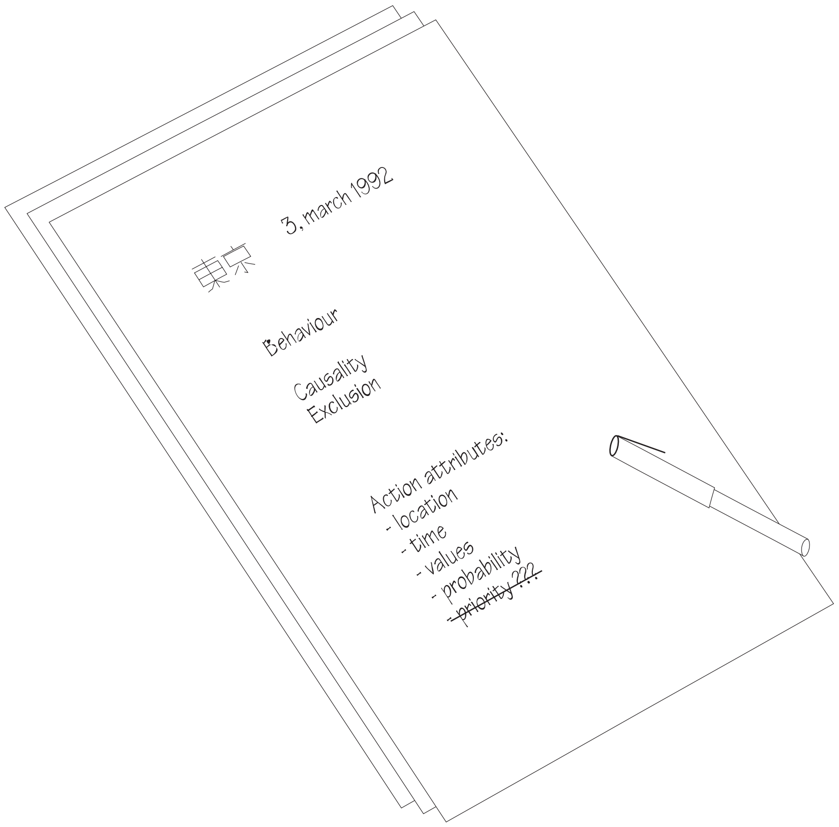
Architectural Notes: a Framework for Distributed Systems Development



Luís Ferreira Pires

Architectural Notes: a Framework for Distributed Systems Development

CTIT Ph. D-thesis series No. 94-01

P.O. Box 217 - 7500 AE Enschede - The Netherlands
telephone +31-53-893779 / fax +31-53-333815

Centre for
Telematics and
Information
Technology

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Ferreira Pires, Luís

Architectural notes: a framework for distributed systems
development / Luís Ferreira Pires. - [S.1. : s.n.]. -
I11.
Proefschrift Enschede. - Met lit. opg.
ISBN 90-9007461-9 geb.
Trefw.: systeemontwerp

ARCHITECTURAL NOTES:
A FRAMEWORK FOR DISTRIBUTED SYSTEMS DEVELOPMENT

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de Rector Magnificus,
prof.dr. Th.J.A. Popma,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op vrijdag 23 september 1994 te 13:15

door
Luís Ferreira Pires

geboren op 7 april 1961
te São Paulo, Brazilië

Dit proefschrift is goedgekeurd door de promotor
prof.dr.ir. C.A. Vissers

"The tribal frame of values that condemned the brujo and led to his punishment was one kind of good, for which Phaedrus coined the term 'static good'. ... But in addition there's a Dynamic good that is outside of any culture, that cannot be contained by any system of precepts, but has to be continually rediscovered as a culture evolves." Lila – an inquiry into morals, Robert M. Pirsig (1991)

Preface

Information systems play increasingly important roles in our society, which makes society increasingly dependent of their correct functioning. This results in high demands for functionality, reliability and availability of such systems. The need for a better control of a systems' design process has solicited the scientific community to develop design methodologies. These methodologies in general draw attention to system planning, also called high-level design, before the actual implementation is performed.

The importance of distribution aspects in information systems design has also increased in the last decades. The critical requirements acknowledged for stand-alone information systems, get an extra dimension in the case of distributed cooperating information systems, mainly due to severe inter-operability requirements. Especially in the area of distributed systems design we can talk of a *methodology crisis*: people of different expertise from different design cultures, used to different methodologies and emphasizing different design aspects, trying to work together, but finding enormous difficulties in communicating due to the lack of common conceptual basis and terminology.

Some methodologies meant to support information system design are based on formal methods. In the area of distributed systems design, formal methods have been presented for some time as a panacea, but their limitations to fulfil the designer's requirements and the frustration that followed have often put them in discredit. Most of these limitations are caused by the historical fact that these methods have been developed to support 'distributed programming', as an evolution of 'sequential programming', and have been applied to support 'distributed systems design'. It might be clear now to the scientific community that distributed programming is quite a different discipline than distributed systems design, and that methods developed for the former fail to support the latter in a satisfactory way.

Designer's frustrations with formal methods for distributed systems design have been caused by unfulfilled promises, such as for example:

- *unambiguous representation*: due to simplification, normally motivated by mathematical elegance or opportunism rather than design objectives, formal models sometimes do not distinguish critically different design concepts. This may simplify the mathematical theory that guarantees mathematical precision, but from a design point of view it results in architectural ambiguity. An example is the arbitrary interleaving semantics, which maps independence and interleaving of action occurrences onto the same concepts;

- *automated transformations*: most automated design transformations available now-a-days are incapable of handling more than toy examples, and will probably remain so until a more general framework for transformations is defined.

Most formal methods for distributed systems design allow a limited coverage of the design process as a whole. We believe that formal methods will not be completely by industry if their support is limited only to some design slice between requirements capturing and realisation, and covering only part of the functional requirements, which is the state-of-the-art today.

It is interesting to observe the current industrial practise. In most of the software industry, for example, structured programming, characterized by encapsulation rules and clear definition of interfaces between software modules, is now becoming current practise, which is around ten to fifteen years behind research developments. Formal methods are only mastered by some research groups, and have a very small penetration in industry.

We believe that designers are not yet prepared to incorporate formal methods in their daily practises, and that formal methods are not mature enough to be applied at a large scale. In the first place designers are in need of conceptual tools to reason about design, represent their design decisions, etc. These conceptual tools have to be intuitively easy to understand and should allow the representation of various abstractions of the systems under design. The development of such conceptual tools motivated the development of this thesis.

When we started this work, our objective was to develop design support (methods, tools, environments, etc.) for system realisation from formal specifications. While studying this problem we came to the conclusion that the design model we were using at that time had some severe limitations that blocked our way down to realisation. At this point we started reconsidering our design model, and we found out that still a lot had to be done in the area of design concepts and their manipulations. Therefore the focus of our work shifted from realisation to the development of a design model. We firmly believe that in this way we have paved a solid road towards realisation, which can be used in the years to come.

This thesis aims at the development of a sufficiently general framework, in which those limitations of current formal methods can be studied and avoided in the light of more advanced design concepts. The process of developing a framework which is more general than the 'usual' one does not happen without pain. Like the eccentric *brujo*[1] of the village in Pirsig's Lila, one runs the risk of being misunderstood and becoming isolated when criticizing a framework which has been generally accepted for a long time. In that book, the term Dynamic quality is used to denote the changing forces, while static quality denotes the existing framework. Back to the village after some years, one sees that the brujo has become the leader, being recognized by the community. The lesson is: when Dynamic quality wins the battle the whole community experiences an evolution cycle.

1. from spanish: male witch; magician with evil powers; man who has made an agreement with the devil

Abstraction is sometimes mistaken for lack of expressive power. In this way design concerns that cannot be handled by one's favourite formalism are sometimes ignored, as if such design concerns are not relevant. In order to avoid this mistake, this thesis focuses on the definition and manipulation of design concepts before using any formal representation. However concepts have to represented, which constitutes a dilemma: when we introduce some form of representation we may again fall in the limitations of this representation. We acknowledge this problem, but we have noticed that the results obtained by ignoring the limitations of current available design languages have already paid off.

Some of the concepts discussed in this thesis are compared with their representation in LOTOS throughout the text. These comparisons are meant to serve as a reference for those readers that are acquainted with LOTOS, rather than to show advantages and disadvantages of our design model.

The title of this thesis deserves some explanation. The two antagonistic terms *notes* and *framework* are meant to give an idea of the main difficulties around the development of this thesis. 'Architectural notes' is a term first used by Prof. Vissers to denote our collection of unrelated, unsorted, incomplete and often messy annotations which we started writing during his sabbatical leave in Japan in the beginning of 1992. These notes addressed, most of the time, problems related to distributed systems modelling and refinement of designs. 'Framework' denotes a structure to support anything. Most of the effort in the development of this thesis has been concentrated on the development of a framework for distributed systems development (design and implementation), while squeezing the architectural notes into it. Sometimes the framework was too small, in which case it was enlarged, and sometimes the notes would not fit, in which case they were reformulated. The result is this thesis, a single (large) architectural note, where the framework and most of the architectural notes are accommodated.

This thesis aims at providing design methods that allow:

- to shorten the design gap between the formulation of the user requirements and the realization of the system;
- to bridge this gap faster, in order to support industrial competition objectives;
- to improve the quality of the resulting products.

These objectives are achieved by:

- providing more insight in the conceptual basis of architectural and implementation design concepts;
- defining a series of milestones for the elaboration of designs, in terms of specific design objectives identified in the design process;
- providing guidelines and design and structuring concepts for the elaboration of designs at these milestones;
- providing examples of general purpose technical solutions to some design problems;
- providing guidance how to handle these technical solutions in concrete design instances, by indicating how the technical solutions can be related to the design concepts;

This thesis is structured as follows:

- *Chapter 1: Introduction* provides a global problem definition, by defining the boundaries of the thesis (area of research, scope and objectives), relates the thesis to the current trends and design cultures, and indicates the relevance of the work;
- *Chapter 2: Overview of the Design Methodology* presents an overview of the design methodology which is developed in this thesis, by identifying basic design concepts for distributed systems design, and global abstraction levels at which these systems or system parts can be considered;
- *Chapter 3: Behaviour Definition* introduces the concepts necessary to define behaviours, by developing a basic design model in which a behaviour is defined in terms of the causality relations amongst its actions and interactions;
- *Chapter 4: Causality-oriented Behaviour Composition* introduces a behaviour structuring technique called the causality-oriented behaviour composition. This structuring technique is motivated by the need to structure complex behaviours in order to make them understandable for designers, and by the need to define general purpose behaviour modules to be re-used in various instances of behaviour definitions;
- *Chapter 5: Constraint-oriented Behaviour Composition* introduces a behaviour structuring technique called constraint-oriented behaviour composition. This technique consists of representing behaviours in terms of compositions of constraints on the occurrence of actions. These constraints are represented by (sub-)behaviours, which are superposed to each other, resulting in the original behaviour;
- *Chapter 6: Behaviour Refinement* discusses a design operation in which behaviours are replaced by more refined behaviours, which makes it possible to decompose a functional entity in multiple cooperating functional entities;
- *Chapter 7: Action Refinement* discusses a design operation in which actions are replaced by possibly complex activities, which makes it possible to refine interfaces between functional entities;
- *Chapter 8: Design of an Interaction Server* discusses the high-level design of a general purpose component that supports complex interactions between multiple functional entities: the so called *interaction server*;
- *Chapter 9: Conclusion* presents a summary of the conclusions drawn in the other chapters and gives some ideas for further developments;
- *Appendix A: Formal Models for Behaviour Definitions* contains some formal notions that can be useful for defining a formal semantics for causality relations;
- *Appendix B: Additional Consulted References* contains a list of references that have been consulted but have not been mentioned in any chapter.

