sign***(S'_i) that can be lifted back to ***c-DSGN*** as a morphism ***dsgn***$(\sigma) \to S'_i$.

The two diagrams satisfy the following important property:

8.3.3 Proposition

In the circumstances laid out above, if *p:dia→ S* and *p':**dia**+(ι_i)→ S'* are colimits, there is a unique refinement morphism *S→ S'* that is also a morphism *p→ p'*.

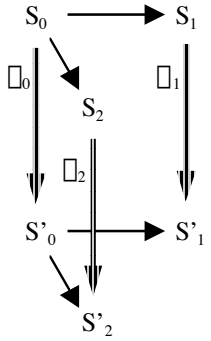


This property is another form of *compositionality*: it states that refinement of the whole can be obtained from refinements of the parts. Compositionality, as already discussed in 6.2.4, is a key issue in the design of complex systems because it makes it possible to reason about a system using the descriptions of their components at any level of abstraction, without having to know how these descriptions are refined in the lower levels (which includes their implementation).

This form of compositionality can be formulated more precisely in CommUnity by extending the notion of refinement to configurations much in the same way as we extended the notion of realisation to configurations of specifications in 6.2.4.

8.3.4 Definition – refinement of configurations

Given two configurations **dia**:***I*→ *c-DSGN*** and **dia'**:***I*→ *c-DSGN***, a *refinement of **dia** over **dia'*** is an |**I**|-indexed family $(\iota_i:\mathbf{dia}(i)\to \mathbf{dia'}(i))_{i\in|\mathbf{I}|}$ of morphisms in ***r-DSGN*** s.t., for every *f:i→ j* in ***I***, ***c-sign(dia(f));r-sign(ι_j)=r-sign(ι_i);c-sign(dia'(f))***.



In order to ensure compositionality, i.e., that the colimit of ***dia*** is refined by the colimit *of* ***dia***', it is necessary to further require that:

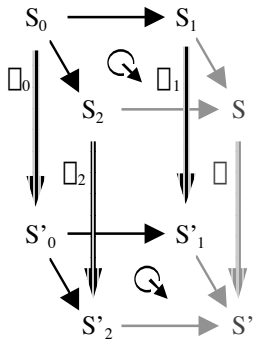
- The diagram ***dia***' cannot establish the instantiation of any input channel that was left "unplugged" in ***dia***. That is to say, the input channels of the composition are preserved by refinement.
- The diagram ***dia***' cannot establish the synchronisation of actions that were defined as being independent in ***dia***.

8.3.5 Proposition – compositionality

Consider two well-formed configurations **dia**:$I \to$ ***c-DSGN*** and **dia**':$I \to$ ***c-DSGN*** over a set J of components, and a refinement $(\sigma_i$:***dia***$(i) \to$ ***dia***'$(i))_{i \in |I|}$ of ***dia*** over ***dia***' such that, for every f:$i \to j$ in $\boldsymbol{I}$,

1. for every $v' \in in(V'_i)$, if $v' \in \sigma_i(V_i)$ then ***dia***'$(f)(v') \in \sigma_j(in(V_j))$
2. for every $g' \in \sigma'_j$, if $\sigma_j(g)$ and ***dia***'$(f)(g')$ are defined then σ_i(***dia***'$(f)(g'))$ is also defined
3. for every $i \in |\boldsymbol{I}| \setminus J$, σ_{iac} is injective

Then, there is a unique morphism σ:$S \to S'$ in ***r-DSGN*** s.t., for every $i \in |\boldsymbol{I}|$, ***c-sign***(σ_i);***r-sign***(σ)=***r-sign***(σ_i);***c-sign***(σ'_i) where $(\sigma_i$:***dia***$(i) \to S)_{i \in |\boldsymbol{I}|}$ and $(\sigma_i$:***dia***'$(i) \to S')_{i \in |\boldsymbol{I}'|}$ are colimits of **dia** and **dia**', respectively.



An outline of the proof of this result can be found in [77].

9 Architectural description

9.1 Motivation

Although components have always been considered to be the fundamental building blocks of software systems, it is in the way that the components of a system interact that the emergence of global properties of the system resides. With no interaction there is no emergence of new behaviour and, therefore, no value to the system as a whole that is not already provided through its components in isolation.

We can safely say that most of the complexity of system construction lies in the definition of the interconnections that should regulate how components interact. Designing small, encapsulated components that, through the computations that they perform locally, provide services with certain functionalities is something that can be mastered, without much difficulty, with existing methods and development techniques. Knowing how to interconnect components so that, from the interactions, the global properties that are required of the system can emerge is a totally different matter. Most of the times, it is an error prone process. What in the literature is known as the "feature interaction problem" [115] is just a symptom of this difficulty: the emergence of "strange", "unexpected" or "undesired" behaviour from feature composition is intrinsic to the use of methods for putting together systems from individual features as basic units of functionality; while we compose features having in mind the emergence of certain properties that constitute requirements on the behaviour of the system, it is difficult to predict which other forms of behaviour will also emerge, namely ones that are not of interest and whose "negation" is normally omitted from the requirements specification because one never thought of them being possible… Hence, situations like feature interaction are not problems that need to be solved but phenomena that are intrinsic to the way we build systems and that "just" need to be controlled. For that purpose, we need first-class representations of the interconnections.

This level of complexity is aggravated by the need to evolve systems. As the world of business in general becomes more and more aggressive and competitive, for instance as a consequence of the impact of the Internet and Wireless Technologies, companies need their information systems to be easily adaptable to changes in the business rules with which they operate, most of the time in a way that does not imply interruptions to the services that they provide. Quoting directly from [46], "… the ability to change is now more important than the ability to create e-commerce systems in the first place. Change becomes a first-class design goal and requires business and technology architecture whose components can be added, modified, replaced and reconfigured". All this means that the "complexity" of software has definitely shifted from *construction* to *evolution*, and that methods and technologies are required that address this new level of complexity and adaptability.

Software Architectures [49,90] is a “recent” topic in Software Engineering aimed at addressing the gross decomposition and organisation of systems in which, through so-called connectors, component interactions are recognised as being first-class design entities [100]. According to [2], an architectural connector (type) can be defined by a set of *roles* and a *glue* specification. For instance, a typical client-server architecture can be captured by a connector type with two roles – client and server – which describe the expected behaviour of clients and servers, and a glue that describes how the activities of the roles are coordinated (e.g. asynchronous communication between the client and the server). The roles of a connector type can be *instantiated* with specific components of the system under construction, which leads to an overall system structure consisting of components and connector instances establishing the interactions between the components.

The similarities between architectural constructions as informally described above and parameterised programming [55] are rather striking and have been developed in [58] in the context of the emerging interest in Software Architectures. The view of architectures that is captured by the principles and formalisms used in parameterised programming is reminiscent of Module Interconnection Languages and Interface Definition Languages [54]. This perspective is somewhat different from the one we motivated above in the sense that, whereas they capture functional dependencies between the modules that need to be linked to constitute a given program, we focus instead on the organisation of the *behaviour* of systems as compositions of components ruled by protocols for communication and synchronisation.

In this chapter, we show that the mathematical "technology" of parameterised programming can also be used for the formalisation of architectural connectors in the interaction sense. The mathematical framework that we propose for formalising architectural principles is

not specific to any particular Architecture Description Language (ADL). In fact, it will emerge from the examples that we shall provide that, contrarily to most other formalisations of SA concepts that we have seen, Category Theory is not another semantic domain for the formalisation of the description of components and connectors (like, say, the use of CSP in [2] or first-order logic in [89]). Instead, it provides for the very semantics of "interconnection", "configuration", "instantiation" and "composition", i.e. the principles and design mechanisms that are related to the gross modularisation of complex systems. Category Theory does this at a very abstract level because what it proposes is a toolbox that can be applied to whatever formalism is chosen for modelling the behaviour of systems as long as that formalism satisfies some structural properties. It is precisely the structural properties that make a formalism suitable for supporting architectural design that we shall make our primary focus. However, we need some concrete language in which to illustrate and motivate our approach. Not surprisingly, we will use CommUnity for that purpose.

9.2 Connectors in CommUnity

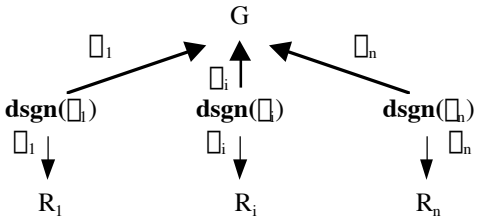
According to [2], an architectural connector (type) can be defined by a set of *roles* that can be instantiated with specific components of the system under construction, and a *glue* specification that describes how the activities of the role instances are to be coordinated. Using the mechanisms that we introduced in the previous chapter for configuration design in CommUnity, it is not difficult to come up with a formal notion of connector that has the same properties as those given in [2] for the language WRIGHT:

9.2.1 Definition – architectural connector

A *connection* consists of

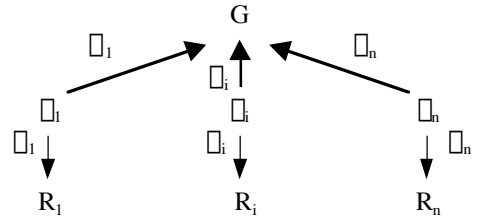
- two designs G and R, called the glue and the role of the connection, respectively;
- a signature Σ and two morphisms $\sigma:\mathbf{dsgn}(\Sigma) \to G, \rho:\mathbf{dsgn}(\Sigma) \to R$ connecting the glue and the role.

A *connector* is a finite set of connections with the same glue that, together, constitute a well-formed configuration (see 8.2.5).

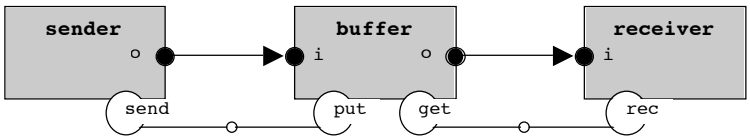


The *semantics* of a connector is the colimit of the diagram formed by its connections.

Because we are working in a coordinated category, we know already that we can adopt a simplified notation for diagrams such as these by using signatures directly in place of their discrete lifts (cables):



For instance, asynchronous communication through a bounded channel can be modelled by a connector *ASYNC* with two connections, as depicted below using the graphical notation that we have already introduced for configurations:



The glue of *ASYNC* is the bounded buffer with FIFO discipline presented in 8.1. It prevents the *sender* from sending a new message when there is no space, and prevents the *receiver* from reading a new message when there are no messages. The two roles – *sender* and *receiver* – define the behaviour required of the components to which the connector can be applied. For the *sender*, we require that no message be produced before the previous one has been processed. Its design is the one given already in section 8.1. For the *receiver*, we simply require that it have an action that models the reception of a message.

```
design receiver [t:sort] is
in     i: t
do     rec:□true,false □  skip
```

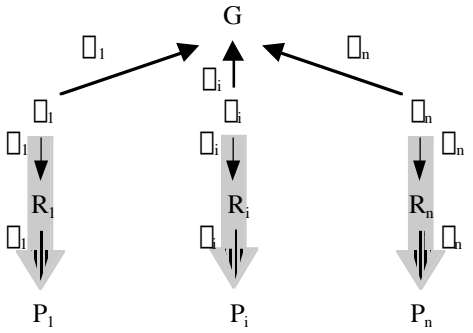
What we have described are connector *types* in the sense that they can be instantiated. More concretely, the roles of a connector type can be instantiated with specific designs. In WRIGHT [2], role instantiation has to obey a compatibility requirement expressed via the refinement relation of CSP [68]. In CommUnity, the refinement relation is formalised through the morphisms defined in 8.3.1, leading to the following notion of instantiation:

9.2.2 Definition – connector instantiation

An *instantiation* of a connection with role R consists of a design P together with a refinement morphism $\sigma : R \to P$.

An instantiation of a connector consists of an instantiation for each of its connections.

In order to define the semantics of such an instantiation, notice that, as discussed in 8.3.2, each instantiation $\sigma : R \to P$ of a connection can be composed with $\sigma : \mathbf{dsgn}(\sigma) \to R$ to define $\sigma ; \sigma : \mathbf{dsgn}(\sigma) \to \mathbf{c\text{-}sign}(P)$. Because the category of designs is coordinated over signatures (8.2.4), every such signature morphism can be lifted to a design morphism $\sigma ; \sigma : \mathbf{dsgn}(\sigma) \to P$. Hence, an instantiation of a connector defines a diagram in ***c-DSGN*** that connects the role instances to the glue.



Moreover, because each connection is according to the rules set for well-formed configurations as detailed in 8.2.5, the diagram defined by the instantiation is, indeed, a configuration and, hence, has a colimit.

9.2.3 Definition – semantics of connector instantiation

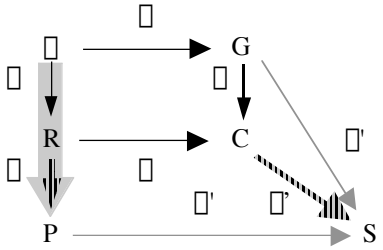
The interconnection (configuration) defined by a connector instantiation is the diagram in ***c-DSGN*** formed as described above by composing the role morphism of each connection with its instantiation.

The semantics of a connector instantiation is the colimit of the interconnection that it defines.

Because, as already argued, colimits in ***c-DSGN*** express parallel composition, this semantics agrees with the one provided in [2] for the

language WRIGHT. In the next section, we shall take this analogy with WRIGHT one step further. Moreover, the categorical formalisation makes it possible to prove that the design that results from the semantics of the instantiation is a refinement of the semantics of the connector itself.

As an example, let us consider, for simplicity, a connector with just one role.