Acknowledgements

There are a couple of people who have, in one way or another, made it possible for me to write this thesis. I thank some of these people below.

I start with my promotor, Prof. Chris Vissers, who has been a continuous source of inspiration and has made his best to guide and encourage me even when his time became scarce. I am very proud of having him as a promotor, a boss and, most important, as a friend.

Other members of the Discipline Group Architecture, Joost Katoen, Harro Kremer and Mark de Weger, have also given support to the development of this thesis, by commenting on drafts of this work and discussing relevant topics with me. Dick Quartel deserves a special mention, not only for his comments and for trying to apply some of the methods presented in this thesis on examples, which helped me improving them, but also for the encouragement I received from him in some periods of stress. Marten van Sinderen has helped setting up the framework and the behaviour definition technique based on causality.

The Tele-Informatics and Open Systems (TIOS) group are thanked for creating an ideal working atmosphere in a multi-disciplinary environment. I thank particularly Rom Langerak for the fruitful discussions on the relationship between formal methods and architectural concepts. Prof. Ed Brinksma is also thanked for his comments on this thesis.

In the beginning of 1991 Prof. Motoshi Saeki (Tokyo Institute of Technology) and Prof. Kokichi Futatsugi (Electrotechnical Laboratory, Tsukuba) have given Prof. Vissers and I the opportunity to work for approximately three months exclusively on research, in the scope of the Toshiba Chair. Most of the ideas generated in that period have been worked out in this thesis.

I also thank Ingrid Heinen, my partner in life, for her support, and for trying to show me during the last months that there is much more in life than the writing of a PhD Thesis.

Luís Ferreira Pires

List of Contents

Chapter 1

Introduction

This chapter presents the motivation of this thesis, its objectives, its relevance in the area of distributed systems, and the strategy adopted in its development. The concept of design culture is introduced in this chapter to serve as a framework in which important issues in the area of distributed systems modelling, design and implementation are recognized. We conclude that a better understanding of basic design concepts is fundamental in the development of solutions to support the correct design and implementation of distributed systems. This better understanding and the evaluation of its consequences is one of the objectives of this work.

The chapter is further structured as follows: section 1.1 identifies some trends in distributed systems design, section 1.2 introduces and discusses the concept of design culture, section 1.3 discusses aspects related to the translation of specifications into implementations, section 1.4 defines the objectives of this thesis and section 1.5 discusses the strategy adopted in the development of this work.

1.1 Trends in Distributed Systems Design

Experience shows that effective design methodologies are based on the principle of *separation of (design) concerns*. This design principle helps designers to distinguish, isolate and stratify various design concerns along a design trajectory. Each step along the design trajectory addresses only a limited set of design concerns, allowing designers to find optimal solutions for the problems related to these concerns, while not being bothered by other concerns. Such design methodologies provide guidance for designers to perform a series of concrete design tasks in an effective way.

Associated with separation of concerns is the concept of *abstraction*. To abstract means to ignore characteristics of an object which are not relevant from some point of view. Different hierarchically related points of view determine different abstraction levels. In design methodologies, the object we consider is the system to be designed. At the beginning of the design trajectory we abstract from design concerns that determine the irrelevant details of the construction of the system. These design concerns become relevant throughout the design trajectory. This allows one to structure the design trajectory according to well-defined goals and activities, i.e. providing these construction details step-by-step, until the realization of the system is obtained.

The most abstract definition of the system reflects the user requirements, abstracting completely from the many ways in which these requirements can be implemented and realized. This abstract definition is of prime interest to the users, which merely want to be able to understand the functions of the system and use it accordingly. The architects of the system also make use of this most abstract definition as a form of contract, i.e. an agreement with the users on the characteristics of the system that will be built ([5], [9]).

Stretching of the Abstraction Gap

The market demands for the production of increasingly more complex systems make it necessary that more powerful mechanisms to conceive, understand, design and clearly express these systems are available, such that the controllability of the design process is increased. These mechanisms allow one to make the proper abstractions of these systems, causing a tendency towards the introduction of increasingly abstract concepts and higher abstraction levels. Consequently the design trajectory between abstract definition and concrete realization is stretched.

An example is the concept of *transaction*, which can be defined as a single service provision that can be used to represent a composition of different services, such that in order to be provided it requires a simultaneous agreement between the providers of these different services. The definition of a transaction introduces a more abstract view for considering the provision of the services of this transaction, since now one can refer to a transaction without having to consider the details of these services. A concrete example is a traveller who wants to book a trip at a travel agency. The trip should only be booked if the train and plane tickets are confirmed, hotel accommodations are booked and a car is rent. A transaction can be used to model agreement between all involved actors (train and flight companies, hotel, car rental and travel agency). This should be the abstraction level of interest to the clients of the travel agency, since this abstraction level ignores the functions and interactions, either automated or not, which make the transaction possible. These functions and interactions are in most cases far from trivial, and may consume periods varying from a couple of minutes, hours or even days, but the clients are only interested in their outcome, namely the transaction. Transactions can be used to define more complex business operations, which may generate even more abstract concepts (e.g. groups of transactions).

The stretching of the design trajectory presents a number of major problems. By introducing increasingly abstract design concepts, more design concerns are introduced. The order in which these design concerns should be addressed must be determined, while an optimal ordering may depend on specific design instances. This makes a set of alternative design trajectories possible. The gap between requirements capturing and system realization also gets larger when the design trajectory is stretched whereas the design process may become more time consuming. Therefore it becomes more difficult for designers to bridge this gap, and, at the same time, the design process becomes more expensive ([4]).

In the environment of industrial competition, the introduction of high level design concepts is often considered ineffective. Normally only the burden of the many design concerns and many alternative orders in which these can be handled is felt, while the benefits brought on by a systematic approach and the potential improvement on the quality of the final product are not recognized. Very often these benefits are not exploited due to lack of insight.

The increasing number of abstraction levels in the representation of designs can be compared with the introduction of programming languages and compilers to replace hand coding in machine code in the early years of computer science. In both cases the objective has been to define concepts which are closer to the intellectual capabilities of human beings than their implementation forms. Once these concepts are introduced, they can be used as building blocks for the definition of more complex concepts.

One must be careful with the way the design trajectory is structured. Segmenting the design trajectory in too many steps may delay the design process. This can be vital for a company since it may cause the loss of competition with respect to other companies that may not structure the design process so well, but are able to reach the concrete products in a shorter time scale. Defining too few design steps may make these steps too complex and are therefore bound to contain errors.

1.2 Design Cultures

An environment of industrial competition is influenced by a number of factors. The success or failure of a product depends on availability of suppliers, marketing strategies, users support, product maintenance, etc. Amongst these factors, the design and production processes play a fundamental role, since they are the means to bring products into existence.

Industrial competition requires that design and production are performed according to certain measures of speed and effectiveness, which determine their *productivity*. This thesis focuses on the design process. The productivity of this process is determined not only by the availability of automated tools and clear design methods ("cook-book" style), but also by the skills and preferences of the design team, namely those people that design the system. The combination of tools, methods, skills and preferences that determine the productivity of the design process is called a *design culture* ([13]).

Another aspect of industrial competition is *confidentiality*. Industries try to avoid that certain sensitive elements of their design culture, namely those that can guarantee high productivity, are adopted without their consent by their competitors. This means that different companies potentially have different design cultures.

1.2.1 Design and Specification

The purpose of the design process is to produce an instance of a technical object: the *real system*. At the beginning of the design process the real system does not exist. Yet its design must be analysed, manipulated and communicated between designers. This means that at each point in the design process, the technical object has to be represented. The representation of an object has to concentrate on the characteristics of the object that are relevant at each abstraction level of the design trajectory, thus abstracting from irrelevant details.

Design concepts are abstractions of aspects of technical objects. A set of design concepts together with their combination rules define a *design model*. Although a technical object may be unique, there is a multitude of possible abstractions of its aspects. This makes it possible to choose amongst design concepts, and also explains the multitude of design models available in the literature. The choice of a design model must be guided by design objectives and technical needs. An abstraction of a technical object of concern is called a *design*.

Designs have to be documented, communicated and analysed. This implies that a *specification language* is necessary, as a notation for *representing* designs in a concise, complete and unambiguous way. Elements of a specification language, such as its syntax and semantics, must be derived from the relevant design concepts of the technical area of concern, making the language general purpose in its application area.

A design only exists in the designers' imagination, while a *specification* is a representation of a design using a specification language, which means that a design and its specification are distinct entities. Designers can only rely on specifications to refer to designs during the design process.

Figure 1.1 depicts the distinction between a design and its specification, and their relationship with design concepts and specification language.

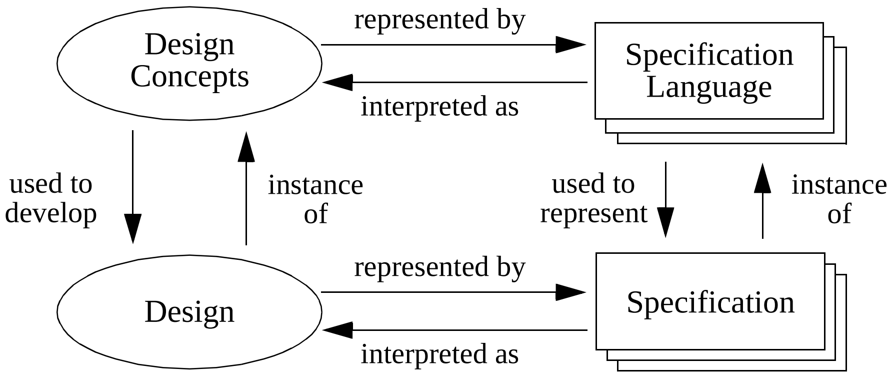


Figure 1.1: Distinction between a design and its specification

1.2.2 Specification Languages

A specification language is suitable for representing designs at a certain abstraction level if there is a clear relationship between design concepts and compositions of language elements to represent them. The term *architectural semantics* denotes the relationship between design concepts and their representations in a specification language ([11], [9]).

Designers should concentrate on the elaboration of designs, using the specification language merely as a vehicle for the representation of design characteristics. A specification language can only be a useful general purpose language if its language elements, syntax and semantics, are defined based on the needs of those who are supposed to use this specification language in the elaboration of designs.

In order to allow precise and unambiguous interpretations of specifications, the semantics of some specification languages are defined as a mapping from syntax constructs of this language onto mathematical structures or models. Language semantics defined in this way are called *formal semantics*. Specification languages with a formal semantics are called *formal description techniques* (FDTs) in the literature.

We conclude that (formal) specification languages should be defined taking into consideration both the design model it supports and the mathematical models of its formal semantics. Furthermore the design model has precedence, which means that the design model should not be corrupted to allow easier mappings onto specific mathematical models.

Figure 1.2 depicts these two important aspects of formal specification languages.

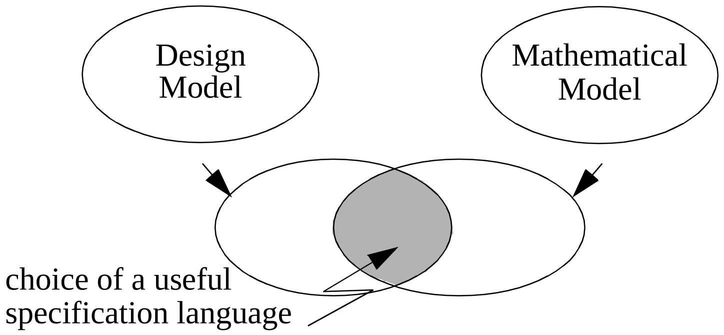


Figure 1.2: Specification language with a design model and mathematical model

The formal semantics of a specification language has the primary goal of allowing one to compare specifications, such that these specifications can be distinguished or considered identical. The relation involving identical specifications is often a mathematical *equivalence*. In case the formal semantics of the specification language closely relates to the design model of an application area these mathematical equivalence relations also allows one to compare designs.