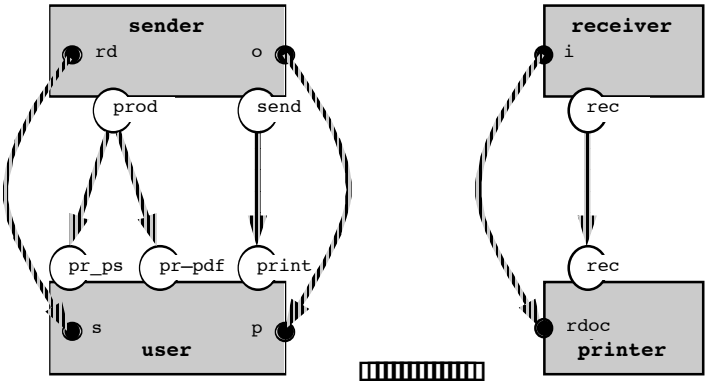


The semantics of the connector is given by the colimit of the pair $\langle \square,\square \rangle - \langle \square:R \square C, \square:G \square C \rangle$. The instantiation of the role with the component *P* through the refinement morphism $\square$ is given by the colimit of $\langle \square;\square,\square \rangle - \langle \square':P \square S, \square':G \square S \rangle$. We can easily prove that there exists a refinement morphism $\square:C \square S$, which establishes the "correctness" of the instantiation mechanism. This is because all the different objects and morphisms involved can be brought into a more general category in which the universal properties of colimits guarantee the existence of the required refinement morphism. A full proof of this property can be found in [77].

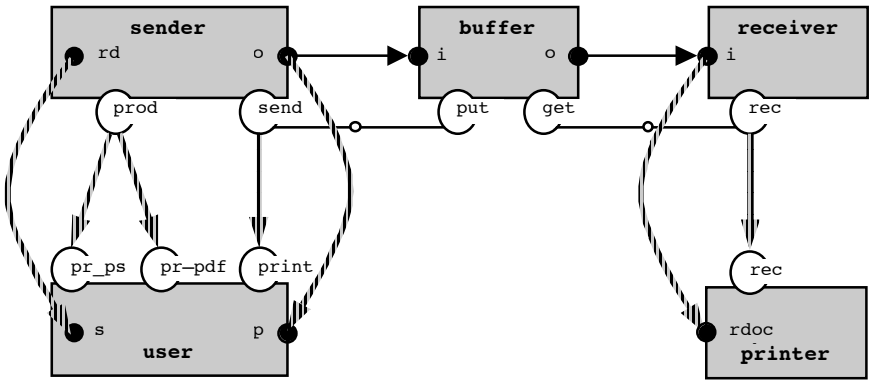
As an example, consider again the connector *ASYNC*. We have already seen in section 8.3 that *sender* is refined by the design *user*. Likewise, *printer* is a refinement of *receiver* via the refinement morphism $\square$*:receiver*$\square$ *printer* defined by

$\square_{ch}(i)=rdoc$
$\square_{ac}(rec)=rec$

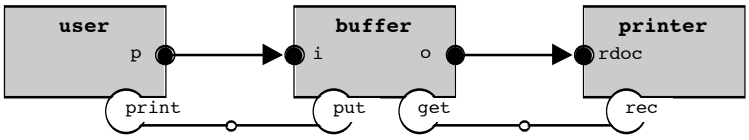
In *printer*, the reception of a message from the input channel (named *rdoc)* corresponds to downloading it into the private channel *pdoc*. This action is only enabled if the previous message has already been printed.



Therefore, we can instantiate *ASYNC* to connect the user to the printer asynchronously:

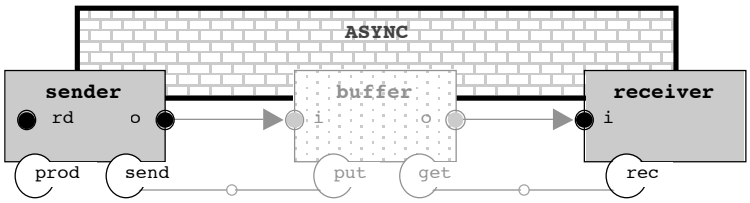


The final configuration is obtained by calculating the composition of the signature morphisms that define the two connections of *ASYNC* with the refinement morphisms σ,σ. For instance, the channel *p* of *user* gets connected to the input channel *i* of *buffer* because $\sigma(o)=p$ and *o* is connected to *i* of *buffer*. The resulting configuration is exactly the one we have already presented in section 8.2

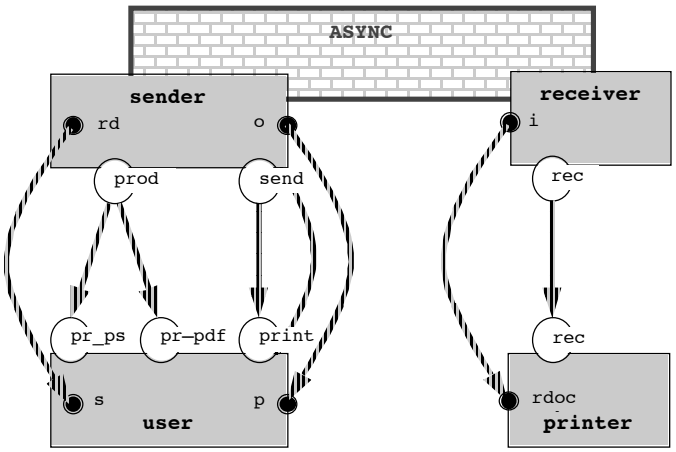


In order to simplify the notation when making use of connectors, we will "hide" the glue and its connections to the roles, leaving just the

roles visible to suggest that they provide the “interface” of the connector:



Instantiation will be denoted as follows:



Architectural connectors are used for systematising software development by offering "standard" means for interconnecting components that can be reused from one application to another. In this sense, the "typical" glue is a program that implements a well-established pattern of behaviour (e.g. a communicationm protocol) that can be superposed to existing components of a system through the instantiation of the roles of the connector.

However, architectures also fulfil an important role in supporting a high-level description of the organisation of a system by identifying its main components and the way these components are interconnected. An early identification of the architectural elements intended for a system will help to manage the subsequent design phases according to the organisation that they imply, identifying opportunities for reuse or the integration of third-party components. From this point of view, it seems useful to allow for connectors to be based on glues that are not yet fully developed as programs but for which concrete commitments

have already been made to determine the type of interconnection that they will ensure. For instance, at an early stage of development, one may decide on adopting a client-server architecture without committing to a specific protocol of communication between the client and the server. This is why, in the definition of connector in CommUnity, we left open the possibility for the glue not to be a program but a design in general.

However, in this more general framework, we have to account for the possible refinements of the glue. What happens if we refine the glue of a connector that has been instantiated to given components of a system? Is the resulting design a refinement of the more abstract design from which we started? More generally, how do connectors propagate through design, be it because the instances of the roles are refined or the glue is refined? One of the advantages of using Category Theory as a mathematical framework for formalising architectures is that answers to questions like these can be discussed at the right level of abstraction. Another advantage is that the questions themselves can be formulated in terms that are independent of any specific ADL and answered by characterising the classes of ADLs that satisfy the given properties. This is what we will do in later sections.

9.3 Examples

We now present more examples of connectors, namely some that we will need in later sections for illustrating algebraic operations on connectors. Their application will be illustrated on examples related to a case study on mobility [95]:

One or more carts move continuously in the same direction on a U-units long circular track. A cart advances one unit at each step. Along the track there are stations. There is at most one station per unit. Each station corresponds to a check-in counter or to a gate. Carts take bags from check-in stations to gate stations. All bags from a given check-in go to the same gate. A cart transports at most one bag at a time. When it is empty, the cart picks a bag up from the nearest check-in. Carts must not bump into each other. Carts also keep a count of how many laps they have done, starting at some initial location.

The program that controls a cart is

```
design cart is
in     idest: 0..U-1, ibag:int
out    obag, laps : int
prv    loc: 0..U-1, dest: -1..U-1, initloc: int
do     move: loc≠dest □
             loc:=loc+1 ‖ laps:=if(loc=initloc,laps+1,laps)
[]     get:  dest=-1 □ obag:=ibag ‖ dest:=idest
[]     put:  loc=dest □ obag:=0 ‖ dest:=-1
```

where $+_U$ is addition modulo U.

Locations are represented by integers from zero to the track length minus one. Bags are represented by integers, the absence of a bag being denoted by zero. Whenever the cart is empty, its destination is an unreachable location (–1), so that the cart keeps moving until it gets a bag and a valid gate location through action *get*. When it reaches its destination, the cart unloads the bag through action *put*. Notice that, because input channels may be changed arbitrarily by the environment, the cart must copy their values to output/private channels to make sure the correct bag is unloaded at the correct gate.

A check-in counter manages a queue of bags that it loads one by one onto passing carts.

```
design check-in is
out     bag: int, dest: 0..U–1,
prv     loc: 0..U–1, next: bool, q: list(int)
do      new: q≠[]∧next → bag:=head(q)‖q:=tail(q)‖next:=false
[]      put: ∧next → next:=true
```

Channel *next* is used to impose sequentiality among the actions. In a configuration in which a cart is loading at a gate, the *put* action must be synchronised with a cart's *get* action and channels *bag* and *dest* must be shared with *ibag* and *idest*, respectively.

A gate keeps a queue of bags and adds each new bag to the tail.

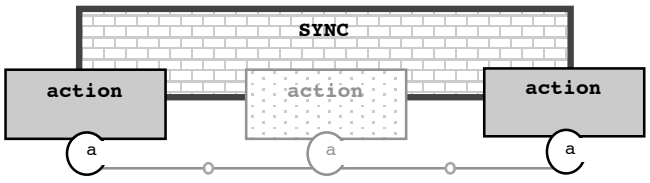
```
design gate is
in      bag: int
prv     loc: 0..U-1, q: list(int)
do      get: true → q:=q.bag
```

In a configuration in which a cart is unloading at a gate, action *get* of the gate must be synchronised with the cart's *put* action, and channel *bag* must be shared with *obag*.

9.3.1 Synchronisation

We begin with the connector that allows us to synchronise two actions of different components. A plain cable would suffice for this purpose, but it is not able to capture the general case of transient synchronisation [95]. Having already a connector for the simpler case makes the presentation more uniform.

The glue and roles of the synchronisation connector are the same:



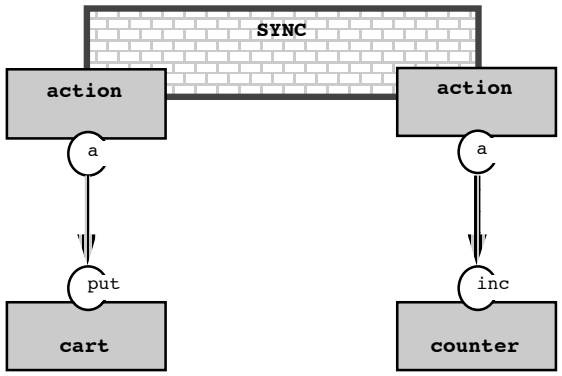
where

```
design action is
do     a: true,false □ skip
```

Notice that the action has the least deterministic specification possible: its guard is given the widest possible interval and no commitments are made on its effects. Hence, it can be refined by any action.

According to the colimit semantics of connectors, when the two roles are instantiated with particular actions a_1 and a_2 of particular components, the components have to synchronise with each other every time one of them wants to execute the corresponding action: either both execute the joint action, or none executes.

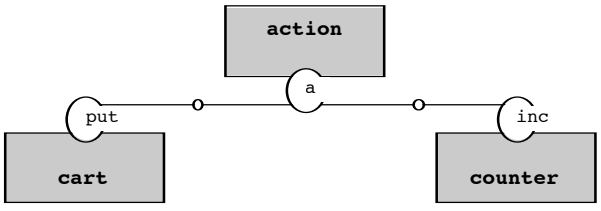
As an example of using this connector, if we wish to count how often a cart unloads, we can monitor its *put* action with a counter:



where

```
design counter is
out    c:int
do     inc: true □ c:=c+1
[]     reset: true □ c:=0
```

According to what was defined in 9.2.2, the interconnection defined by this instantiation is the following configuration:



The resulting semantics is the synchronisation of *put* and *inc*. The following program captures the joint behaviour of the interconnected components:

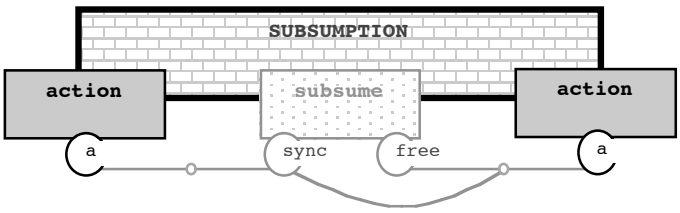
```
design monitored_cart is
in     idest: 0..U-1, ibag: int
out    obag, laps, unloads : int
prv    loc: 0..U-1, dest: -1..U-1, initloc : int
do     move: loc≠dest
            ▯ loc:=loc+1 ‖ laps:=if(loc=initloc,laps+1,laps)
[]     get: dest=-1 ▯ obag:=ibag ‖ dest:=idest
[]     put|inc: loc=dest
            ▯ obag:=0 ‖ dest:=-1 ‖ unloads:=unloads+1
```

9.3.2 Subsumption

Intuitively, synchronisation corresponds to an "equivalence" between the occurrence of two actions: the occurrence of each of the actions "implies" the occurrence of the other. In many circumstances, we are interested in one of the implications, like in remote method invocation: a call requests the execution of the method, but that same method may be invoked by other calls.

For instance, to avoid a cart colliding with a cart that is right in front of it, we only need one implication: if the first one moves, so must the one in front. The other implication is not necessary. The analogy with implication also extends to the counter-positive: if the front car cannot move, for instance because it is (un)loading a bag, then neither can the rear one.