Experience with available FDTs shows that, although most of these FDTs help improving the correctness of designs by offering the possibility of early evaluation through simulation, verification or prototyping, they frustrate the designer by failing to formally represent essential characteristics of distributed systems due to their limited expressive power ([16]). This has been a motivation for this thesis.

Most of the ideas presented in this thesis have been originated from our experience with distributed system design and with the use of the FDT LOTOS ([2], [7]). This experience inspired us to devise a design model which is more general than the models imposed by the available design languages. This design model has been developed by identifying and studying the design concepts and their combinations which are necessary for distributed system design, without being restricted by any particular design language. However when we want to reason about design concepts we are forced to represent them using some notation. This notation can be considered as a sort of design language and is again vulnerable to limitations. This is actually a dilemma, which we have to acknowledge, but the use of general design model has brought an insight in structuring techniques and design operations that would probably not be possible otherwise.

1.2.3 Design Languages

Specification languages have been originated from the need to represent designs. However designs are abstractions of technical objects at specific abstraction levels. A *broad spectrum specification language* should support the representation of designs at many abstraction levels. Representing designs at different abstraction levels is however not enough to properly support the design process, since we should be able to *move* from one abstraction level to another. Therefore we need notions which determine whether a design corresponds to a more abstract design, allowing abstraction levels to be bridged. This notion is often called *implementation relation*.

Figure 1.3 depicts the relationships between design at different abstraction levels.

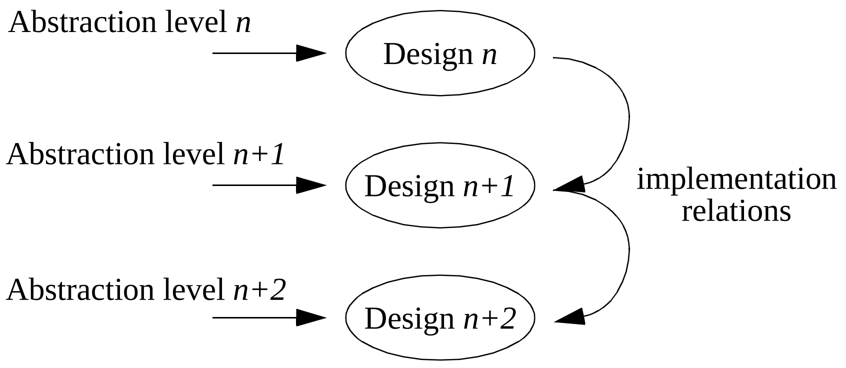


Figure 1.3: Designs at different abstraction levels

A specification language with a suitable collection of implementation relations and capable of representing designs at many abstraction levels is called a *design language*.

Effective design methodologies should define specific abstraction levels for the elaboration of designs. A design at a certain abstraction level has to be elaborated according to specific design objectives, which determine the role of the design in the context of the design methodology. Examples of such design objectives are qualitative design principles and manipulations of specific design concepts. Therefore design languages should also serve the qualitative objectives of some abstraction levels defined in a design methodology.

In addition to allowing specifications to be compared, the formal semantics of a design language also plays an important role in the implementation of a design. The formal semantics of a design language determines the interpretation of specifications in this language, therefore it is also the only information which can be used by an implementer, in a consistent and systematic way, to determine what has to be implemented from a specification. Informal annotations meant to add information to the interpretation of a specification should be avoided. In case a formal semantics has a restricted capability of distinguishing design options its applicability in supporting the design process is limited.

For example, the formal semantics of LOTOS maps both interleaving and independence of events onto interleaving. Technically correct implementations of specifications where interleaved events are defined should make these events interleaved, even if the designer actually meant that the events should be independent. This follows from the assumption that the only realiable information available to systematically determine what to implement from specifications is the interpretation of these specifications according to the formal semantics. This specific limitation of LOTOS restricts implementation freedom of implementers, by ruling out independent events, possibly resulting in low performance systems and bigger efforts for implementing these systems.

1.2.4 Design Supporting Tools

Another aspect of a design culture related to industrial competition is the use of *design supporting tools*. Formally sound design languages in combination with well-defined design methodologies allow the development of software tools for (partly or fully) automation of verification, transformation, simulation, etc. of designs.

Figure 1.4 depicts some elements of a design culture and their relationships.

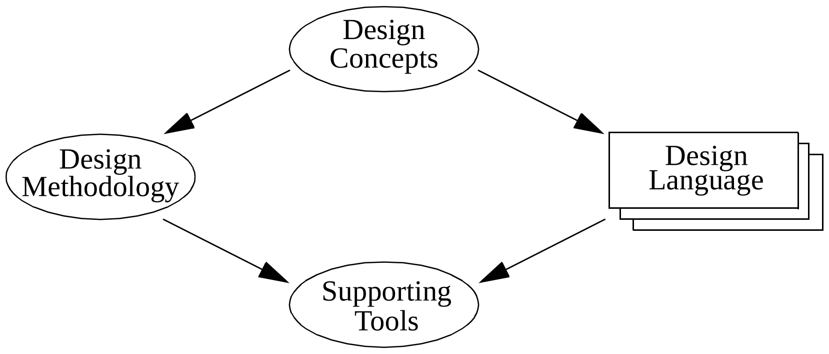


Figure 1.4: Elements of a design culture

1.2.5 Specification Styles

Designs may in principle be structured and formulated in many different ways by different designers, characterizing personal preferences or styles. However, a design methodology may recommend the use of well-defined specification styles, to be applied in different phases of a design trajectory, according to specific design objectives ([15]). Experience has shown that there are some benefits of using these specification styles, such as the improvement in the communication between designers, better understanding of specific design tasks and simplification of design manipulations. Specification styles can be derived from qualitative design principles and design objectives. Furthermore the design language must support the specification styles in its language elements.

Using the constraint-oriented style ([15]), the behaviour of a system is structured as a composition of constraints, each constraint represented by a certain (sub-)behaviour. This structuring technique is especially useful in the initial design steps, in which internal structure should not be revealed. LOTOS has been designed to support the development of constraint-oriented specifications, by including the concept of multi-way synchronization in its design model.

Figure 1.5 relates the basic elements of a design culture to a design and its specification.

1.2.6 OSI Design Cultures

In order to avoid being bound to specific computer manufacturers, the community of users of distributed information systems demanded that components produced by different companies should be able to work together. In this way the behaviour of the global system is guaranteed, according to protocol and service specifications agreed upon by the computer manufacturers. Systems that

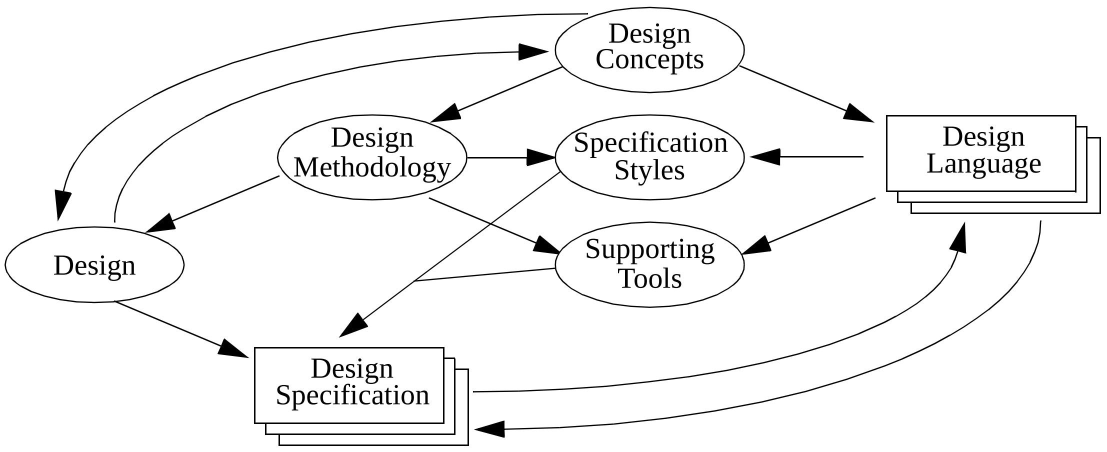


Figure 1.5: Application of a design culture

can be interconnected in this way are called *open systems*, and the design principle for their design is called *openness*.

Openness requires the development of standard solutions for the interactions between components of these open systems. These standard solutions should be defined in an implementation independent way, such that the real systems embedding these components still can be implemented and produced according to specific design cultures of the companies, making some room for competition. However these standard solutions are also designed, and therefore subject to a design culture, namely the design culture of the standardization bodies or working groups. This forces companies to adapt their design cultures to match the design culture of the standardization bodies. Some companies naturally try to influence the design culture of the standardization bodies in such a way that their investments in adaptations are minimal.

Standardization bodies such as ISO and CCITT have standardized many services and protocols of the Open System Interconnection Reference Model (OSI RM). Services and protocols are therefore the technical objects subject to design by these standardization bodies. Implementation freedom required by manufacturers impose that OSI services and protocols are defined in such a way that implementation details are not considered. Design concepts for service and protocol design are service primitive, service access point, connection end-point, connection, service element, service type, quality of service, protocol entity, etc.

The quality of the designs developed using these concepts is strongly dependent of a good common understanding of these concepts among the designers working in the standardization bodies. Furthermore, the quality of the products derived from these designs does not only depend on the quality of the designs themselves, but also on the clear understanding of their basic design concepts by designers and implementers in an industrial company.

(OSI) Formal Design Languages

Formal design languages to represent OSI design concepts should only have been developed when these design concepts were well understood and agreed upon. Practice shows, however, that two completely different interpretations of the OSI concept of service primitive led to the development of two completely incompatible types of standard specification languages: synchronous communication based, such as LOTOS, and asynchronous communication based, such as SDL and Estelle.

One could expect that the choice of design language should not interfere in the characteristics of the resulting designs, since design languages are merely alternative ways to represent the (abstractions of the) same technical objects. Nevertheless the choice of design concepts has influenced the choice of language elements for the design languages, which may influence the way specifications can be structured (specification styles), and may enable or disable specific design methods and strategies of a design methodology ([14]).

For example the constraint-oriented specification style defined in [15] is possible in LOTOS, and completely impossible in SDL and Estelle. Constraint-oriented and resource-oriented specification styles can support a design approach towards service and protocol design, such that the local constraints of the service are kept as local abstract interfaces of the protocol entities, and only the remote interfaces of the service are decomposed in terms of the protocol functions and underlying service. The inability of SDL and Estelle to support the constraint-oriented specification style automatically disables the use of this design approach.

Maturity of OSI Design Cultures

SDL and Estelle have been developed using rather traditional architectural concepts and semantic models for distributed systems design (asynchronous interactions and synchronous state machines), for which a lot of supporting implementation tools and methodologies were available. One could say that the design culture for this kind of modelling is mature, since it has been used for around two decades.

LOTOS is one of the first industrially applicable design languages developed using synchronous interactions and asynchronous labelled transition systems as basic architectural concept and underlying formal semantics model respectively. This means that although LOTOS has been a milestone in the formal representation of designs and implementation notions at a high-level of abstraction, still a lot can be done, for instance, to improve its expressive power and enhance its industrial applicability. One could say that the design culture for this kind of modelling is promising, but it is not yet established.

This thesis can be seen as a contribution in the design culture of synchronous interactions. Synchronous interactions form in this thesis the starting point for the development of a framework of structuring techniques and design methods. Unlike LOTOS, our design model for behaviour definition is based on causality between actions, and therefore it should be not adequate for representing parallel or concurrent behaviours.

1.3 Implementation Aspects

A specification that represents most of the design decisions that can be expressed using the design model supported by an abstract design language is called a *final specification*. A final specification has to be translated to an implementation, such that the design process can be accomplished.

Some aspects that influence the elaboration of final specifications and their translations to implementations are discussed in this section.

1.3.1 Target Implementation Environment

The elaboration of specifications should make use of some knowledge about the available implementation components. Especially at the end of the design trajectory the availability of this knowledge becomes rather critical. Knowledge about available implementation components is often called *bottom-up knowledge*.

We define an implementation environment as the set of available resources, in hardware or software, that can be used to construct an implementation. In software design these resources are normally programming languages and operating systems, debuggers, compilers, etc. In hardware design these are VLSI components, boards, buses, etc.

A *target implementation environment* is an implementation environment imposed by the user requirements. The requirement of a target implementation environment is guided by a combination of technical and sometimes strategical or even political arguments. For example the choice of a specific workstation type, programming language and operating system as the target implementation environment can be guided by their technical quality, but also e.g. by their cheaper price when compared to others, or due to the fact that the technical staff are used to them, so that changing them will have an impact on the staff productivity in short term.

1.3.2 Abstraction Levels

Broad spectrum design languages are able to cover different abstraction levels, but they are not (yet) able to replace implementation languages. Implementation languages are capable of representing other technical aspects of distributed information systems which cannot be represented using these design languages, but that must be represented somehow. Furthermore implementation languages can normally be directly translated into a running model of the system, which is not supported by any available broad spectrum design language.

In software implementations of (parts of) distributed systems, for example, we have to cope with allocation of memory, buffer management, coding of information, allocation of communication bandwidth, concrete interfaces, etc. at some relatively low abstraction level. Programming languages and operating system facilities are able to represent these aspects. In hardware implementations, for example, we have to cope with gate delays, fan in and fan out of components, interfacing, etc. Hardware design languages are able to represent these aspects.

The elements of a design culture are present at all abstraction levels along the design process, so that patterns presented in Figures 1.4 and 1.5 apply to multiple different abstraction levels. At higher abstraction levels we may have different design concepts, design languages, specification styles and supporting tools than the ones we have at lower abstraction levels, leading to different forms of designs and design specifications.

Correct and effective implementation implies that there must be a relationship between the elements of the design culture at high and low abstraction levels. We concentrate on the design, its specification, the design language and the design concepts. We call the design concepts at higher and at lower abstraction levels *architecture concepts* and *implementation concepts*, respectively. Implementation concepts are those concepts manipulated by implementation environments. An implementation environment consists of implementation languages, operating systems and supporting tools.

In the sequel, a design at a higher and at a lower abstraction levels are called an *architecture* and an *implementation*, respectively. The design language to express an architecture is called an *abstract design language*, and the representation of an architecture is called an *architecture specification*. Similarly the design language to express an implementation is called an *implementation language*, and the representation of an implementation is called an *implementation specification*.

Figure 1.6 depicts the relationship between these abstraction levels.

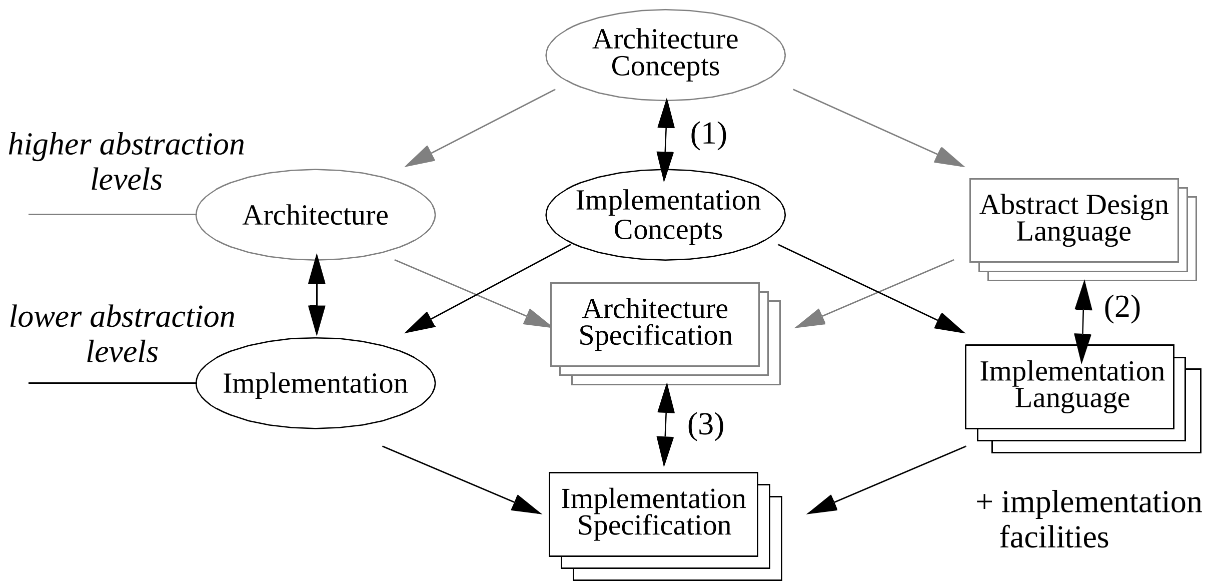


Figure 1.6: Design and specification at different abstraction levels

The following aspects are expected to play an important role in the development of correct implementations from architectures:

1. correspondence between architecture concepts and implementation concepts;
2. implementation languages or implementation environment facilities or both should be adapted or newly developed in order to match those language elements of the abstract design language;
3. mappings from specification constructs onto implementation constructs have to be defined, in terms of implementation notions and transformations.

These aspects are indicated beside the bi-directional arrows between the higher and lower abstraction levels in Figure 1.6.

One must be careful when considering the mechanisms to derive an implementation from a specification. In other engineering areas we normally do not expect to derive the implementation automatically from requirement specifications. Like it is stated in [8], it makes no sense to expect that a bridge can be automatically derived from the description of the river and the expected traffic. This means that lots of well-considered design decisions have to be taken, until all the physical characteristics of the bridge are selected and verified (simulated) with respect to the requirements. Only at this point it is conceivable to connect design information with the production line, so that pieces of the bridge are produced by (computer) automated machines. Making the analogy with the design of distributed information systems, the abstraction level at which some form of 'automatic translation' is possible has to be bridged with human generated design decisions, and cannot be totally automated as some people might expect.

Difference of Design Models

The abstract design model, which is the set of architecture concepts and their combination rules, is defined at a high level of abstraction with respect to implementation details. Therefore architecture concepts used for the elaboration of a final specification may differ from the concepts available in the implementation environment. This difference can be accommodated, such that for each concept represented in a final specification there is a corresponding (combinations of) concept(s) in the implementation environment.

There is a relationship between the complexity of the translation of a final specification to an implementation, and the degree of support of architecture concepts in the implementation environment. The choice of a target implementation environment may have a very strong influence on the implementation strategy to be taken.

Generally one could think of two ways of approaching the translation of a final specification to an implementation with respect to the architecture and implementation design models:

1. restrict the use of design concepts in the elaboration of the final specification to only those concepts that can be directly found in the target implementation environment;
2. enhance the capabilities of the implementation environment, so that a set of key architectural concepts are already implemented and available in the implementation environment, making the translation straightforward.

A translation from a final specification to an implementation is complete when all abstract design concepts and their combinations have a counterpart in the implementation design concepts and in their combinations. A combination of both approaches identified before seems to be the most effective way to handle this translation problem.

Figure 1.7 depicts the design freedom in the elaboration of a final specification. Figure 1.7 also shows that there must be some room left for structural (architectural) design, since the enhancement of implementation environments is limited by the need to define system structure.

A communication service is a typical example of technical object that has to endure architectural design before being implemented. Once a designer elaborates a service description, this has to be refined in terms of a distributed perspective, namely a protocol, in order to be implemented. Implementing the service specification directly may result in systems that do not comply to

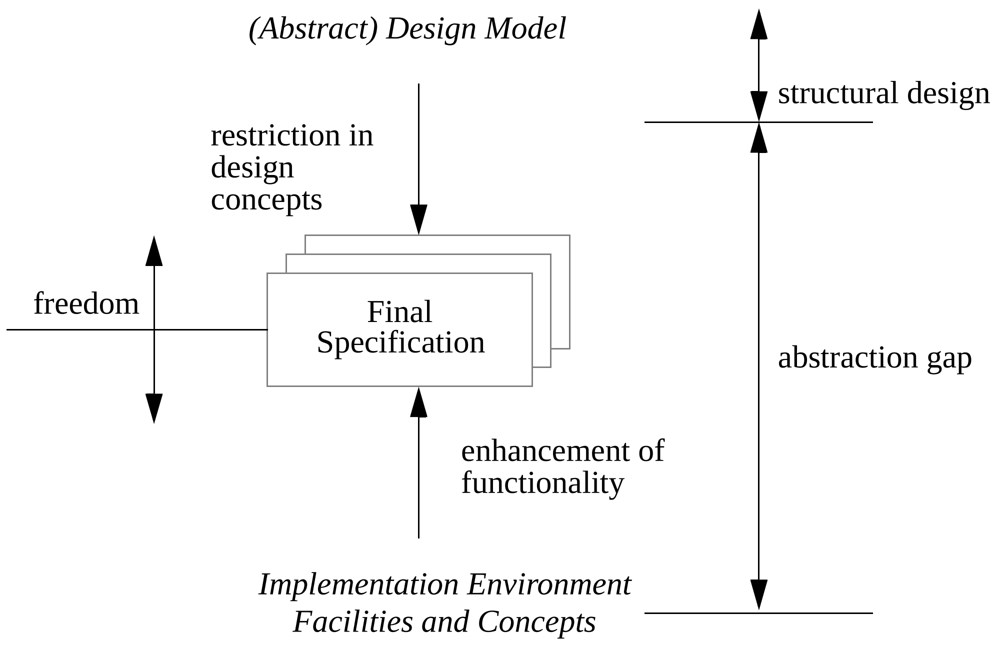

(Abstract) Design Model
restriction in design concepts
structural design
freedom
Final Specification
abstraction gap
enhancement of functionality
Implementation Environment Facilities and Concepts

Figure 1.7: Aspects that influence the elaboration of a final specification

requirements of geographical distribution. Such requirements are normally fulfilled by the structural design of the system.

1.3.3 Refinements and Compilation

The bridging of the design gap between a specification and an implementation can be done in principle by using two extremes in the range of implementation strategies: (i) design refinements and (ii) compilation.

Firstly we suppose the exhaustive use of refinements in the elaboration of implementations. In this case we would possibly obtain good results in non-functional characteristics (performance and resources usage), but the design process could take too long, especially when most of the design refinements are not automated, which is the state-of-art today. Furthermore severe version control mechanisms are needed. Propagation of modifications throughout all intermediate designs could be another serious problem.

Secondly we suppose a minimal use of refinements, and the use of a compiler as soon as the components to be implemented in embedded systems are identified. In this case we expect that an implementation would be reached quite quickly, and that modifications can be made quite easily, but the non-functional characteristics may be very poor. Therefore this extreme approach should only be used for *early prototyping*.

A combination of these two approaches seems to be the most effective way to handle the problem.

Figure 1.8 depicts these two approaches by only considering the final intermediate designs that can be obtained, such that backtracking can be ignored.