We call this “one-way” synchronisation *action subsumption*. The subsumption connector is given by the following configuration:



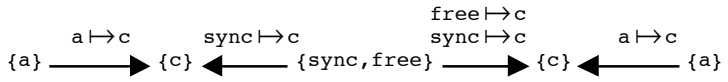
where the glue is now given by

```
design subsume is
do     sync: true,false ▯ skip
[]     free: true,false ▯ skip
```

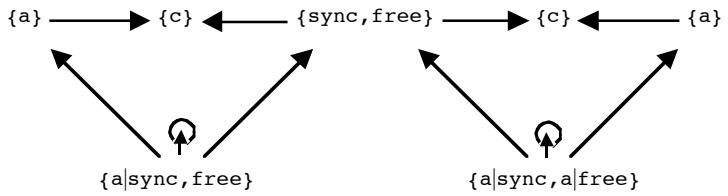
Notice that although the two roles are the same, the connector is not symmetric because the connections treat the two role actions differ-

ently: the right-hand one may be executed alone at any time, while the left-hand one must co-occur with the right-hand one, through action *sync*. Indeed, the semantics of the connector generates the following synchronisation sets: $a_1|sync|a_2$ and $free|a_2$ where a_1 is a renaming of the left-hand role action and a_2 is a renaming of the left-hand role one. Hence, action a_1 can only occur together with a_2 but that a_2 can occur without a_1.

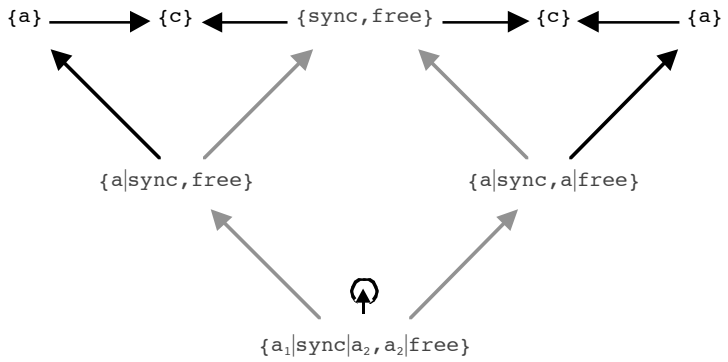
In order to understand how subsumption works, let us detail the construction of the semantics of the connector, starting with the corresponding diagram of signatures. Because only actions are involved, we will take the diagram directly over pointed sets:



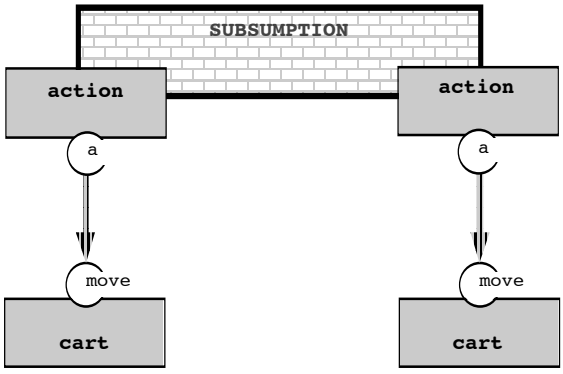
The limit of this diagram can be computed by first taking the two obvious pullbacks:



Notice that, because the interconnecting morphisms are different, different synchronisations are generated even if the nodes are the same. Finally, the middle pullback relates the pairwise synchronisations to make up the global interconnection:

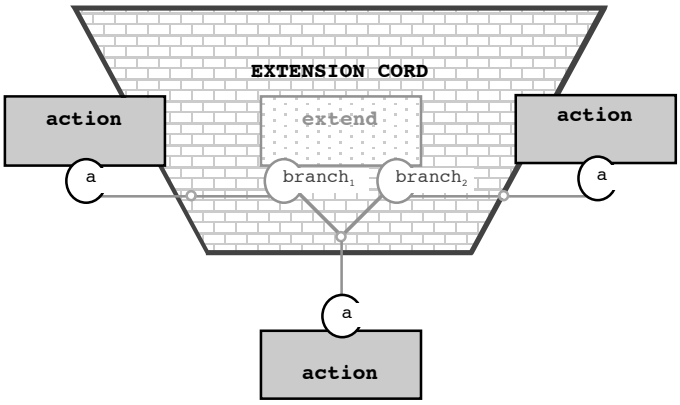


As an example of the application of this connector, consider its instantiation with the *move* actions of two carts: any movement of the cart on the left implies a movement of the cart on the right. Hence, this instantiation can be used to prevent collision when the left cart is too close behind the right cart.



9.3.3 Extension cord

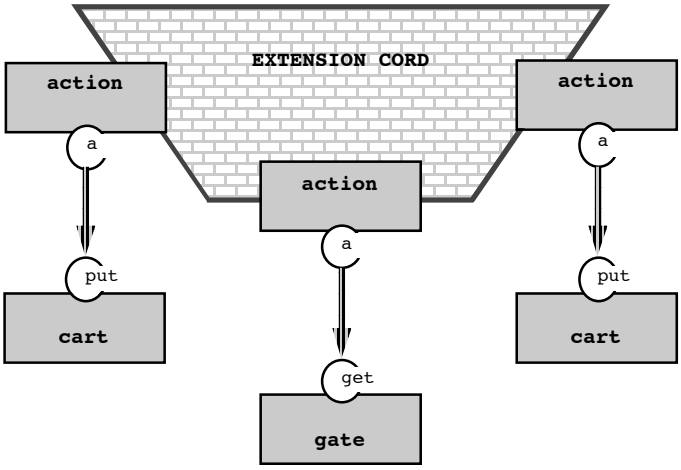
A generalisation of the subsumption and synchronisation connectors is to allow an action to synchronise, independently, with two actions of two different components, achieving an effect similar to an extension cord that one can use to connect two independent devices to the same power supply.



where

```
design extend
do      branch1: true, false ▯ skip
▯       branch2: true, false ▯ skip
```

If we instantiate the left-hand role with an action a_1 and the right-hand role with an action a_2, and the middle role with an action b, the semantics of the interconnection, as obtained through the colimit, is given by two synchronisation sets: $a_1|branch_1|b$ and $a_2|branch_2|b$. Notice that action b will always occur simultaneously with either a_1 or a_2 but not with both. For instance, a gate that can handle two carts downloading simultaneously can configured as follows:



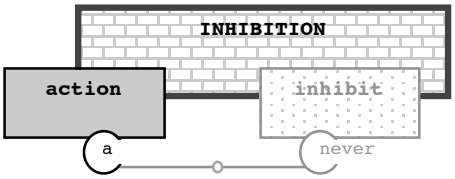
This connector can be generalised to any finite number n of ramifications (branch actions), giving rise to an *n-EXTENSION CORD*. through which a server can be connected simultaneously, but independently, to a fixed maximum number of clients.

The more attentative reader may also have noticed that the glue *extend* of *EXTENSION CORD* is the same as (isomorphic to) *subsume* of *SUBSUMPTION*. Indeed, the difference between the two connectors is in the roles that they offer; *SUBSUMPTION* does not offer the "free"-ramification of the subsumed action as a role for interconnection. We can also say that *SUBSUMPTION* is no more than a *1-EXTENSION CORD*. This relationship will be discussed in more detail in 10.1.2.

9.3.4 Inhibition

Another basic connector type is the one that allows us to inhibit an action by making its guard false. This is useful when, for some reason, we need to prevent an action from occurring but without having to reprogram the component. Indeed, the mechanism of superposition that we have used as a semantics for the application of architectural connectors allow us to disable an action without changing the guard directly but by

just inducing this effect: it suffices to synchronise the action with one that has a false guard.



The glue is:

```
design inhibit is
do     never: false □ skip
```

When the role is instantiated with an action with guard B, the result of the interconnection is the same action guarded by $B \wedge \mathit{false}$.

This connector can be generalised to arbitrary conditions with which one may strengthen the guards of given actions. The inhibitor just has to be provided with the data that is necessary to compute the condition C that will strengthen the guard, for instance through the use of input channels through which we can select the sources of the information that will disable the action.

```
design inhibit(C) is
in     ...
do     never: C □ skip
```

The result of instantiating the role with an action with guard B is the same action guarded by $B \wedge C$.

9.4 An ADL-independent notion of connector

The notion of connector that we presented in 9.2 can be generalised to design formalisms other than CommUnity. In this section, we shall discuss the properties that such formalisms need to satisfy for supporting the architectural concepts and mechanisms that we have illustrated for CommUnity.

Before embarking on this discussion, we need to fix a framework in which designs, configurations and relationships between designs, such as refinement, can be formally described.

9.4.1 Definition – design formalism

A formalism supporting system design consists of:

- a category ***c-DESC*** of component descriptions in which systems of interconnected components are modelled through diagrams;

- for every set *CD* of component descriptions, a set *Conf(CD)* consisting of all well-formed configurations that can be built from the components in *CD*. Each such configuration is a diagram in ***c-DESC*** that is guaranteed to have a colimit. Typically, *Conf* is given through a set of rules that govern the interconnection of components in the formalism.
- a category ***r-DESC*** with the same objects as ***c-DESC***, but in which morphisms model refinement, i.e. a morphism $\sigma: S \to S'$ in ***r-DESC*** expresses that *S'* refines *S*, identifying the design decisions that lead from *S* to *S'*. Because the description of a composite system is given by a colimit of a diagram in ***c-DESC*** and, hence, is defined up to an isomorphism in ***c-DESC***, refinement morphisms must be such that descriptions that are isomorphic in ***c-DESC*** refine, and are refined exactly by, the same descriptions.

Summarising, all that we require is a notion of system description, a relationship between descriptions that captures components of systems, another relationship that captures refinement, and criteria for determining when a diagram of interconnected components is a well-formed configuration.

In the context of this categorical framework, we shall now discuss the properties that are necessary for supporting Software Architectures. A key property for supporting architectural design is a clear separation between the description of individual components and their interaction in the overall system organisation. In other words, the formalism must support the separation between what, in the description of a system, is responsible for its computational aspects and what is concerned with coordinating the interaction between its different components.

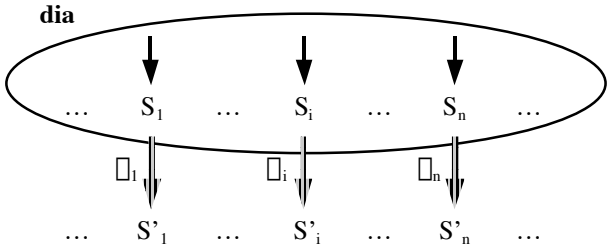
In the case of CommUnity, as we have seen, only signatures are involved in interconnections. The body of a component design describes its functionality and, hence, corresponds to the computational part of the design. At the more general level that we are discussing, we shall take the separation between coordination and computation to be materialised through a functor ***sign: c-DESC*** $\to$ ***SIGN*** mapping descriptions to signatures, forgetting their computational aspects. The fact that the computational side does not play any role in the interconnection of systems can be captured by requiring ***sign*** to be coordinated in the sense of 7.5.4.

Another crucial property for supporting architectural design is in the interplay between structuring systems in architectural terms and refinement. We have already pointed out that one of the goals of Software Architectures is to support a view of the gross organisation of systems in terms of components and their interconnections that can be carried through the refinement steps that eventually lead to the implementation of all its components. Hence, it is necessary that the application of ar-

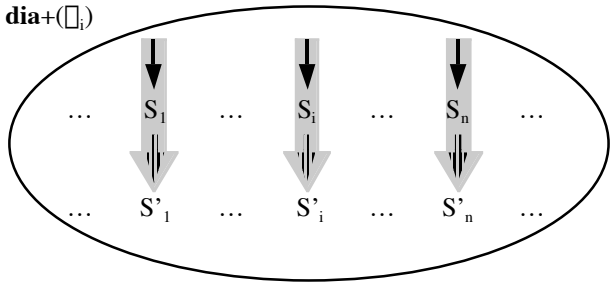
chitectural connectors to abstract designs, as a means of making early decisions on the way certain components need to be coordinated, will not be jeopardised by subsequent refinements of the component designs towards their final implementations. Likewise, it is desirable that the application of a connector may be made on the basis of an abstract design of its glue as a means of determining main aspects of the required coordination without committing to the final mechanisms that will bring about that coordination.

One of the advantages of the categorical framework that we have been proposing is that it makes the formulation of these properties relatively easy, leading to a characterisation of the design formalisms that support them in terms of the structural properties that we have been discussing. For instance, we have already seen in section 0 that, in the situations in which refinement morphisms map directly to signature morphisms, we can simply put together, in a diagram of signatures, the morphisms that define the interactions and the morphisms that establish the refinement of the component descriptions.

More precisely, in the situations in which there exists a forgetful functor ***r-sign: r-DESC*** $\rightarrow$ ***SIGN*** that agrees with the coordination functor ***sign*** on signatures – i.e. ***r-sign****(S)*=***sign****(S)* for every *S*:***c-DESC*** – given a well-formed configuration diagram ***dia*** of a system with components $S_1,...,S_n$ and refinement morphisms $\sigma_i: S_i \rightarrow S'_i: i \in 1..j$,



we can obtain a new diagram in ***c-DESC*** and, hence, a new configuration, by composing the morphisms ***r-sign****(*σ_i*)* with those in ***dia*** that originate in channels (signatures) and have the S_i as targets.

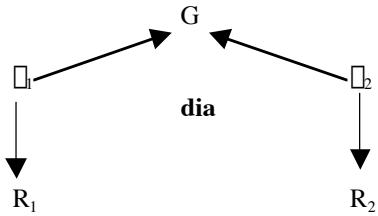


In general, it may be possible to propagate the interactions between the components of a system when their descriptions are replaced by more concrete ones even when refinement morphisms do not map to signature morphisms. This more general situation can be characterised as follows:

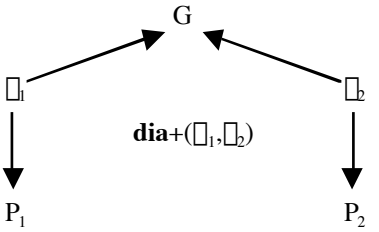
9.4.2 Definition – compositional design formalism

We say that a design formalism is *compositional* whenever, for every well-formed configuration ***dia*** involving descriptions $\{S_1,...,S_n\}$ and refinements morphisms $\{\sigma_i: S_i \rightarrow S'_i : i \in 1..n\}$, there is a well-formed configuration diagram ***dia***$+(\sigma_i)$ that characterises the system obtained by replacing the S_i by their refinements and satisfies the following correctness criterion: the colimit of ***dia***$+(\sigma_i)$ provides a refinement for the colimit of ***dia***.