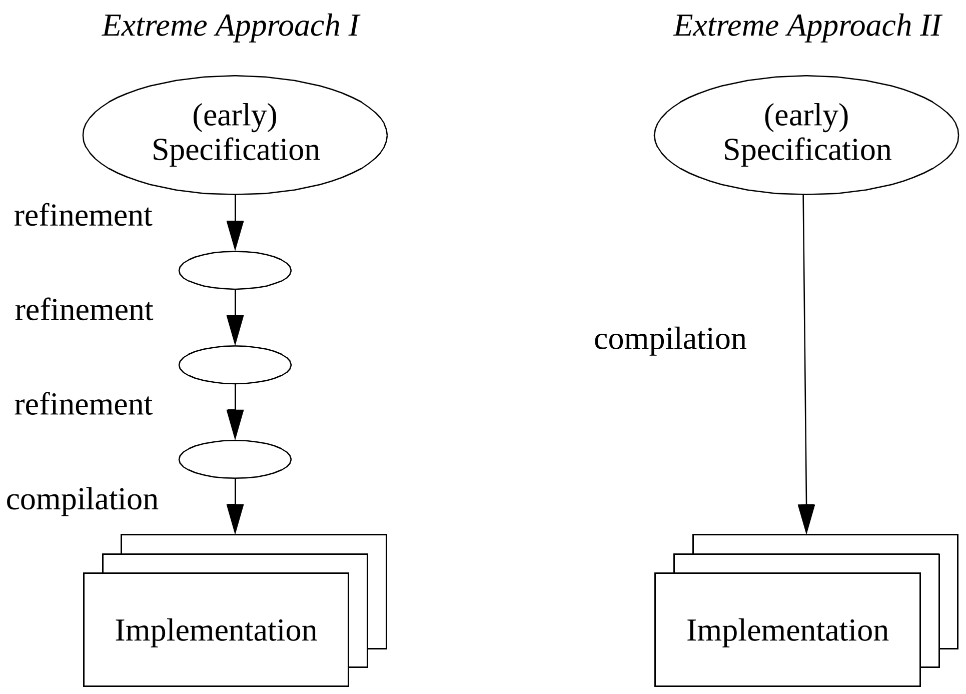


Figure 1.8: Refinement against compilation

Compilation, either manually (hand-coding) or automated, can hardly be avoided. The combination of these approaches should aim at increasing the automation on the refinements and reducing the number of refinements, without making them too complex.

1.3.4 Implementation Concerns

Implementation concerns are concrete problems which have to be solved in order to translate a final specification into a final implementation. These problems are mentioned here in order to help understanding how the different implementation strategies handle the translation from a specification to implementation environment concepts.

Data Types

Data type definitions in a specification should represent the data sorts, operations and relationships between these operations (e.g. in terms of equations), abstracting from the specific ways on how to implement them. Abstract data types (ADTs) support this kind of definition. However, ADTs defined in specifications should be also translated to implementation constructs when an implementation has to be obtained. There are basically two techniques for implementing ADTs: (i) application of rewrite rules or (ii) by hand-coding.

The use of rewrite rules can be automated, but normally it results in poor performance and inefficient allocation of memory area. Hand-coding can be very time consuming, but it potentially results in better performance and more efficient allocation of memory. The correctness of implementations using rewrite rules is guaranteed by the translation algorithms. Hand-coding requires testing, which in most cases does not guarantee correctness.

Rewrite-rules are normally used in simulators and prototypes, while hand-coded implementations of ADTs are used in realistic implementations. A good discipline towards the implementation of ADTs is to define and use libraries of ADT specifications and implementations, combined with structuring disciplines such as e.g. encapsulation. Object-oriented approaches towards ADT implementation also seem promising.

ADT implementation is only marginally addressed in this work. We assume that for each ADT definition containing sorts and operations related by equations, there is a corresponding software implementation, such that for each ADT sort there is a concrete data type in the software implementation, and for each ADT operation there is a concrete value or a procedure in the software implementation. ADT equations are used as requirements for the implementations of the procedures, such that they are correct implementations of the ADT operations. This pragmatic approach towards ADT implementation has been chosen, since the subject of ADT implementation is a research area in itself. A more precise approach towards ADT implementation can be found, for example, in [10].

Concurrent and Cooperating Functional Entities

Functional entities in the design model developed in this thesis can be seen as cooperating and concurrent units of processing. A correct implementation of these functional entities implies that either the implementation environment supports the creation and deletion of concurrent instances of processing, or the concurrence in the model has to be destroyed in order to allow a mapping onto sequential instructions.

In case concurrent instances of processing are supported by the target implementation environment, one could consider the actual mapping onto processors. In multiprocessor systems, different functional entities could be assigned to different processors. In mono-processor systems, concurrence has to be simulated by process management components, which define a transparent time-sharing of the processing capabilities (pseudo-parallelism) between the different functional entities.

Another possibility is to implement functional entities as tasks (sometimes called processes) of an operating system, and to use inter-task communication mechanisms to implement the interactions between these functional entities. Available concurrence kernels, such as the light weight processes library for Unix, can also be used for this purpose.

Interactions

Interactions model synchronous and reliable communication between (possibly multiple) functional entities. This means that either synchronous and reliable communication facilities are available in the implementation environment, or they have to be simulated. Functional entities may even have multi-way synchronization, which is not directly supported by any programming language commercially available now-a-days. This means that either multi-way synchronization is removed from a design through design refinements, or mechanisms (algorithms) to implement multi-way synchronization have to be implemented and made available in the implementation environment. Synchronization kernels can be designed in order to provide such synchronous communication capabilities.

In principle three forms of data exchange are possible between functional entities: value passing, value checking and value negotiation. The first two forms of data exchange can be compared to classic cases of interaction supported by many implementation environments, such as input/output and pure synchronization. Value negotiation however is a rather sophisticated model of data exchange, which is not directly supported by any commercial implementation environment. Value negotiation is often used to postpone implementation decisions; in most cases value negotiation is removed beforehand by taking explicit design decisions on how the values are established, and therefore it should not appear in a final specification.

Interactions between functional entities are symmetrical, which means that the concept of initiative for an interaction makes no sense at the abstraction level where the interaction is defined. However, many available implementations of synchronous communication are asymmetrical mechanisms, in which one of the functional entities has to take the initiative. In this case, rules to determine the roles of the functional entities in the synchronization have be defined. In our framework this can be done if we consider the interaction at a lower abstraction level, i.e. in terms of its implementation, where the initiative for interaction can be explicitly defined in the mechanism that implements the interaction.

Figure 1.9 summarizes the aspects to be considered in the elaboration of implementations.

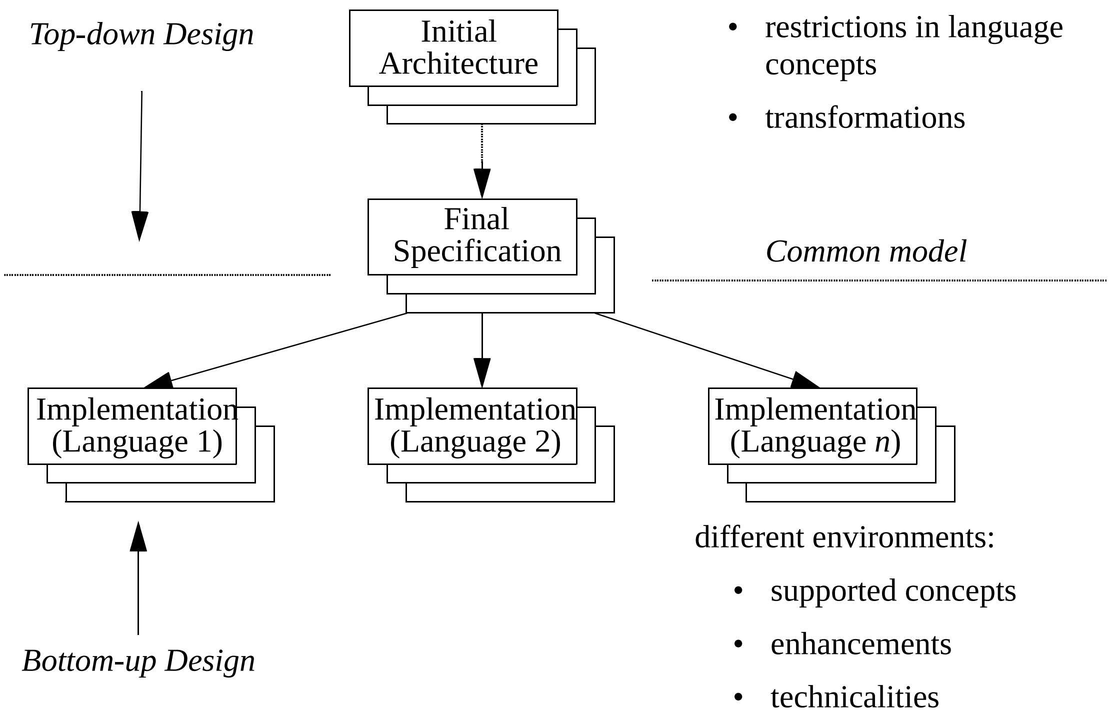


Figure 1.9: Aspects of implementation

The objective of defining models which are commonly used to enhance implementation environments, to define abstract design language constructs and to elaborate specifications is to allow the development of design and implementation support for various different implementation environments.

1.3.5 Pre-defined Implementation Constructs

An approach towards implementation of specifications of large distributed systems is the use of *pre-defined implementation constructs* ([4]). This approach consists of transforming specifications as soon as possible into a composition of pre-defined implementation constructs (PDICs). A PDIC is a general-purpose, possibly parameterized construct, which is represented by its specification and its corresponding implementation(s) in the target implementation environment.

Using this approach, designers are encouraged to develop a library of PDICs, such that many different specifications can be implemented using the entries of this library. The correctness of PDIC implementations is established before they are used, and the correctness of the final implementation becomes exclusively dependent of the correctness of the composition of PDICs.

Figure 1.10 depicts this approach.

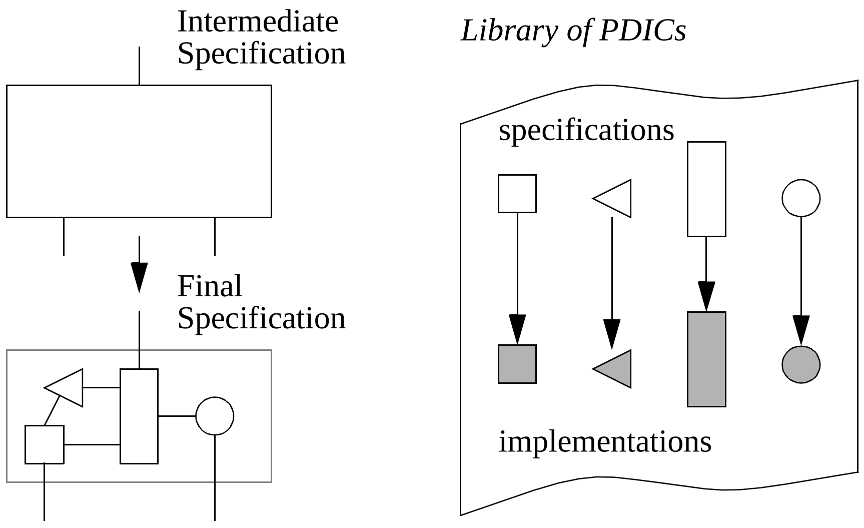


Figure 1.10: Pre-defined implementation constructs approach

The use of PDICs brings a clear distinction between structural (architectural) design and the technicalities inherent to different implementation environments. It also potentially reduces validation effort and shortens the design life cycle of individual distributed systems. Re-usability of components is also favoured by this approach.

1.4 Objectives

Globally the objective of this thesis is to discuss and define design concepts, techniques and methods that are necessary for supporting the design process of distributed systems in a systematic way. These concepts, techniques and methods form what we call a *design methodology.*

A design methodology that is based on step-wise refinement has to define a handful of design phases based on precise design objectives. Each of these design phases consists of design steps, and results in a *design milestone*. The design methodology should also define the methods to perform design steps, in terms of design objectives and design operations. Design operations provide the technical guidance to the elaboration of a design step, by defining the manipulations of design concepts and the correctness notions for the design step.