When we consider the specific case of the configurations obtained by direct instantiation of an architectural connector, this property reflects the compositionality of the connector as an operation on configurations. Compositionality ensures that the semantics of the connector is preserved (refined) by any system that results from its instantiation. For instance, in the case of a binary connector



and given instantiations $\sigma_1: R_1 \rightarrow P_1$ and $\sigma_2: R_2 \rightarrow P_2$, compositionality means that the description returned by the colimit of ***dia*** is refined by the description returned by the colimit of ***dia***$+(\sigma_1,\sigma_2)$.



Likewise, compositionality guarantees that if a connector with an abstract glue G is applied to given designs, and the glue is later on refined through a morphism $\sigma: G \rightarrow G'$, the description that is obtained through the colimit of ***dia***$+\sigma$ is a refinement of the semantics of the original in-

stantiation. In fact, we can consider the refinement of the glue to be a special case of an operation on the connector that delivers another connector – a refinement of the original one in the case at hand.

9.4.3 Definition – architectural school

A design formalism $F=<$***c-DESC,Conf,r-DESC***$>$ supports architectural design, and is called an *architectural school,* iff

- ***c-DESC*** is coordinated over a category ***SIGN*** through a functor ***sign: c-DESC*** $\rightarrow$ ***SIGN***
- F is compositional

9.5 Adding abstraction to connectors

The mathematical framework that we presented in the previous sections provides not only an ADL-independent semantics for the principles and techniques that can be found in existing approaches to Software Architectures, but also a basis for extending the capabilities of existing ADLs. Until the end of this chapter, we will present and explore some of the avenues that this mathematical characterisation has opened, hoping that the reader will want to explore them even further, or find new ones!

As already mentioned, the purpose of the roles in a connector is to impose restrictions on the local behaviour of the components that are admissible as instances. In the approach to architectural design outlined in the previous sections, this is achieved through the notion of correct instantiation via refinement morphisms. As also seen above, roles do not play any part in the calculation of the resulting system. They are used only for defining what a correct instantiation is. This separation of concerns justifies the adoption of a more declarative formalism for the specification of roles, namely one in which it is easier to formulate the properties required of components to be admissible instances.

In this section, we are going to place ourselves in the situation in which the glues are designs, the roles are specifications, and the instantiations of the roles are, again, designs. We are going to consider that specifications are given as a category ***SPEC***, e.g. the category of theories of a logic formalised as an institution – see 6.5.11 above. We take the relationship between specifications and designs to be captured through the following elements:

- a functor ***spec:SIGN*** $\rightarrow$ ***SPEC*** mapping signatures and their morphisms to specifications.
- The idea behind the functor ***spec*** is that, just like, through ***c-desc*** (the left adjoint of ***sign***) signatures provide the means for interconnecting

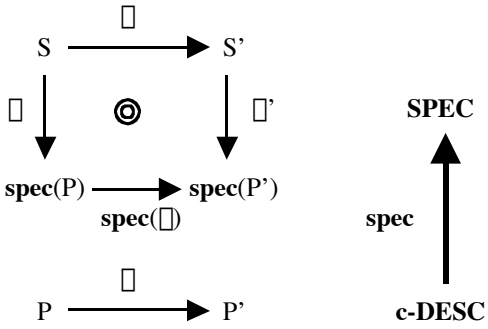
designs, they should also provide means for interconnecting specifications. Hence, every signature generates a canonical specification – the specification of a cable. However, it is not necessary for ***spec*** to satisfy as many structural properties as ***c-desc*** because, for the purposes of this section, we are limiting the use of specifications to the definition of connector roles. Naturally, if we wish to address architecture building at the specification level, then we will have to require ***SPEC*** to satisfy the properties that we discussed in section 9.4.

- a satisfaction relation $\models$ between design morphisms and specification morphisms satisfying the following properties:
  1. If $\sigma:P \to P' \models \sigma:S \to S'$, then $id_P \models id_S$ and $id_{P'} \models id_{S'}$.
  2. If $\sigma_1:P_1 \to P_2 \models \sigma_1:S_1 \to S_2$ and $\sigma_2:P_2 \to P_3 \models \sigma_2:S_2 \to S_3$ then $\sigma_1;\sigma_2 \models \sigma_1;\sigma_2$
  3. Let $\mathbf{s}:I \to \mathbf{SPEC}$ be a diagram of specifications and $\mathbf{p}:I \to \mathbf{c\text{-}DESC}$ a diagram of designs with the same shape such that, for every edge $f:i \to j$ in I, $\mathbf{p}_f:\mathbf{p}_i \to \mathbf{p}_j \models \mathbf{s}_f:\mathbf{s}_i \to \mathbf{s}_j$. We require that, if $\mathbf{p}$ admits a colimit $\sigma_i:\mathbf{p}_i \to P$, then $\mathbf{s}$ admits a colimit $\sigma_i:\mathbf{s}_i \to S$ such that, for every node $i:I$, $\sigma_i \models \sigma_i$.
  4. If $id_P \models id_S$ and $\sigma:P \to P'$ is a refinement morphism, then $id_{P'} \models id_S$
  5. For every signature Σ, $id_{\mathbf{c\text{-}desc}(\Sigma)} \models id_{\mathbf{spec}(\Sigma)}$

The satisfaction relation is defined directly on morphisms because our ultimate goal is to address interconnections, not just components. Satisfaction of component specifications by designs is given through the identity morphisms. The properties required of the satisfaction relation address its compatibility with the categorical constructions that we use, namely composition of morphisms and colimits. The last two properties mean that refinement of component designs leaves the satisfaction relation invariant, and that the design (cable) generated by every signature satisfies the specification (cable) generated by the same signature.

We have already seen in 5.1.3 that a satisfaction relation between specifications and programs can be defined if we are able to establish a functor $\mathbf{r\text{-}DESC} \to \mathbf{SPEC}$ that maps every design to the maximum set of properties that it "satisfies". Intuitively, this functor should be an extension of ***spec*** in the sense that, for every signature Σ, $\mathbf{spec}(\Sigma)$ should be isomorphic to $\mathbf{spec}(\mathbf{c\text{-}desc}(\Sigma))$. Therefore, we will call it ***spec***. In this case, it makes sense to define

$$\sigma:P \to P' \models \sigma:S \to S' \text{ iff}$$
there exist $\sigma:S \to \mathbf{spec}(P)$ and $\sigma':S' \to \mathbf{spec}(P')$ s.t. $\sigma;\mathbf{spec}(\sigma)=\sigma;\sigma'$



Notice that we get $id_P \models id_S$ iff there exists $\sigma:S \to \mathbf{spec}(P)$, which is exactly how satisfaction was defined in 5.1.3. Moreover, properties 1-4 are obtained free from the fact that ***spec*** is a functor. Condition 5 is satisfied because $\mathbf{spec}(\sigma)$ has been required to be isomorphic to $\mathbf{spec}(\mathbf{c\text{-}desc}(\sigma))$.

Given such a setting, we generalise the notion of connector as follows:

9.5.1 Definition – (generalised) architectural connector

A (generalised) connection consists of

- a design G and a specification R, called the glue and the role of the connection, respectively;
- a signature Σ and two morphisms $\sigma:\mathbf{c\text{-}desc}(\Sigma) \to G$, $\rho:\mathbf{spec}(\Sigma) \to R$ in ***c-DESC*** and ***SPEC***, respectively, connecting the glue and the role via the signature (cable).

A *(generalised) connector* is a finite set of connections with the same glue.

An instantiation of a connection with signature Σ and role morphism ρ consists of a design P and a design morphism $\sigma:\mathbf{c\text{-}desc}(\Sigma) \to P$ such that $\rho \models \sigma$.

An instantiation of a connector consists of an instantiation for each of its connections. An instantiation is said to be correct if the diagram defined by the instantiation morphisms and the glue morphisms is a well-formed configuration. The colimit of this configuration defines the semantics of the instantiation, guaranteed to exist if the instantiation is correct.

Although the generalisation seems to be quite straightforward, we do not have an immediate generalisation for the semantics of connectors. This is because the glue is a design and the role is a specification, which means that a connector does not provide us with a diagram like in the homogeneous case that we studied in section 9.2. However, if we are provided with a specification for the glue, we can provide semantics for the connector at the specification level:

9.5.2 Definition – complete architectural connector

A complete connection consists of

- a design G and a specification R, called the glue and the role of the connection, respectively;
- a signature Σ and two morphisms σ:***c-desc***(Σ)$\to$ G,ρ:***spec***(Σ)$\to$ R in ***c-DESC*** and ***SPEC***, respectively, connecting the glue and the role via the signature (cable);
- a specification S and a morphism τ:***spec***(Γ)$\to$ S such that $\Gamma\models\Sigma$. Notice that this means that the design G satisfies the specification S.

A *complete connector* is a finite set of complete connections with the same glue design and specification. Its semantics is given by the colimit, if it exists, of the ***SPEC***-diagram defined by the Σ_i and the Γ.

Notice that we obtain the following property:

9.5.3 Proposition

The semantics of the instantiation of a complete connector satisfies the semantics of the connector.

An illustration of an abstract architectural school can be given in terms of a first-order extension[12] of the linear temporal logic that we studied in 3.5. As an example, we present below the specifications of a typical sender and receiver of messages through a pipe.

```
specification  pipe_sender[t:sort] is
signature      o:t, eof:bool, send
axioms         eof    G(□send ⇒ eof)

specification  pipe_receiver[t:sort] is
signature      cl, eof:bool, rec
axioms         cl    G(□rec ⇒ cl)
                     ((eof   Geof) ∧ (eof ⇒ □cl))   (□recUcl)
```

The specification *pipe_sender* accounts, through *send*, for the transmission of data of sort t through a channel o. The end of data transmission is signalled through channel *eof*: the axiom requires that *eof* be stable (remains true once it becomes true) and transmission of messages to cease once *eof* becomes true.

The specification *pipe_receiver* accounts, through *rec*, for the reception of data through the channel i. The other means of interaction with the environment is concerned with the closure of communication. Through channel *eof*, a Boolean can be received that indicates if transmission along i has ceased. Closure of communication is signalled in the channel *cl*. The first axiom requires that *cl* be stable and the recep-

[12] The extension is straightforward: the reader is encouraged to formalise it as an exercise, for which [38,39,66] can be consulted.

tion of messages to cease once *cl* becomes true. The second axiom expresses that, if the information received through *eof* is stable, the receiver is obliged to close the communication as soon as it is informed that there will be no more data. However, the receiver may decide to close the communication before that.

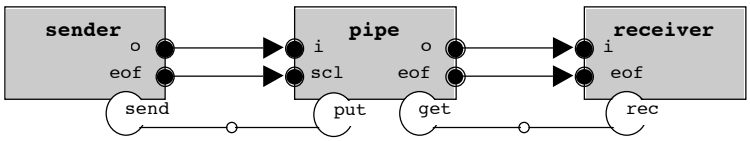
It remains to capture the relationship between specifications and designs in CommUnity.

First we notice that every design signature Σ can be mapped into a temporal signature by forgetting the different classes of channels and actions, as well as the write frames of actions. Then, we notice that part of the semantics of CommUnity designs can be encoded in LTL:

- for every action g, the negation of $L(g)$ is a blocking condition for its occurrence $(g \Box L(g))$;
- for every local channel v, $D(v)$ consists of the set of actions that can modify it – $\bigwedge_{g \in D(v)} g \quad (\mathbf{X}v=v))$
- for every action g, the condition $R(g)$ holds in every state in which g is executed – $.(g \quad \Sigma R(g))$ where Σ is a translation that replaces any primed variable v' by the term $(\mathbf{X}v)$;
- private actions that are infinitely often enabled are guaranteed to be selected infinitely often – $(\mathbf{GF}U(g) \quad \mathbf{GF}g)$

This encoding extends to refinement morphisms, establishing a functor $\textbf{\textit{r-DESC}} \Sigma \textbf{\textit{THEO}}_{FOLTL}$, from which we can define a satisfaction relation. Notice that the fact that refinement of CommUnity designs is contravariant on the upper bound of actions is crucial: refinement morphisms are liveness preserving but interconnection morphisms are not!

In order to illustrate the generalised notion of connector in this setting, we present below the connector *cpipe*,



where *pipe* is:

```
design pipe [t:sort, bound:t] is
in     i:t, scl:bool
out    o:t, eof:bool
prv    rd: bool, b: list(t)
do     put: true → b:=b.i
[] prv next: |b|>0 ∧ rd → o:=head(b) ‖ b:=tail(b) ‖ rd:=true
[]     get: rd → rd:=false
[] prv sig: scl ∧ |b|=0 → eof:=true
```

This design, which is the glue of the connector, models a buffer with unlimited capacity and a FIFO discipline. It signals the end of data to

the consumer of messages as soon as the buffer gets empty and the sender of messages has already informed, through the input channel *scl*, that it will not send anymore messages.

The two roles – the specifications *sender* and *receive*r introduced before – define the behaviour required of the components to which the connector *cpipe* can be applied. It is interesting to notice that, due to the fact that LTL is more abstract than CommUnity, we were able to abstract away completely the production of messages in the role *sender*. In the design of *sender* presented in 8.1 we had to consider an action modelling the production of messages.

We can now generalise these constructions even further by letting the connections use different specification formalisms and the instantiations to be performed over components designed in different design formalisms. In this way, it will be possible to support the reuse of third-party components, namely legacy systems, as well as the integration of non-software components in systems, thus highlighting the role of architectures in promoting a structured and incremental approach to system construction and evolution.

However, to be able to make sense of the interconnections, we have to admit that all the design formalisms are coordinated over the same category of signatures. That is to say, we assume that the integration of heterogeneous components is made at the level of the coordination mechanisms, independently of the way each component brings about its computations. Hence, we will assume given

- a family $\{\boldsymbol{DSGN}_d\}_{d:D}$ of categories of designs, all of which are coordinated over the same category ***SIGN*** via a family of functors $\{\boldsymbol{dsgn}_d\}_{d:D}$,
- a family $\{\boldsymbol{SPEC}_c\}_{c:C}$ of categories of specifications together with a family $\{\boldsymbol{spec}_c{:}\boldsymbol{SPEC}_c \to \boldsymbol{SIGN}\}_{c:C}$ of functors,
- a family $\{\models_s\}_{s:S}$ of satisfaction relations, each of which relates a design formalism *d(s)* and a specification formalism *c(s)*. We do not require *S* to be the cartesian product $D \times C$, i.e. there may be pairs of design and specification formalisms for which no satisfaction relation is provided.

Given such a setting, we generalise the notion of connector as follows:

9.5.4 Definition – (heterogeneous) architectural connector

A *heterogeneous* connection consists of

- a design formalism $\boldsymbol{DSGN}_d$ and a specification formalism $\boldsymbol{SPEC}_c$;
- a design $G{:}\boldsymbol{DSGN}_d$ and a specification $R{:}\boldsymbol{SPEC}_c$, called the glue and the role of the connection, respectively;

- a signature Σ:***SIGN*** and two morphisms γ:***dsgn***$_d(\Sigma) \to G$, ρ:***spec***$_c(\Sigma) \to R$ in ***DSGN***$_d$ and ***SPEC***, respectively, connecting the glue and the role via the signature (cable).

An heterogeneous connector is a finite set of connections with the same glue.

An instantiation of a heterogeneous connection with specification formalism ***SPEC***$_c$, signature Σ and role morphism ρ consists of

- a design formalism ***DSGN***'$_{d'}$ and a satisfaction relation $\models_{<d',c>}$ between ***DSGN***'$_{d'}$ and ***SPEC***$_c$ such that $<d',c> \in S$;
- a design P and a design morphism γ:***dsgn***$_{d'}(\Sigma) \to P$ such that $\rho \models_s \gamma$.

An instantiation of a connector consists of an instantiation for each of its connections.

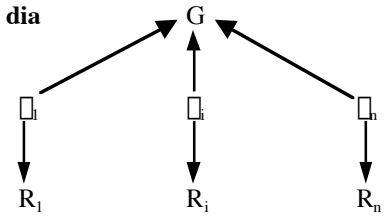
Given that in the context of heterogeneus connectors we have to deal with several design formalisms, providing a semantics for the resulting configurations requires a homogeneous formalism to which all the design formalisms can be mapped. Clearly, because we want the heterogeneity of formalisms to be carried through to the implementations in order to be able to support the integration of legacy code, third-party components and even non-software components, this common formalism cannot be at the same level as that of designs, and the mapping cannot be a simple translation. What seems to make more sense is to choose a behaviour model that can be used to provide a common semantics to all the design formalisms so that the integration is not performed at the level of the descriptions but of the behaviours that are generated from the descriptions. Indeed, when we talk about the integration of heterogeneous components, our goal is to coordinate their individual behaviours. An architecture should provide precisely the mechanisms through which this coordination is effected.

10 An algebra of connectors

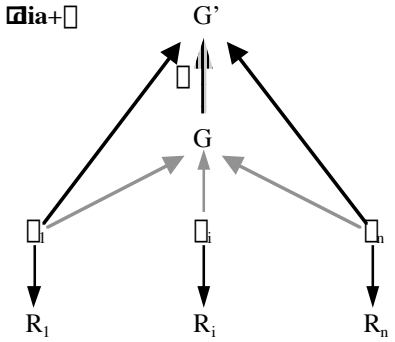
As argued in [86], the level of support that Architectural Description Languages (ADLs) provide for connector building is still far from the one awarded to components. For instance, although considerable amounts of work can be found on several aspects of connectors [2,14,86], further steps are still necessary to achieve a systematic way of constructing new connectors from existing ones. Yet, the ability to manipulate connectors in a systematic and controlled way is essential for promoting reuse and incremental development, and to make it easier to address complex interactions. At an architecture level of design, component interactions can be very simple (for instance a shared variable), but they can be very complex as well (e.g., database-accessing and networking protocols). Hence, it is very important that we have mechanisms for designing connectors in an incremental and compositional way, as well as principled ways of extending existing ones, promoting reuse. This is especially important for connectors that are used at lower levels of design because it is well known that the implementation of complex protocols is a very difficult and error prone part of system development.

It is not always possible to adapt components to work with existing connectors. Even in those cases where it is feasible, a better alternative may be to modify the connectors because, typically, there are fewer connector types than components types. Moreover, most ADLs either provide a fixed set of connectors or only allow the creation of new ones from scratch, hence requiring from the designer a deep knowledge of the particular formalism and tools at hand. Conceptually, operations on connectors allow one to factor out common properties for reuse and to better understand the relationships between different connector types.

The notation and semantics of such connector operators are, of course, among the main issues to be dealt with. Our purpose in this section is to show how typical operators can be given an ADL-independent semantics by formalising them in the categorical framework that we presented in the previous chapter. For instance, given a connector expressed through a configuration diagram ***dia***



and a refinement $\varphi: G \rightarrow G'$ of its glue, we can construct through $dia+\varphi$ another connector that has the same roles as the original one, but whose glue is now G'.



A fundamental property of this construction, given by compositionality, is that the semantics of the original connector, as expressed by the colimit of its diagram, is preserved in the sense that it is refined by the semantics of the new connector. This means that all instantiations of the new connector are refinements of instantiations of the old one. This operation supports the definition of connectors at higher levels of abstraction by delaying decisions on concrete representations of the coordination mechanisms that they offer, thus providing for the definition of specialisation hierarchies of connector types.

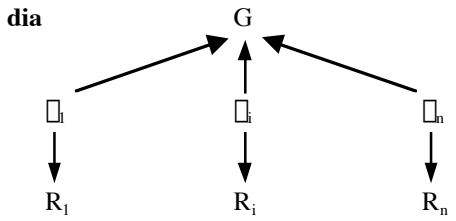
In this chapter we present three connector transformations that operate on the roles rather than the glue. Transformations that, like above, operate at the level of the glue are more sensitive in that they interfere more directly with the semantics of the connector to which they are applied. Hence, they should be restricted to engineers who have the power, and ensuing responsabilities, to change the way connectors are implemented. Operations on the roles are less critical and can be performed more liberally by users who have no access to the implementation (glue). In the last section, we present higher-order mechanisms that can be applied to connectors to obtain other connectors. Throughout the chapter, we will be working over a fixed architectural school $F=<$***c-DESC,Conf,r-DESC***$>$ (see 9.4.3).

10.1 Three operations on connectors

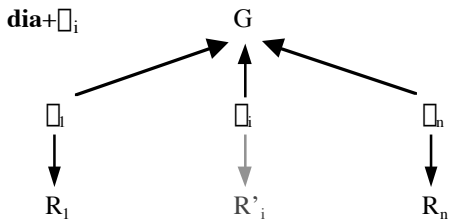
10.1.1 Role Refinement

To tailor general-purpose connectors for a specific application, it is necessary to replace the generic roles by specialised ones that can effectively act as "formal parameters" for the application at hand. Role replacement is done in the same way as applying a connector to components: there must be a refinement morphism from the generic role to the specialised one. The old role is cancelled, and the new role morphism is the composition of the old one with the refinement morphism in the sense discussed in section 3.

Given an n-ary connector



and a refinement morphism $\square_i: R_i \square R'_i$ for some $1 \le i \le n$, the role refinement operation yields the connector



Notice that this operation can be applied to both abstract and heterogeneous connectors.

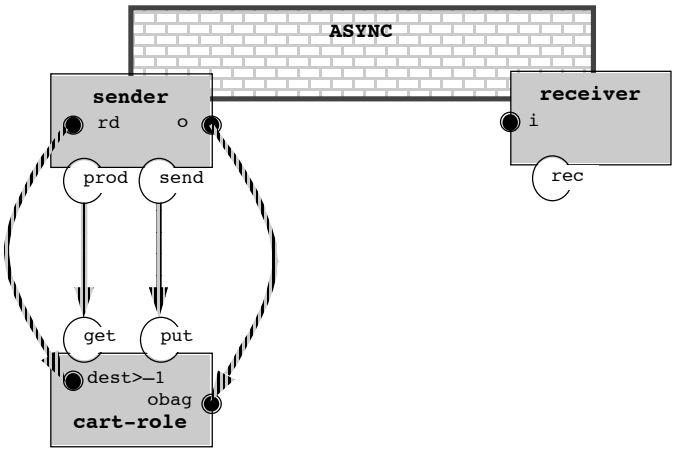
This operation preserves the semantics of the connectors to which it is applied in the sense that any instantiation of the refined connector is also an instantiation of the refined one. This is because the refinement morphism that instantiates R'_i can be composed with $\square_i$ to yield an instantiation of R_i.

As an example of role refinement, consider the asynchronous connector shown previously. This connector is too general for our luggage distribution service because the sender and receiver roles do not impose

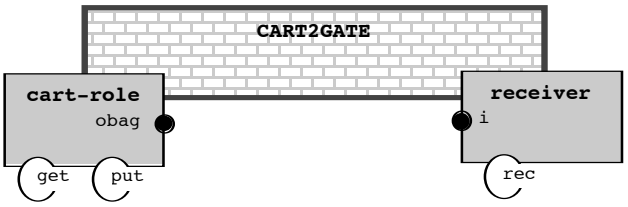
any constraints on the admissible instances. We would like to refine these roles in order to prevent meaningless applications to our example like sending the location of the check-in as a bag to the cart. This can be done through the refinement of *sender* with

```
design cart-role is
out    obag: int
prv    dest: -1..U–1
do     get[obag]: dest=-1 □ dest'>-1
[]     put: dest>-1,false □ obag:=0 ‖ dest:=-1
```

where the channel *rd* of *sender* is refined by the term *dest>–1*. Notice that this is a generalised refinement as observed after 8.3.1 in the sense that we are using a term of the target language to refine a channel. See [37] for details on this extension.



The resulting connector is

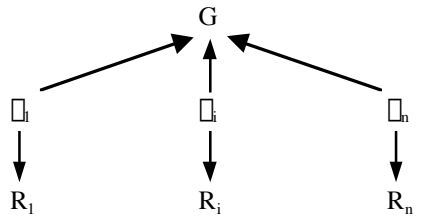


Notice that the invalid combinations are not possible because *cart-role* cannot be refined with a gate or a check-in. Moreover, the *obag* channel of *cart-role* cannot be refined by channel *laps* of cart.

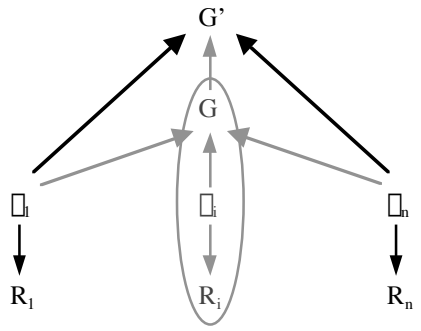
10.1.2 Role Encapsulation

To prevent a role from being further refined, the second operation we consider, when executed repeatedly, decreases the arity of a connector by encapsulating some of its roles, making the result part of the glue. This operation requires that the glue and the encapsulated role be in the same formalism.

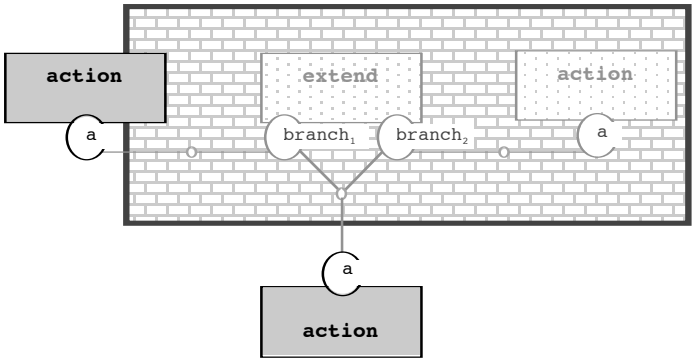
Given an n-ary connector



the encapsulation of the i-th role is performed as follows: the pushout of the i-th connection is calculated, and the other connections are changed by composing the morphisms that connect the channels to the glue with the morphism that connects the glue with the apex of the pushout, yielding a connector of arity *n–1*.



For instance, we can obtain *SUBSUMPTION* (9.3.2) from *EXTENSION CORD* (9.3.3) through encapsulation of the right-hand side *action* role. Indeed, as observed in 9.3.3, both connectors have isomorphic glues: the difference between them is in the roles that they offer. On the other hand, the pushout of the glue-role connection that is involved in the encapsulation returns, up to isomorphism, that same glue because the role does not require any properties of the instances; it just performs name bindings.

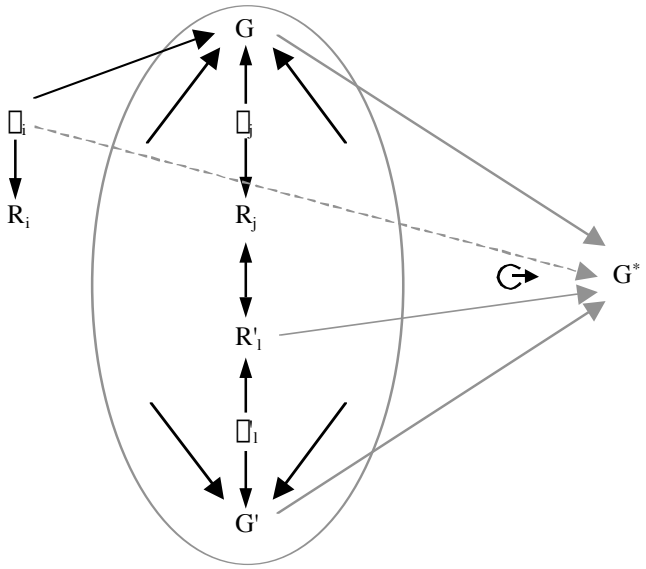


10.1.3 Role Overlay

The third operation allows combining several connectors into a single one if they have some roles in common, i.e., if there is an isomorphism between those roles. The construction is as follows.