The quite general objective of defining a design methodology can be split into the following sub-objectives:

- definition of a collection of design milestones aimed at moving from abstract designs to more concrete implementations;
- identification of the design concepts necessary to represent the design milestones;
- identification of design operations to move between design milestones;
- development of concrete guidance for performing design operations and assessing their correctness;
- design of a pre-defined implementation construct to directly support the multi-way communication between functional entities.

Non-objectives

This thesis does not intend to solve all problems in the area of design and implementation of distributed information systems. We rather aim at developing solutions to be applied to some relevant design problems. In this thesis some design concepts are manipulated throughout different abstraction levels in order to derive the solutions in a consistent way.

We do not aim at evaluating any specific design language in this thesis either. Some of the concepts introduced here are compared to their corresponding representation or lack of representation in LOTOS. In this way we provide some reference to the readers that are acquainted with LOTOS, as an extra illustration for our ideas. LOTOS is used for this purpose since it is the only design language based on synchronous interactions which also supports multi-way synchronization and data type representation.

1.5 Strategy

A design methodology for distributed systems has been presented in [12] and improved in [1] and [9]. The development of this design methodology started by considering a system from an integrated and from a distributed perspective. In the distributed perspective a system is considered as a composition of interacting parts. The mechanism that makes the interaction between parts possible is called an *interaction system* ([12]). A *service* corresponds to an interaction system from an integrated perspective and a *protocol* corresponds to an interaction system from a distributed perspective.

The concepts of interaction and interaction points have been introduced in order to enable the definition of behaviours (functions) of systems and system’s parts. An event is defined as the most elementary form of interaction, such that behaviours can be defined in terms of events and their possible temporal ordering. Processes are abstractions of behaviours, which allows one to instantiate possibly parameterized behaviours. Behaviour composition operators have also been defined, allowing behaviours to be structured, which enhances the design overview and controllability. The FDT LOTOS is a design language that has been developed to support the representation of events, behaviours, processes and behaviour compositions. Issues related to the development of a design language have been addressed in [3].

Figure 1.11 depicts the steps followed in the development of the design methodology presented in [12].

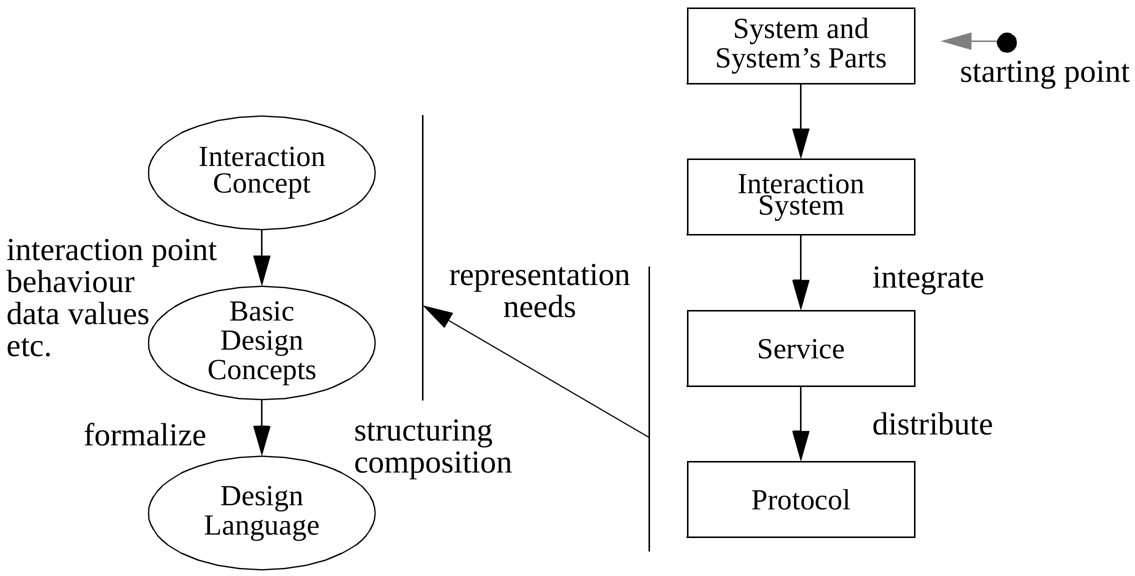


Figure 1.11: The development of a design methodology

The strategy adopted in this thesis considers the concept of interaction as a starting point. In [9] it has been pointed out that at the abstraction level where interactions are considered one cannot properly represent some important behavioural aspects, such as, for example, timing and probability requirements. Similar observations have been reported in [6] and [16]. Therefore we introduce the concept of *action*, as a representation of an interaction which abstracts from specific distributions of responsibilities amongst the functional entities that participate in the interaction. We conclude that a framework to support realistic instances of design for distributed systems should be able to represent and manipulate both actions and interactions.

Behaviours are represented in terms of causality relations involving actions and interactions. However, behaviour representation techniques are not enough for distributed systems design, since one should also be able to reason about the actors of behaviour, namely the functional entities. In this way two inter-related domains for distributed systems development are identified in this thesis: the *entity domain* and the *behaviour domain*. The entity domain concerns the functional entities and their interconnections, while the behaviour domain concerns the representation and structuring of behaviours of functional entities. Entity and behaviour domains serve as the basis for an implementation strategy, in terms of design steps and their objectives, and for the development of design languages.

Figure 1.12 depicts the strategy adopted in this thesis.

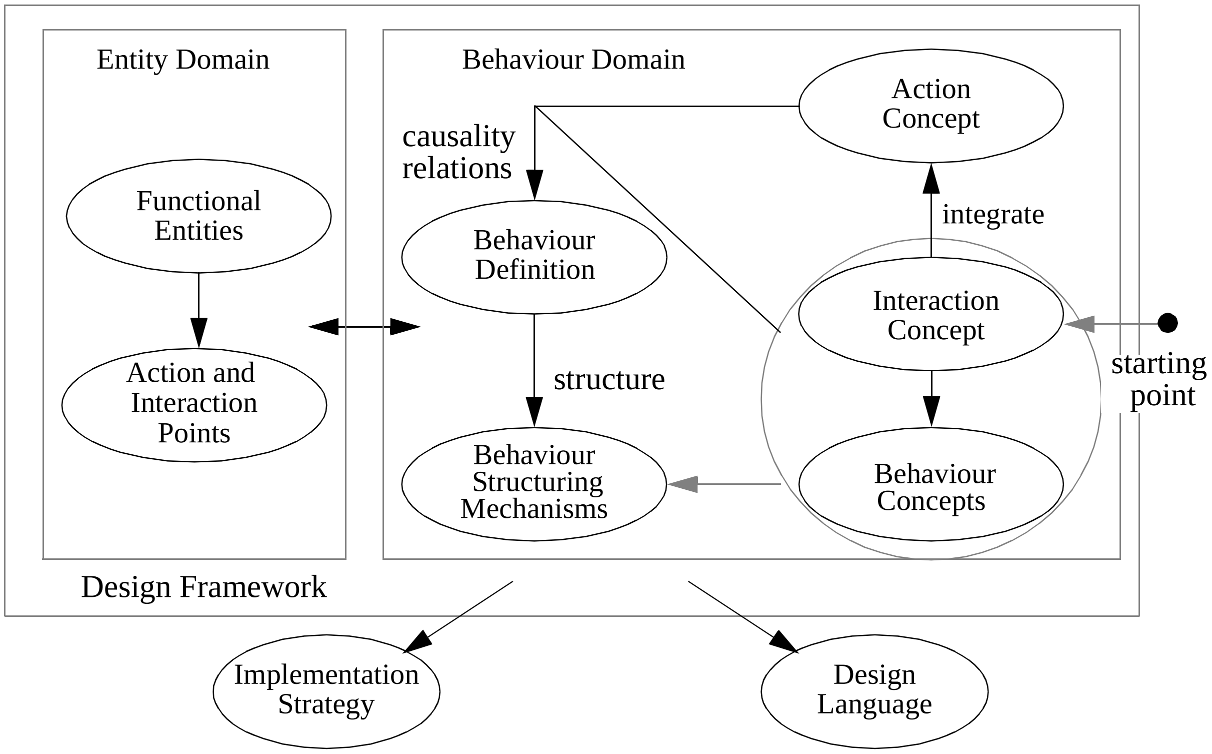


Figure 1.12: Strategy applied in this thesis

The design model developed in this thesis should be supported by a proper design language. Due to our objectives of searching for adequate design concepts and investigating their manipulations during the design process, we have deliberately decided not to dive into the intricacy of language design, which falls outside the scope of our expertise. In this way we have addressed structuring techniques and methods for design operations from the point of view of their essential characteristics, providing enough requirements for the development of effective design languages and tool support.

This thesis is expected to be a useful reference for designers, including those who lack mathematical training. This justifies our choice for an informal and intuitive presentation, as much as possible illustrated by diagrams and examples. The formalization of our ideas in terms of a mathematical theory does not constitute a primordial goal of this work.

1.6 References

[1] K. Bogaards. *A Methodology for the Architectural Design of Open Distributed Systems*. PhD thesis, University of Twente, Enschede, Netherlands, 1990.

[2] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems 14*, pages 25–59, 1987.

[3] E. Brinksma. *On the design of Extended LOTOS*. PhD thesis, University of Twente, Enschede, Netherlands, 1988.

[4] L. Ferreira Pires, M. van Sinderen, and C. A. Vissers. On the use of pre-defined im-

plementation constructs in distributed systems design. In *Third IEEE Workshop on Future Trends of Distributed Computing Systems in the 1990's*, pages 114–120. IEEE Computer Society Press, 1992.

[5] L. Ferreira Pires and C. A. Vissers. Overview of the Lotosphere design methodology. In Commission of the European Communities, editor, *ESPRIT 1990, Conference Proceedings*, pages 371–387. Kluwer Academic Publishers, 1990.

[6] L. Ferreira Pires, C. A. Vissers, and M. van Sinderen. Advances in architectural concepts to support distributed systems design. In *11o Simpósio Brasileiro de Redes de Computadores. Tutoriais e Minicursos*, Campinas, Brazil, 1993. Departamento de Engenharia de Computação. Faculdade de Engenharia Elétrica. Universidade Estadual de Campinas. also available as: Memorandum Informatica 93-17 (TIOS 93-09), University of Twente, Enschede, the Netherlands.

[7] ISO. Information processing systems – Open Systems Interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour, 1989. IS 8807.

[8] D. L. Parnas. Mathematics of computation for (software and other) engineers. In *AMAST'93 - Participant's proceedings*, pages 11–21, Enschede, the Netherlands, 1993. University of Twente.

[9] J. Schot. *The role of Architectural Semantics in the formal approach towards Distributed Systems Design*. PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[10] M. Thomas. Implementing algebraically specified abstract data types in imperative programming language. In *Proceedings of TAPSOFT'87 Colloquium on Functional and Logic Programming and Specification (CFLP)*, pages 197–211, March 1987.

[11] K. Turner. An architectural semantics for LOTOS. In H. Rudin and C. West, editors, *Protocol Specification, Testing and Verification, VII*, The Netherlands, 1987. Elsevier Science Publisher B.V. (North-Holland).

[12] C. A. Vissers, L. Ferreira Pires, and D. A. Quartel. *The Design of Telematic Systems*. University of Twente, Enschede, the Netherlands, Nov. 1993. Lecture Notes.

[13] C. A. Vissers, L. Ferreira Pires, and J. van de Lagemaat. Lotosphere, an attempt towards a design culture. In T. Bolognesi, E. Brinksma, and C. A. Vissers, editors, *Third Lotosphere Workshop and Seminar, Workshop Proceedings*, volume 1, pages 1–30, 1992.

[14] C. A. Vissers and G. Scollo. Formal specification of OSI. In G. Muller and R. Blanc, editors, *International Seminar on Networking in Open Systems*, volume 248 of *Lecture Notes in Computer Science*, pages 338–359. Springer-Verlag, 1987.