Consider a connector with roles $\{R_i\}_{1\le i\le n}$ and glue G, and a connector with roles $\{R'_k\}_{1\le k\le m}$ and glue G' such that, R_j and R'_l are isomorphic. We construct a new connector by overlaying the isomorphic roles.

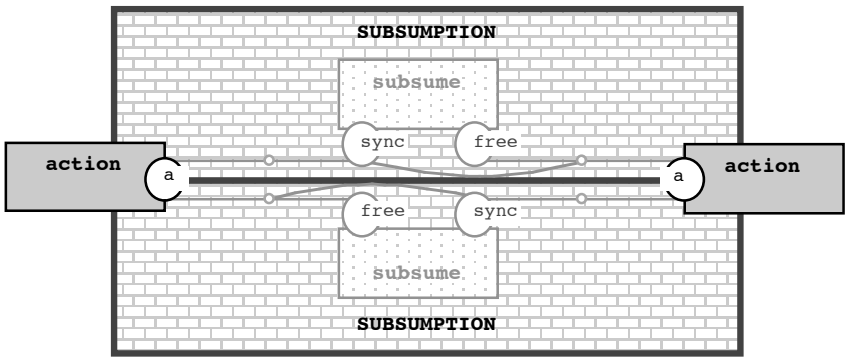
The glue of the new connector is calculated as follows. Consider the diagram that consists of all the connections that have isomorphic roles together with the isomorphisms, and calculate its colimit. The apex of the colimit is the new glue.



Each of the pairs of connections involved gives rise to a connection of the new connector. The role of this connection is one of the roles of the old connections; because they are isomorphic, it does not matter which one is chosen (in the figure, we chose R'_l). This role is connected directly to the new glue through one of the morphisms that results from the colimit; hence, its channel is the role signature.

Each of the connections of the original connectors that is not involved in the calculation of the new glue, which means that it does not share its role with a connection of the other connector, becomes a connection of the new connector by composing the old new glue morphism with the colimit morphism that connects the old glue to the new one. This is exemplified in the figure with the connection with role R_i.

This operation provides a way of building *SYNC* from *SUBSUMPTION*. Indeed, we can obtain full synchronisation of two actions by making each subsume the other. This is achieved by overlaying two copies of *SUBSUMPTION* in a symmetric way:



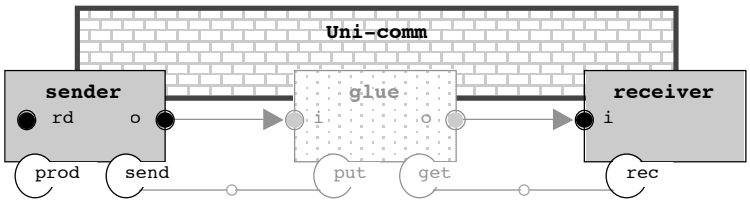
Notice that the colimit of the undeelying diagram makes all actions collapse into a single one. Hence the glue of the resulting connector is isomorphic to the 'sync' component. Moreover, each pair of overlaid roles results into a single one, and therefore there will be only two 'action' roles. In summary, the resulting connector is *SYNC*.

10.2 Higher-order connectors

As explained before, it is important to have principled forms of adapting connectors to new situations, for instance in order to incorporate compression, fault-tolerance, security, monitoring, etc. Let us consider *compression* as an example. The goal is to be able to adapt any connector that represents a communication protocol in order to compress

data for transmission in a transparent way. In order to give a first-class description of this form of adaptation, the kind of communication protocol modelled by the adapted connector needs to be made more precise. We shall describe the *compression* adaptation mechanism only for connectors that model unidirectional communication protocols.

A generic unidirectional communication protocol can be modelled by the binary connector *Uni-comm[s]*



where

```
design glue[s] is
in     i:s
out    o:s
do     put: true,false □ skip
[] prv prod[o]: true,false □ skip
[]     get: true,false □ skip
```

and *sender[s]* and *receiver[s]* are defined as before (see sections 8.1 and 9.2, respectively). Notice that this glue leaves the way in which messages are processed and transmitted completely unspecified.

Our aim is to install a compression/decompression service over *Uni-comm*. That is to say, our aim is to apply an operator to *Uni-comm* such that, in the resulting connector, a message sent by the sender is compressed before it is transmitted through *Uni-comm* and then decompressed before it is delivered to the receiver. We shall see that such an operator can be described by a higher-order connector where the compression and decompression algorithms are taken as parameters as captured by the following algebraic specification:

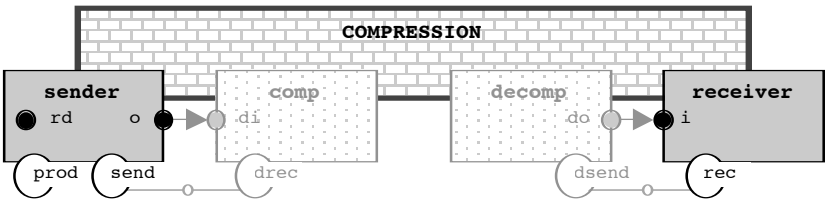
```
spec   □^cd is □^nat +
sorts  s,t
ops    comp:t->s
       decomp:s->t
       size_s:s->nat
       size_t:t->nat
axioms decomp(comp(x))=x, for any x:t
       size_s(comp(x)) ≤ size_t(x), for any x:t
```

Sorts *t* and *s* represent the types of original and compressed messages, respectively. The operation *comp* represents the process of compression of a single message, and *decomp* the inverse process of decompression. The size of the compressed message is required not to be

greater than the size of the original message. At configuration time, these data elements must be instantiated with specific sorts and operations.

The higher-order connector itself, which we name *Compression(Uni-comm)[Σ^{cd}]*, is defined by

- the binary connector *COMPRESSION*



where the glue *comp-decomp[Σ^{cd}]* is defined in terms of a configuration with the following two components:

```
design comp[Σcd] is
in     di:t
out    co:s
prv    v:t; rd,msg:bool
do     drec: ¬msg → v:=di ‖ msg:=true
[] prv comp: rd msg → co:=comp(v) ‖ rd:=true
[]     csend:rd → rd:=false ‖ msg:=false

design decomp[Σcd] is
in     ci:s
out    do:t
prv    v:s; rd,msg:bool
do     crec: ¬msg → v:=ci ‖ msg:=true
[]     dec: rd msg → do:=decomp(v) ‖ rd:=true
[]     dsend: rd → rd:=false ‖ msg:=false
```

Design *comp[Σ^{cd}]* models the compression of messages of type *t* received through *di* into messages of type *s* that are then transmitted through *co*. Design *decomp[Σ^{cd}]* models the decompression of messages of type *s* received through *ci* into messages of type *t* that are then transmitted through *do*.

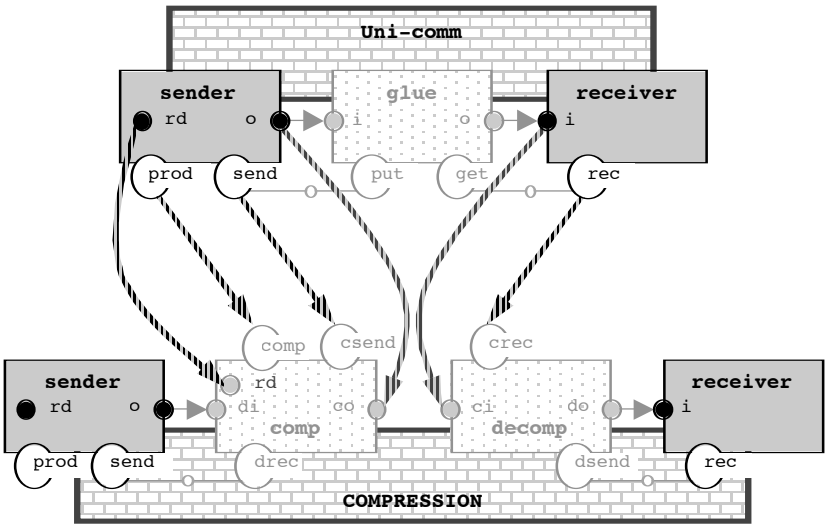
- the connector *Uni-comm[s]* — the formal parameter;
- the refinement morphisms σ_s*:sender[s]* → *comp-decomp[Σ^{cd}]* and σ_r*:receiver[s]* → *comp-decomp[Σ^{cd}]* induced, respectively, by

  σ_s*:sender[s]* → *comp[Σ^{cd}]*
  $\sigma_s(o)=co$, $\sigma_s(rd)=rd$, $\sigma_s(comp)=prod$, $\sigma_s(csend)=send$
  σ_r*:receiver[s]* → *decomp[Σ^{cd}]*
  $\sigma_r(i)=ci$, $\sigma_r(crec)=rec$

Because components *comp* and *decomp* do not interact, any component refined by one of them is also refined by their composition

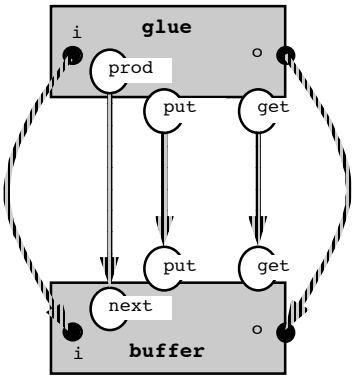
comp-decomp[$\Box^{cd}$]. The corresponding induced morphisms have only to take into account the renaming of variables and actions that takes place in composition.

Putting all these elements together, we get a graphical representation of the higher-order connector *Compression(Uni-comm)*[$\Box^{cd}$]:



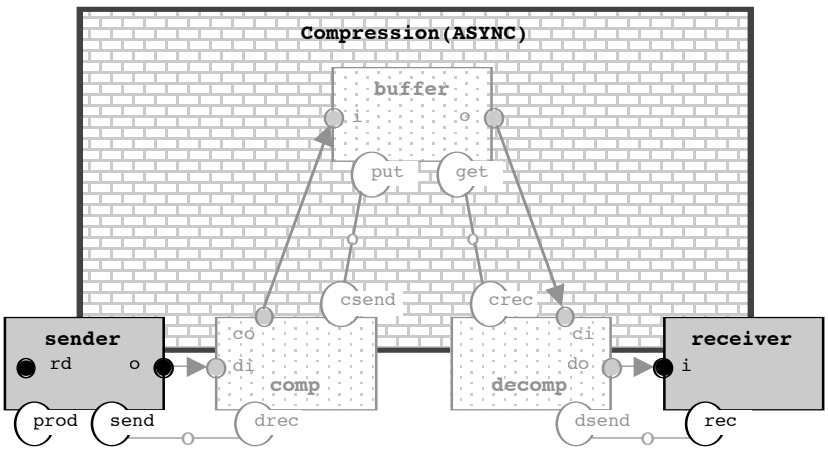
In summary, *Compression(Uni_comm)*[$\Box^{cd}$] has the formal parameter *Uni-comm[s]*, which restricts the actual connectors to which the service of compression/decompression can be applied — it requires that any actual parameter (connector) models a unidirectional communication protocol. The connector *COMPRESSION* describes, on the one hand, that messages sent by the actual sender are transmitted to *comp*; which compresses them; and, on the other hand, that *decomp* decompresses the messages it receives and delivers the result to the actual receiver. Finally, the two refinement morphisms establish the instantiation of *Uni-comm[s]* with *comp[s]* in the role of *sender*, and *decomp[s]* in the role of *receiver*. In this way, the formal parameter *Uni-comm[s]* is the connector used to transmit compressed messages.

Now it remains to explain the procedure of parameter passing, i.e., how the service just described can be installed over a specific connector and how the resulting connector is obtained. As an example, consider again *ASYNC*. It is not difficult to realize that we may replace *Uni-comm[s]*, as the formal parameter of *Compression(Uni-comm)*[$\Box^{cd}$], by *ASYNC* because this connector does model a unidirectional communication protocol. More concretely, *ASYNC* has exactly the same roles as *Uni-comm* and its glue – *buffer[s]* – is a refinement of *Uni-comm*'s glue.



In a more general situation, the instantiation of a higher-order connector is established by a suitable fitting morphism from the formal to the actual connector. Such a morphism formulates the correspondence between the roles and glue of the formal parameter with those of the actual parameter connector. We will present and discuss these morphisms in more detail further down.

The construction of a new connector from the given higher-order connector and the actual parameter connector is straightforward. We only need to compose the interconnections of the *buffer* to *sender* and *receiver* with the refinements σ_s and σ_r that define the instantiation of *Uni-comm* with *comp* and *decomp*, respectively.



For example, channel *co* of *comp* becomes connected to the input chanel *i* of *buffer* because *co* corresponds to the chanel *o* of *sender* which in turn is, in *ASYNC*, connected to *i*. The resulting configuration fully defines the connector *Compression(ASYNC)*[σ^{cd}+*K*]. Its roles are

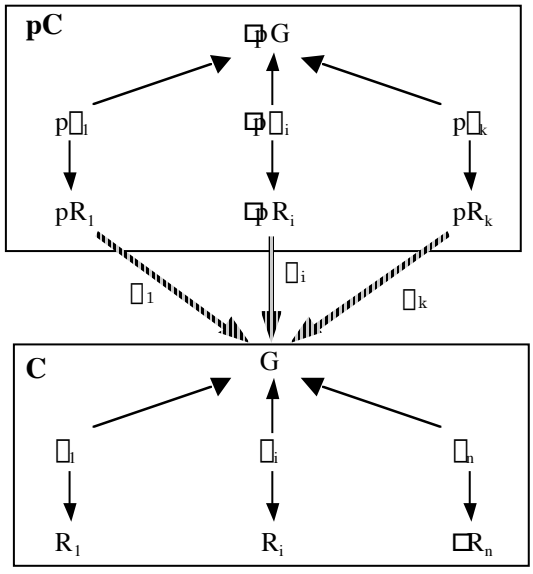
sender and *receiver* and its glue is defined in terms of a configuration involving *comp*, *decomp* and *buffer* as shown above.

Summarising, we described the installation of a compression-decompression service over a unidirectional communication protocol as a parameterised entity that has connectors as parameters and delivers a connector as a result of its instantiation. This is why it is called a higher-order connector. Then we explained how the higher-order connector can be instantiated with a specific connector and, finally, we showed how the resulting connector is obtained.

10.2.1 Definition – higher-order connector

A *higher-order connector* (hoc) consists of

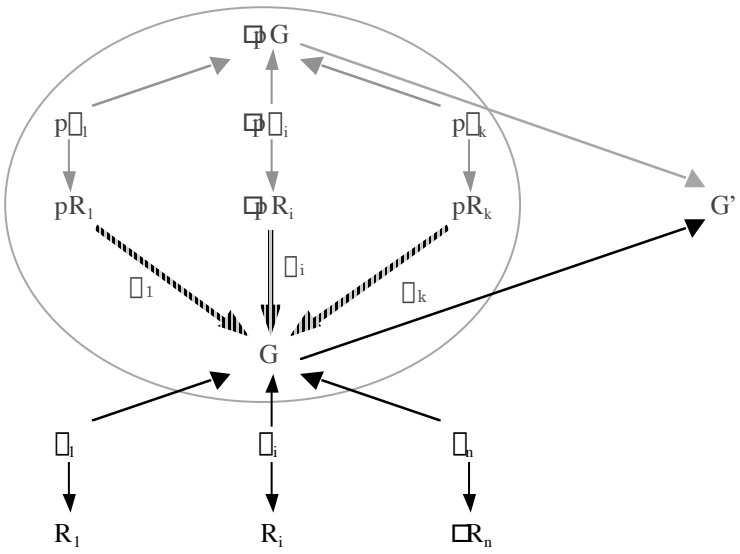
- a connector *pC*, called the formal parameter of the hoc; its roles, glue and connections are called, respectively, the parametric roles, the parametric glue and the parametric connections of the hoc;
- a connector *C* – its roles and glue are also called the roles and the glue of the hoc;
- an instantiation of the formal parameter connector with the glue of the hoc, i.e., a refinement morphism σ_i from each of the parametric roles to the glue, such that the diagram in ***c-DESC*** obtained by composing the role morphism of each parametric connection with its instantiation constitutes a well-formed configuration.



More examples can be found in [78].

10.2.2 Definition – semantics of a higher-order connector

The semantics of a higher-order connector is the connector depicted below. Its roles are the roles of *C* and its glue is *G'*, the design returned by the colimit of the configuration $pC+(\square_i)$.



For simplicity, we have imposed one single parameter to the higher-order connector. However, the definition can be extended to the case of several parameters in a straightforward way.

Intuitively, the instantiation of the formal parameter of a higher-order connector can be regarded as the replacement of a connector (the formal parameter *pC*) that was instantiated to given components of a system (the glue of the hoc) by another connector (the actual parameter). In addition, the type of interconnection that *pC* ensures must be preserved. In other words, the design that results from the replacement must be a refinement of the design from which we started.

Like for connectors, the instantiation of the formal parameter of a higher-order connector is established via a fitting morphism from the formal to the actual parameter. These morphisms, on the one hand, formulate the correspondence between roles and glue of the formal with those of the actual parameter and, on the other hand, capture conditions under which the "functionality" of the formal parameter is preserved.

In order to be able to use, in the design of a given system, a connector *C* in place of a connector *C'*, it is obvious that the two connectors must have the same number of roles. Furthermore, *C'* must be able to

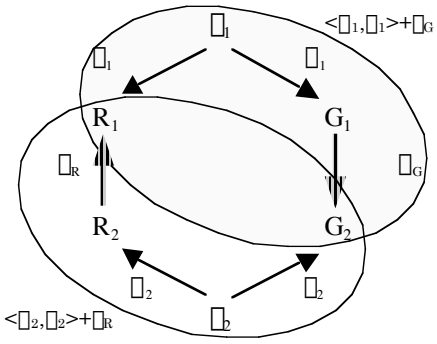
be instantiated with the same components as *C*. That is to say, every restriction on the components to which *C*' can be applied must also be a restriction imposed by *C*. In this way, fitting morphisms must require that each of the roles of *C*' is refined by the corresponding role of *C*.

As shown through the connector *Uni-comm*, connectors may be based on glues that are not fully developed as designs (may be underspecified). Nevertheless, the concrete commitments that they make may determine, to some extent, the type of interconnection that the connector will ensure. The type of interconnection is clearly preserved if we simply consider a less unspecified glue, i.e., if we refine the glue. Hence, fitting morphisms must allow for arbitrary refinements of the glue.

Having this in mind, we arrive at the following notion of fitting morphism:

10.2.3 Definition – fitting morphism

A morphism ϕ from a connection $\langle \square_1 : \mathbf{desc}(\square_1) \square G_1, \square_1 : \mathbf{desc}(\square_1) \square R_1 \rangle$ to a connection $\langle \square_2 : \mathbf{desc}(\square_2) \square G_2, \square_2 : \mathbf{desc}(\square_2) \square R_2 \rangle$, also called a *fitting morphism*, consists of a pair $\langle \square_G : G_1 \square G_2, \square_R : R_2 \square R_1 \rangle$ of refinement morphisms in ***r-DESC*** s.t. the interconnection $\langle \square_1, \square_1 \rangle + \square_G$ of R_1 with G_2 is refined by the interconnection $\langle \square_2, \square_2 \rangle + \square_R$.



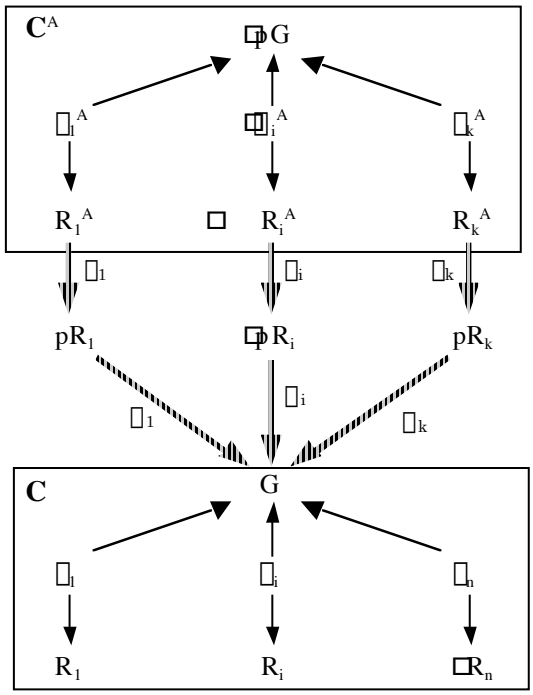
A fitting morphism ϕ from a connector C_1 to a connector C_2 with the same number of connections consists of a fitting morphism ϕ from each of C_1's connections to each of C_2's connections, all with the same glue refinement $\square_G$.

If there exists a fitting morphism from a connector C_1 to a connector C_2, then we may replace each occurrence of the connector C_1 in an architectural description of a system by an occurrence of C_2. The compositionality of the design formalism ensures that every coordination decision made previously is preserved.

Based on fitting morphisms between connectors, we define an instantiation of a higher-order connector.

10.2.4 Definition – instantiation of a higher-order connector

An instantiation of a higher-order connector with formal parameter *pC* consists of a connector C^A (the actual parameter) together with a fitting morphism $\chi : pC \to C^A$ such that a well-formed configuration is obtained by first composing the role morphisms of each actual connection with the corresponding fitting component, and then with the role instantiation.



The semantics of a higher-order connector instantiation is the connector with the same roles as *C* and whose glue is a design returned by the colimit of the configuration $C^A + (\chi_i; \iota_i)$.

Higher-order connectors facilitate the separation of concerns in the development of complex connectors and their compositional construction. An important feature of our notion of higher-order connector is that different kinds of functionality, modelled separately by different higher-order connectors, can be combined, giving rise also to a higher-order connector. In this way, it is possible to analyse the properties that such compositions exhibit, namely to investigate whether undesirable properties emerge and desirable properties are preserved.

The key idea for composition of hocs is the instantiation of a hoc with a hoc — *parameterised instantiation*. Examples, definitions and properties of this mechanism can be found in [78].

References

1. J.Adámek, H.Herrlich and G.Strecker, *Abstract and Concrete Categories*, John Wiley & Sons 1990.
2. R.Allen and D.Garlan, "A Formal Basis for Architectural Connectors", *ACM TOSEM* 6(3), 1997, 213-249.
3. F.Arbab, "What Do You Mean, Coordination?", in *Bulletin of the Dutch Association for Theoretical Computer Science (NVTI)*, March 1998, (available at www.cwi.nl/pub/manifold/NVTIpaper.ps.Z).
4. M.Arbib and E.Manes, *Arrows, Structures and Functors: The Categorical Imperative*, Academic Press 1975.
5. M.Arrais and J.Fiadeiro, "Unifying Theories in Different Institutions", in M.Haveraaen, O.Owe and O-J.Dahl (eds), *Recent Trends in Data Type Specification*, LNCS 1130, Springer-Verlag 1996, 81-101.
6. A.Asperti and G.Longo, *Categories, Types and Structures: An Introduction to Category Theory for the Working Computer Scientist*, MIT Press 1991.
7. E.Astesiano, M.Broy and G.Reggio. "Algebraic Specification of Concurrent Systems", in [8].
8. E.Astesiano, B.Krieg-Bruckner and H.-J.Kreowski (eds) *IFIP WG 1.3 Book on Algebraic Foundations of System Specification*, Springer Verlag, 1999.
9. R.J.Back, ”Refinement calculus II: Parallel and reactive programs”, in J.deBakker,W.deRoever and G.Rozenberg (eds), *Stepwise Refinement of Distributed Systems*, LNCS 430, Springer-Verlag 1990, 67-93.
10. R.J.Back and K.Sere, “Superposition Refinement of Reactive Systems”, *Formal Aspects of Computing*, 8(3):324-346, 1996.
11. J.P.Banâtre and D.Le Métayer, "Programming by Multiset Transformation", *Communications ACM* 16(1), 1993, 55-77.
12. M.Barr and C.Wells, *Category Theory for Computing Science*, Prentice-Hall International Series in Computer Science 1990.
13. H.Barringer, "The Use of Temporal Logic in the Compositional Specification of Concurrent Systems", in A.Galton (ed) *Temporal Logics and their Applications,* Academic Press 1987.
14. L.Bass, P.Clements and R.Kasman, *Software Architecture in Practice*, Addison-Wesley 1998.
15. M.Bednarczyk, *Categories of Asynchronous Transition Systems*, PhD Thesis, University of Sussex, 1988.

16. G.Berry and G.Boudol, "The Chemical Abstract Machine", *Theoretical Computer Science* 96, 1992, 217-248.
17. R.Burstall and J.Goguen, "Putting Theories together to make Specifications", in R.Reddy (ed) *Proc. Fifth International Joint Conference on Artificial Intelligence,* 1977, 1045-1058.
18. R.Burstall and J.Goguen, "The Semantics of CLEAR, a Specification Language", in *Proc. Advanced Course on Abstract Software Specification*, LNCS 86, Springer-Verlag 1980, 292-332.
19. K.Chandy and J.Misra, *Parallel Program Design - A Foundation*, Addison-Wesley 1988.
20. J.F.Costa, A.Sernadas and C.Sernadas, "Data Encapsulation and Modularity: Three Views of Inheritance", in A.Borzyszkowski and S.Sokolowski (eds), *Mathematical Foundations of Computer Science*, LNCS 711, Springer-Verlag 1993, 382-391.
21. J.F.Costa, A.Sernadas, C.Sernadas and H.-D.Ehrich, "Object Interaction", in *Mathematical Foundations of Computer Science*, LNCS 629, Springer-Verlag 1992, 200-208.
22. R.Crole, *Categories for Types*, Cambridge Mathematical Textbooks, Cambridge University Press 1993.
23. E.Dijkstra, *A Discipline of Programming*, Prentice-Hall International 1976.
24. H.-D.Ehrich, J.Goguen and A.Sernadas, "A Categorial Theory of Objects as Observed Processes", in J.deBakker, W.deRoever and G.Rozenberg (eds) *Foundations of Object-Oriented Languages*, LNCS 489, Springer Verlag 1991, 203-228.
25. H.-D.Ehrich, A.Sernadas and C.Sernadas, "Objects, Object Types and Object Identity", in H.Ehrig *et al* (eds) *Categorical Methods in Computer Science with Aspects from Topology*, LNCS 393, Springer-Verlag 1989, 142-156.
26. H.Ehrig, M.Große-Rhode and U.Wolter, "Applications of category theory to the area of algebraic specification in computer science", *Applied Categorical Structures* 6, 1998, 1-35.
27. H.Ehrig, K.-D.Kiermeier, H.-J.Kreowski and W.Kuhnel, *Universal Theory of Automata: A Categorical Approach*, B.G.Teubner, Stuttgart 1974.
28. H.Ehrig and B.Mahr, *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of EATCS Monographs on Theoretical Computer Science, Springer-Verlag 1985.
29. H.Ehrig and B.Mahr, *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*, volume 21 of EATCS Monographs on Theoretical Computer Science, Springer-Verlag 1990.
30. E.Emerson and E.Clarke, "Using Branching Time Logic to Synthesize Synchronisation Skeletons", *Science of Computer Programming* 2, 1982, 241-266.
31. J.Fiadeiro, "On the Emergence of Properties in Component-Based Systems", in M.Wirsing and M.Nivat (eds) *Proc. International Conference on Algebraic*