[15] C. A. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma. Specification styles in distributed systems design and verification. *Theoretical Computer Science*, 89:179–206, 1991.

[16] C. A. Vissers, M. van Sinderen, and L. Ferreira Pires. What makes industries believe in formal methods. In A. Danthine, G. Leduc, and P. Wolper, editors, *Protocol Spec-*

ification, Testing, and Verification, XIII, pages 3–26, The Netherlands, 1993. Elsevier Science Publishers B.V. (North-Holland).

Chapter 2

Overview of the Design Methodology

This chapter presents an overview of the design methodology which is developed in this thesis, by identifying global abstraction levels at which a distributed system or system part can be considered. The purpose of identifying these global abstraction levels is to recognize the basic architectural design concepts necessary to represent designs at these levels, and to provide concrete guidance to designers for the definition of design milestones, which are those well-defined intermediate designs that shall be produced before a concrete implementation in terms of software and hardware can be reached.

The structure of this chapter is as follows: section 2.1 introduces some basic definitions in a tutorial form, section 2.2 identifies design abstraction levels, section 2.3 discusses the objectives of defining actions in the design process, section 2.4 discusses global design milestones and design steps in the design process and section 2.5 illustrates these design milestones by means of simplified examples.

2.1 Basic Definitions

An effective design methodology should be defined in terms of design objectives and design choices, to be applied at each consecutive design step. These design objectives and choices determine milestones to be achieved during a certain instance of a design process. Abstraction allows a designer to focus on some aspects of interest, while ignoring other aspects. Well-defined abstraction levels determine global sequences of aspects to be considered, providing guidance to designers when performing instances of the design process.

When defining a design methodology, we have to carefully consider the level of detail in which its milestones are defined. Too detailed global design objectives and choices may restrict the applicability of the methodology; too general global design objective and choices may make the methodology useless for realistic applications. We conclude that there must be a certain compromise between the degree of freedom allowed by the methodology and the amount of guidance it provides.

A realistic design methodology should define global objectives and choices in a more concrete way than for example the waterfall model (e.g. [1], [2]) in which design, implementation, realization, testing, production, maintenance subsequent phases are identified. Although the generic guidelines presented in the waterfall model are still valid, designers also need to know what specific aspects should be addressed in what order. Since the waterfall model is applicable for generic information systems, a design methodology to distributed information systems aiming at tele-informatics applications should also consider some more specific aspects, such as the logical and geographical distribution of interactions, the logical and geographical distribution of components, the use of available communication infra-structure, etc.

2.1.1 Entity Domain and Behaviour Domain

The design process of distributed systems globally consists of the manipulation of basic design concepts. In most approaches towards the design of distributed systems we can recognize the existence of the following basic design concepts (see for example [4]):

- *functional entity*: a logical and physical part of a system;
- *action*: unit of activity performed by a functional entity;
- *interaction*: common actions shared by two or more functional entities, through which cooperation between functional entities for the purpose of establishing and exchanging information can be defined;
- *action point* and *interaction point*: logical or physical locations at which actions and interactions occur, respectively.

Functional Entities

We use the term functional entity to denote a logical or physical system part in this text, since this term does not imply any explicit relationship with concrete pieces of a system that the term part may suggest. Each functional entity is uniquely identified, such that we can unambiguously refer to them. Functional entities are interconnected by uniquely identified interaction points and may restrain uniquely identified action points.

The following rules apply to functional entities, interaction points and action points:

- each functional entity is delimited by zero or more interaction points, and each interaction point is shared by two or more functional entities;
- each functional entity may restrain zero or more action points, and each action point is restrained by a single functional entity;
- each functional entity is delimited by at least one interaction point or restrain at least one action point.

Functional entities with zero interaction points and zero actions points are of no practical use and therefore are not allowed.

Figure 2.1 depicts some examples of valid functional entities.

In Figure 2.1 functional entity F_1 share interaction points ip_1, ip_2 and ip_3 with its environment, which is considered to be another functional entity. Functional entity F_2 has no interaction points and functional entity F_3 has no action points.

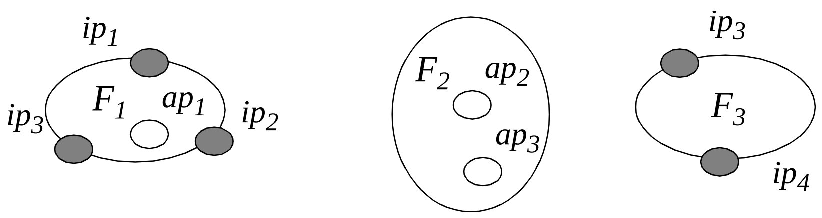


Figure 2.1: Some valid functional entities

Figure 2.2 depicts compositions of functional entities.

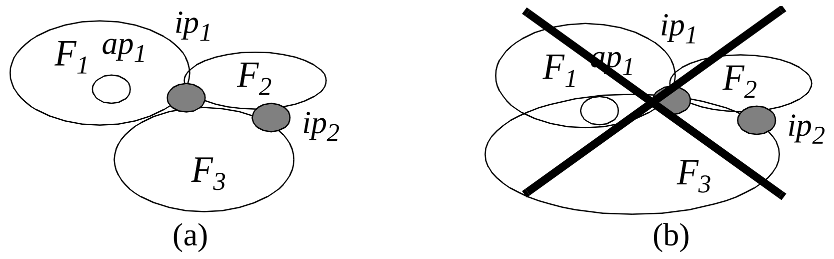


Figure 2.2: Compositions of functional entities

In Figure 2.2, composition (a) is a valid one, since it complies to the rules presented before. However, composition (b) does not comply to the rule that each action point is restrained by a single functional entity.

Behaviours

An interaction point represents the minimum necessary condition for specific functional entities to interact, being the *means of interaction* between these functional entities. Functional entities sharing interaction points do not necessarily interact. Actual interaction is determined by the interaction 'contents' (interaction semantics) and by the specific states at which the functional entities are found at some specific moment of time (behaviour).

The 'function' of a functional entity is defined in terms of its *behaviour*, i.e. the temporal ordering of its the possible actions and interactions, and their relationships (e.g. value dependencies, timing conditions). A behaviour of a functional entity should be defined in terms of conditions and constraints imposed by this functional entity on its actions and interactions. Behaviours and behaviour definitions are treated in more detail in Chapter 3.

The Framework

Considering the design concepts above we can identify two distinct but related domains for system description:

1. the *entity domain*, in which the actors of behaviour, i.e. the functional entities, and their compositions are defined, and
2. the *behaviour domain*, in which the behaviours of the functional entities are defined.

Actions and interactions occur at action points and interaction points respectively, which makes it possible to relate a certain composition of functional entities with their behaviours.

A similar framework for distributed system design has been proposed in [4].

Figure 2.3 depicts these two domains and their related concepts.

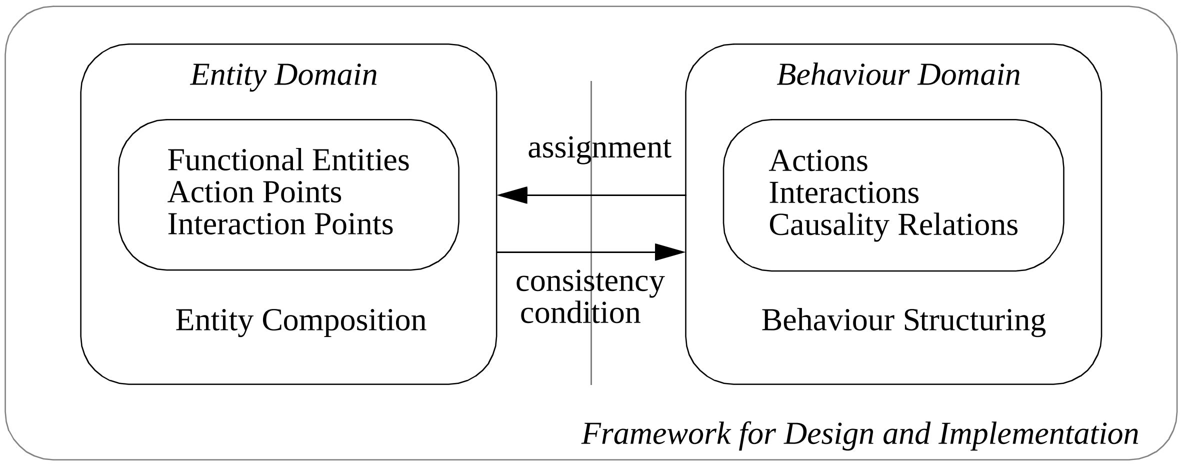


Figure 2.3: Entity and behaviour domains

In the entity domain we consider aspects related to the structure of functional entities. These aspects involve the identification of functional entities and the representation of their interconnection structure. In the behaviour domain we consider aspects related to the representation of behaviours, in terms of actions and interactions and their relationships. Behaviours, especially complex ones, have to be structured, making behaviour structuring techniques necessary.

We consider behaviour from a prescriptive point of view in this text, i.e. to be interpreted as prescriptions to the implementors of functional entities on how to build them.

The entity and behaviour domains are related to each other by an assignment and a consistency condition. Each functional entity has a behaviour assigned to it and consequently actions and interactions are assigned to action points and interaction points. The consistency condition imposes that given a certain assignment of behaviours to functional entities:

1. actions of a behaviour happen at action points restrained by the functional entity to which this behaviour is assigned;
2. interactions of a behaviour happen at interaction points which are shared by the functional entity to which this behaviour is assigned. This means that interactions between functional entities can only occur at interaction points that these functional entities share, i.e. through which these functional entities are interconnected.

According to these rules a functional entity forms a scope in which its behaviour can be defined. Actions and interactions can only be related to each other in the behaviour of a functional entity if their action points and interaction points are restrained or delimit this functional entity.

Figure 2.4 illustrates the entity domain and the behaviour domain and their relationships for a generic composition of functional entities F_1, F_2, F_3 and F_4.

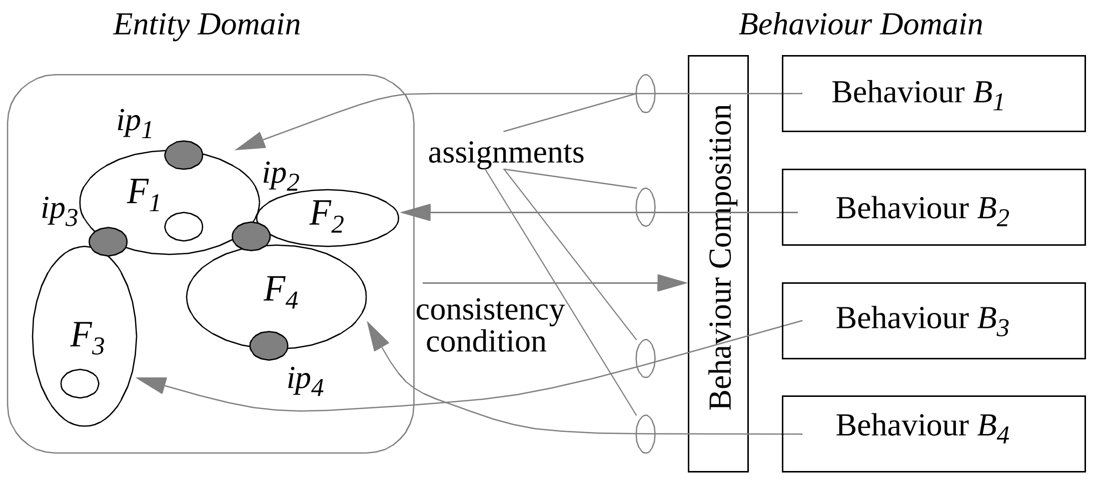


Figure 2.4: Illustration of entity and behaviour domains

In Figure 2.4, behaviours B_1, B_2, B_3 and B_4 are assigned to F_1, F_2, F_3 and F_4, respectively. The composition of behaviours B_1, B_2, B_3 and B_4 should comply to the consistency condition imposed by the composition of functional entities F_1, F_2, F_3 and F_4. In Figure 2.4, for example, interactions shared by B_1 and B_3 can only occur at interaction point ip_3.