Methodology and Software Technology (AMAST'96), LNCS 1101, Springer-Verlag 1996, 421-443.
32. J.Fiadeiro and J.F.Costa, "Mirror, Mirror in My Hand: a duality between specifications and models of process behaviour", *Mathematical Structures in Computer Science* 6, 1996, 353-373.
33. J.Fiadeiro and J.F.Costa, "Institutions for Behaviour Specification", in E.Astesiano, G.Reggio and A.Tarlecki (eds) *Recent Trends in Data Type Specification*, LNCS 906, Springer-Verlag 1995, 273-289.
34. J.Fiadeiro and A.Lopes, "Semantics of Architectural Connectors", in M.Bidoit and M.Dauchet (eds) *Proc. TAPSOFT'97*, LNCS 1214, Springer-Verlag 1997, 505-519.
35. J.Fiadeiro and A.Lopes, "Algebraic Semantics of Coordination, or what is in a signature?", in A.Haeberer (ed) *Proc. International Conference on Algebraic Methodology and Software Technology (AMAST'98)*, Springer-Verlag 1999.
36. J.Fiadeiro, A.Lopes and T.Maibaum, "Synthesising Interconnections", in R.Bird and L.Meertens (eds) *Algorithmic Languages and Calculi*, Chapman Hall 1997, 240-264.
37. J.Fiadeiro, A.Lopes and M.Wermelinger, "A Mathematical Semantics for Architectural Connectors", in R.Backhouse and J.Gibbons (eds) *Generic Programming*, LNCS 2793, Springer-Verlag 2003, 90-234.
38. J.Fiadeiro and T.Maibaum, "Describing, Structuring, and Implementing Objects", in J.deBakker, W.deRoever and G.Rozenberg (eds) *Foundations of Object-Oriented Languages*, LNCS 489, Springer-Verlag 1991, 274-310.
39. J.Fiadeiro and T.Maibaum, "Temporal Theories as Modularisation Units for Concurrent System Specification", *Formal Aspects of Computing* 4(3), 1992, 239-272.
40. J.Fiadeiro and T.Maibaum, "Verifying for Reuse: Foundations of Object-Oriented System Verification", in C.Hankin, I.Makie and R.Nagarajan (eds) *Theory and Formal Methods 1994*, World Scientific Publishing Company 1995, 235-257.
41. J.Fiadeiro and T.Maibaum, "Interconnecting Formalisms: supporting modularity, reuse and incrementality", in G.E.Kaiser (ed) *Proc. 3rd Symposium on Foundations of Software Engineering*, ACM Press 1995, 72-80.
42. J.Fiadeiro and T.Maibaum, "A Mathematical Toolbox for the Software Architect", in *Proc. 8th International Workshop on Software Specification and Design*, IEEE Computer Society Press 1996, 46-55.
43. J.Fiadeiro and T.Maibaum, "Design Structures for Object-Based Systems", in S.Goldsack and S.Kent (eds) *Formal Methods and Object Technology*, Springer-Verlag 1996, 183-204.
44. J.Fiadeiro and T.Maibaum, "Categorical Semantics of Parallel Program Design", *Science of Computer Programming* 28(2-3), 1997, 111-138.
45. J.Fiadeiro and A.Sernadas, "Structuring Theories on Consequence", in D.Sannella and A.Tarlecki (eds) *Recent Trends in Data Type Specification*, LNCS 332, Springer-Verlag 1988, 44-72.

46. P.Finger, "Componend-Based Frameworks for E-Commerce", *Communications of the ACM* 43(10), 2000, 61-66.
47. N.Francez and I.Forman, *Interacting Processes*, Addison-Wesley 1996.
48. P.Freyd and A.Scedrov, *Categories, Allegories*, North-Holland Mathematical Library, vol 39, North-Holland 1990.
49. D.Garlan and D.Perry, "Software Architecture: Practice, Potential, and Pitfalls", in *Proc. 16th ICSE*, 1994, 363-364.
50. D.Gelernter and N.Carriero, "Coordination Languages and their Significance", *Communications ACM* 35(2), 1992, 97-107.
51. J.Goguen, "Mathematical Representation of Hierarchically Organised Systems", in E.Attinger (ed) *Global Systems Dynamics*, Krüger 1971, 112-128.
52. J.Goguen, "Categorical Foundations for General Systems Theory", in F.Pichler and R.Trappl (eds) *Advances in Cybernetics and Systems Research*, Transcripta Books 1973, 121-130.
53. J.Goguen, "Objects", *International Journal of General Systems* 1(4), 1975, 237-243.
54. J.Goguen, "Reusing and Interconnecting Software Components", *IEEE Computer* 19(2), 1986, 16-28.
55. J.Goguen, "Principles of Parametrised Programming", in Biggerstaff and Perlis (eds) *Software Reusability*, Addison-Wesley 1989, 159-225.
56. J.Goguen, "A Categorical Manifesto", *Mathematical Structures in Computer Science* 1(1), 1991, 49-67.
57. J.Goguen, "Sheaf Semantics of Concurrent Interacting Objects", *Mathematical Structures in Computer Science* 1(2), 1991, 159-191.
58. J.Goguen, "Parametrised Programming and Software Architecture", in *Symposium on Software Reusability*, IEEE 1996.
59. J.Goguen and R.Burstall, "Some fundamental algebraic tools for the semantics of computation, part 1: Comma categories, colimits, signatures and theories", *Theoretical Computer Science* 31(2), 1984, 175-209.
60. J.Goguen and R.Burstall, "Some fundamental algebraic tools for the semantics of computation, part 2: Signed and abstract theories", *Theoretical Computer Science* 31(3), 1984, 263-295.
61. J.Goguen and R.Burstall, "Institutions: Abstract Model Theory for Specification and Programming", *Journal of the ACM* 39(1), 1992, 95-146.
62. J.Goguen and R.Burstall, "A Study in the Foundations of Programming Methodology: Specfications, Institutions, Charters and Parchments", in D.Pitt et al (eds) *Category Theory and Computer* Programming, LNCS 240, Springer-Verlag 1986, 313-333.
63. J.Goguen and S.Ginali, "A Categorical Approach to General Systems Theory", in G.Klir (ed) *Applied General Systems Research*, Plenum 1978, 257-270.
64. J.Goguen, J.Thatcher and E.Wagner, "An initial algebra approach to the specification, correctness and implementation of abstract data types", in

R.Yeh (ed) *Current Trends in Programming Methodology* IV, Prentice Hall 1978, 80-149.
65. J.Goguen, J.Thatcher E.Wagner and J.Wright, *A junction between computer science and category theory, I: Basic concepts and examples*, Technical report, IBM Watson Research Center, Yorktown Heights NY, 1973. Report RC 4526 (part 1) and 5908 (part 2).
66. R.Goldblatt, *Logics of Time and Computation*, CSLI 1987.
67. R.Goldblatt, *Topoi: The Categorial Analysis of Logic*, North-Holland1989.
68. C.A.R.Hoare, *Communicating Sequential Processes*, Prentice-Hall International 1985.
69. B.Jacobs, *Categorical Logic and Type Theory*, Studies in Logic and the Foundations of Mathematics, Vol 141, Elsevier 2001.
70. S.Johnson, *Emergence: The connected lives of ants, brains, cities and software*, The Penguin Press 2001.
71. S.Katz, "A Superimposition Control Construct for Distributed Systems", *ACM TOPLAS* 15(2), 1993, 337-35.
72. W.Kent, "Participants and Performers: A Basis for Classifying Object Models", in *Proc. OOPSLA 1993 Workshop on Specification of Behavioral Semantics in Object-Oriented Information Modeling*, 1993.
73. J.Lambek and P.Scott, *Introduction to Higher-Order Categorical Logic*, Cambridge University Press 1986.
74. F.W.Lawvere and S.H.Schanuel, *Conceptual Mathematics – A first introduction to categories*, Cambridge University Press 1997.
75. J.Loeckx, H-D.Ehrich and M.Wolf, *Specification of Abstract Data Types*, Wiley 1996.
76. A.Lopes and J.L.Fiadeiro, "Using Explicit State to Describe Architectures", in E. Astesiano (ed), *FASE'99*, LNCS 1577, Springer-Verlag 1999, 144–160.
77. A.Lopes and J.L.Fiadeiro, "Superposition: Composition vs Refinement of Non-deterministic Action-based Systems", in *Formal Aspects of* Computing, in print (a preliminary version appeared in *Electronic Notes on Theoretical Computer Science*70 (3), 2002).
78. A.Lopes, M.Wermelinger and J.L.Fiadeiro, "Higher-Order Architectural Connectors", in *ACM Transactions on Software Engineering Methodology*, January 2004
79. S.MacLane, *Categories for the Working Mathematician*, Springer-Verlag 1971.
80. J.Magee, J.Kramer and M.Sloman, "Constructing Distributed Systems in Conic", *IEEE TOSE* 15 (6), 1989.
81. T.Maibaum and M.Sadler, "Axiomatising Specification Theory", *Proc. 3rd Abstract Data Type Workshop*, Fachbereich Informatik 25, Springer Verlag 1984.

82. T.Maibaum and W.Turski, "On What Exactly Goes On When Software Is Developed Step by Step" *Proc. 7th Int. Conference on Software Enginnering*, IEEE 1984, 528-533.
83. T.Maibaum, P.Veloso and M.Sadler, "A Theory of Abstract Data Types for Program Development: Bridging the Gap?" in *Formal Methods and Software Development*, LNCS 186, Springer-Verlag 1985.
84. T.Maibaum, P.Veloso and M.Sadler, "A Logical Approach to Specification", submitted for publication, Technical Report, Department of Computing, Imperial College 1990.
85. Z.Manna and P.Wolper, "Synthesis of Communicating Processes from Temporal Logic Specifications", in *ACM Transactions on Programming Languages and Systems* 6(1), 1984, 68-93.
86. N.Mehta, N.Medvidovic and S.Phadke, "Towards a taxonomy of software connectors", in *Proc. of the 22nd Intl. Conf. on Software Engineering*, ACM Press 2000, 178-187.
87. J.Meseguer, "General Logics", in H.-D.Ebbinghaus et al (eds) *Logic Colloquium 87,* North-Holland 1989.
88. B.Meyer, *Object-Oriented Software Construction*, Addison-Wesley 1992.
89. M.Moriconi and X.Qian, "Correctness and Composition of Software Architectures", *Proc. Second Symposium on the Foundations of Software Engineering*, ACM Press 1994, 164-174.
90. D.Perry and A.Wolf, "Foundations for the Study of Software Architectures", *ACM SIGSOFT Software Engineering Notes* 17(4):40-52, 1992.
91. B.Pierce, *Basic Category Theory for Computer Scientists*, MIT Press 1991.
92. D.Pitt, S.Abramsky, A.Poigné and D.Rydeheard (eds), *Category Theory and Computer Programming*, LNCS 240, Springer-Verlag 1985.
93. A.Poigné, "Basic Category theory', in *Handbook of Logic in Computer Science, volume 1*, Oxford University Press 1992, 413-640.
94. H.Reichel, *Initial Computability, Algebraic Specifications, and Partial Algebras*, Oxford University Press 1987.
95. G.-C.Roman, P.J.McCann and J.Y.Plun, “Mobile UNITY: Reasoning and Specification in Mobile Computing”, *ACM TOSEM*, 6(3), 1997, 250-282.
96. R.Rosen, *Life Itself*, Columbia University Press 2000.
97. D.Sannella and A.Tarlecki, "Building Specifications in an Arbitrary Institution", *Information and Control* 76, 1988, 165-210.
98. D.Sannella, S.Sokolowski and A.Tarlecki, "Toward Formal Development of Programs from Algebraic Specifications: Parameterisation Revisited", *Acta Informatica* 29, 1992, 689-736.
99. V.Sassone, M.Nielsen and G.Winskel, "Models for Concurrency: Towards a Classification", *Theoretical Computer Science* 170, 1996, 277-296
100. M.Shaw, "Procedure Calls are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status", in D.A. Lamb (ed) *Studies of Software Design*, LNCS 1078, Springer-Verlag 1996.

101. M.Shaw and D.Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996
102. D.Smith, "Constructing Specification Morphisms", *Journal of Symbolic Computation* 15 (5-6), 571-606, 1993.
103. Y.Srinivas and R.Jüllig, "Specware™:Formal Support for Composing Software", in B.Möller (ed) *Mathematics of Program Construction*, LNCS 947, 399-422, Springer-Verlag, 1995.
104. A.Tarlecki, R.Burstall and J.Goguen, "Some fundamental algebraic tools for the semantics of computation, part 3: Indexed categories", *Theoretical Computer Science* 91, 1991, 239-264
105. P.Veloso and T.Pequeno, "Interpretations between Many-sorted Theories", in *Proc. 2nd Brasilian Colloquium on Logic*, 1978
106. P.Veloso, T.Maibaum and M.Sadler, "Program Development and Theory Manipulation", *Proc. Third International Workshop on Software Specification and Design,* London, IEEE Computer Society Press 1985, 228-232.
107. P.Veloso and T.Maibaum, "On the Modularisation Theorem for Logical Specifications", *Information Processing Letters* 53, 1995, 287-293.
108. R.Walters, *Categories and Computer Science*, Cambridge Computer Science Texts 1991.
109. W.Wechler, *Universal Algebra for Computer Scientists*, EATCS Monographs on Theoretical Computer Science 25, Springer-Verlag 1992.
110. G.Winskel, "Synchronization Trees", *Theoretical Computer Science* 34, 1984.
111. G.Winskel, "Categories of Models for Concurrency", in S.D.Brookes et al (eds), *Proceedings of the Seminar on Concurrency*, Springer-Verlag, 1985, 246-267.
112. G.Winskel, "Petri Nets, Algebras, Morphisms and Compositionality", in *Information and Computation* 72, 1987, 197-238.
113. G.Winskel and M.Nielsen, "Models for Concurrency", in S.Abramsky, D.Gabbay and T.Maibaum (eds) *Handbook of Logic in Computer Science* 4, Oxford University Press 1995.
114. P.Wolper, "On the Relation of Programs and Computations to Models of Temporal Logic", in B.Banieqbal, H.Barringer and A.Pnueli (eds) *Temporal Logic in Specification*, LNCS 398, Springer-Verlag 1989, 75-123.
115. P.Zave, “Feature Interactions and Formal Specifications in Telecommunications”, *IEEE Computer* XXVI(8), 1993, 20-30.
116. P.Zave and M.Jackson, "Conjunction as Composition", *ACM TOSEM* 2(4), 1993, 371-411.

Index

D

E

F

T

U

Z

–