2.1.2 Framework for Design Steps

In the design process, design decisions taken during a design step must refine a design through the manipulation of its basic design concepts. Most design steps can, therefore, be characterized by some manipulation in the entity domain, in terms of some modification of the structure of functional entities or interaction points, and some manipulation in the behaviour domain, in terms of some modification of the behaviours and their assignment to functional entities and interaction points.

Specific design objectives and choices should not be in conflict with the global design objectives and choices dictated by the design methodology. In the first design step no actual manipulation takes place, since in this step the first design of the system should be elaborated. However design objectives and choices are already considered in this step.

In most cases only behaviours are represented using design languages, leaving the mapping of behaviours onto functional entities to intuition. Design specifications can sometimes be manipulated according to specific design objectives, characterizing *specification transformations*. Using formal design languages one could in principle provide some guarantee on the correctness of these transformations. However, care must be taken when defining manipulation of specifications to perform design steps. A typical mistake made by formalists is to consider that these transformations can always be performed by means of automated algorithms, making the designer superfluous in the transformation process. This normally results in transformations of very limited practical use, since automated algorithms cannot substitute a designer in considering design objectives and making design choices. A survey of protocol synthesis methods is presented in [7], which also comes to the conclusion that protocol synthesis methods are still rather immature to support complex design instances.

The framework introduced in this text allows designers to define design objectives and choices at both entity and behaviour domains. Some important design steps are defined from the entity domain, such as functionality (functional entity) decomposition, interaction point refinement, etc. Although the objectives of these design steps are originated from manipulations of elements of the entity domain, viz. functional entities and interaction points, there are some implications at the behaviour domain for performing these design steps. These implications can be defined in terms of correctness criteria, and are further discussed in different parts of this work. Some other design steps are defined in terms of manipulations solely in the behaviour domain, keeping the structure of functional entities and interaction points intact. Examples are reduction of non-determinism, behaviour reduction or extension, etc.

2.2 Design Abstraction Levels

Different aspects of system design can be considered at different abstraction levels. These different abstraction levels and their relationships can be used to define global design objectives. Design concepts necessary to represent designs at these abstraction levels and design steps necessary to move from one abstraction level to another can also be identified, providing a coherent framework and guidelines for the specification and implementation of distributed systems.

2.2.1 Distributed Perspective of a System

In general a system can be viewed as a composition of interacting parts, also called functional entities, whereas the system as a whole interacts with its environment. The representation of a system viewed in terms of its composition of functional entities is called the *distributed perspective* ([9]).

Functional entities can be viewed at different abstraction levels. A functional entity can be implemented by a mechanism in software, hardware or in a combination of both. Examples of functional entities that can form a distributed information system are a file server, an operating system, an application program, etc. Examples of functional entities at lower abstraction levels are a computer, a terminal, a memory board, a printer, etc.

Figure 2.5 depicts a system as a composition of interacting parts (functional entities).

Interaction points delimit functional entities, and determine which functional entities are capable of interacting. For each functional entity one can identify its *environment,* which is a collection of other functional entities and possibly the system's environment. The system's environment can also be considered as a functional entity. The environment of a functional entity should be restricted to those other functional entities with which this functional entity directly interacts.

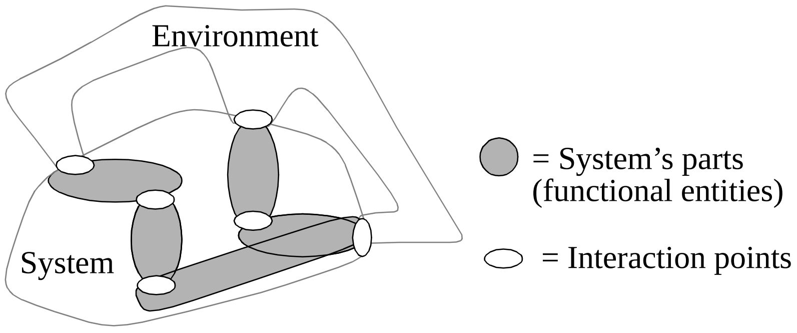


Figure 2.5: Distributed perspective of a system

2.2.2 Integrated Perspective of a System

The definition of the system as a composition of functional entities determines *how* the system is constructed. Since many alternative compositions of functional entities may be able to construct (implement) the same observable behaviour of the system, one should to be able to represent the system in a way that is independent of specific compositions of functional entities. Such a definition is positioned at a high level of abstraction with respect to the system's internal structure, and is called the *integrated perspective* ([9]).

The integrated perspective considers the system regardless of its composition in parts, considering it as a single functional entity, whose behaviour is observable only at the interaction points between the system and its environment. Interaction points which are internal to the system are not observable to its environment. This definition represents the system from the point of view of its environment, representing the *what* of the system.

Figure 2.6 depicts the integrated perspective of a system.

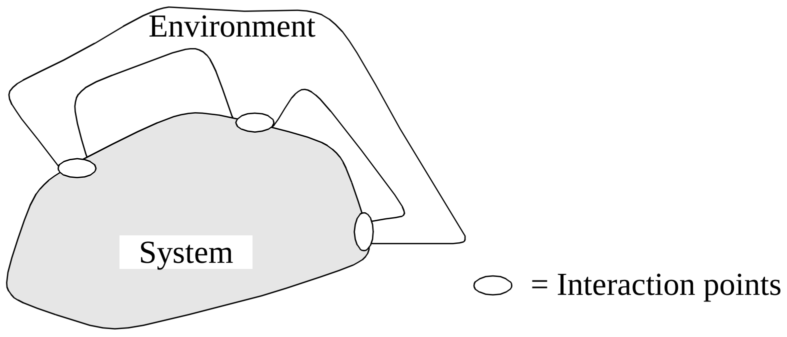


Figure 2.6: Integrated perspective of a system

The behaviour of the system as defined using the integrated perspective can be used as a functional requirement for the definition of the behaviour in the distributed perspective. A definition of a system using the distributed perspective should conform to the definition of the system using the integrated perspective, in the sense that the observable behaviour of the system as defined in the integrated form conforms to the observable behaviour of the distributed form, according to well-defined conformance requirements.

The distributed perspective has to conform to the integrated perspective, but there is still some freedom in the determination of the implementation relation to be used in the behaviour domain. For example we may impose that the distributed perspective has to be equivalent to the integrated perspective if it is necessary, but we can alternatively impose that the distributed perspective has to be a proper reduction of the integrated perspective. The specific implementation condition depends on the objectives of the design step.

The distributed perspective of the system can be obtained in an iterative way, from the integrated perspective. The integrated perspective of a system can be decomposed in parts, these parts can be decomposed again in (sub-)parts, and so on, until a convenient distributed perspective of the system is obtained. This allows one to define of a (top-down) design methodology based on step-wise refinement, such that in each refinement step a more refined structure in terms of compositions of functional entities is derived ([9]).

2.2.3 Integrated Perspective of System and Environment

The definition of a system according to the integrated perspective implies that the individual responsibilities and constraints of the system in performing interactions are defined separately from the behaviour of its environment. The system environment can be defined in the same way as the system. Together these definitions determine *how* the system and its environment contribute to the execution of their common interactions. Since many possible combinations of responsibilities and constraints are possible between the system and its environment, resulting in the same common behaviour, it is possible to represent a functional entity which displays this common behaviour independently of a specific distribution of responsibilities and constraints over system and environment.

This functional entity defines an integrated perspective of a system and its environment, which will be called the *interaction system between the system and its environment*. In the definition of the behaviour of the interaction system only the results of each interaction is defined, but not the different ways in which the system and the environment may contribute to the establishment of these results. The interaction system behaviour defines at a very high abstraction level *what* happens between the system and its environment, not *how* it happens.

Figure 2.7 depicts an interaction system.

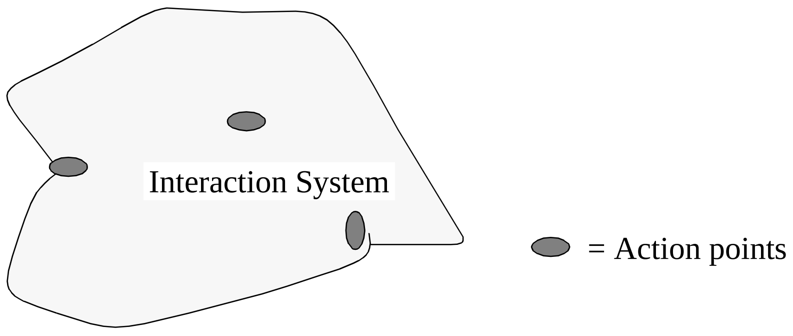


Figure 2.7: Interaction system between a system and its environment

The definition of the behaviour of a system and the behaviour of the system's environment can be derived from the interaction system between the system and its environment, by choosing and defining the individual responsibilities of the system and its environment in the execution of interactions. The combination of the behaviour of the system and its environment should conform to their interaction system, in the sense that the system and its environment together must be a proper implementation of their interaction system.

Integrated Interactions

The definition of an interaction system is based on the concept of *integrated interaction*. An integrated interaction represents an interaction in such a way that the distribution of responsibilities and constraints over its participants is ignored. When defining the behaviour of an interaction system, we consider the whole interaction system as a functional entity. Integrated interactions can then be considered as *actions* of the interaction system, where an action is a unit of activity of a functional entity.

The definition of the interaction system between system and environment is more natural than we may imagine. Taking for example the instructions manual of a television set, it defines what the television does under well-defined actions of the user. For example, if the user presses bottom Channel+ the television tunes the next channel. While from the point of view of the users this may describe the behaviour of the television set, from the point of view of the television manufacturer it describes the interaction system between the users and the television, since it also tells the users what they have to do in order to obtain the desired behaviour.

Behaviour Point of View

There is a relationship between the concept of interaction system between a system and its environment and the point of view from which behaviours are defined. While on one hand the behaviour of a system is defined in terms of the possible interactions between this system and its environment, on the other hand the behaviour of the interaction system between the system and its environment is defined without considering specific responsibilities in their participation in interactions. In the former a behaviour is defined from the point of view of the environment, which corresponds to active participation in interactions, while in the latter a behaviour is defined from a point of view placed at a meta-level with respect to the environment and the system, which corresponds to passive participation in interactions.

This alternative view based on passive participation is very useful in the implementation of parts of a system that fall into a single implementation authority, since in this case the implementation authority has control over the different parts and can determine the distribution of responsibilities. Indeed the meta-level from which behaviours of interaction systems are defined corresponds to the designer's view of the system. Furthermore the definition of interfaces between the system and its environment can also follow from an interaction system definition. Finally, this alternative view allows us to define a design framework in which actions and interactions co-exist, and form the building blocks for the definition of behaviours.

Traditionally the specification of a system (behaviour) is interpreted in either of the two following ways ([6]): (i) as to constrain the behaviour of the environment so that it behaves according to some pre-defined conditions, or (ii) as an expression of what happens in case nothing goes wrong between the system and environment, i.e. the expression of correct behaviour, abstracting from exception handling. These two rather antagonistic alternative interpretations of specifications can be unified if one intentionally describes the interaction system between the system and environment; neither we need to impose the constraint on the environment, nor on the system.

Suppose we want to define the behaviour of a system that *10* seconds after an input produces an output. With the concept of observability by interaction, this specification should be augmented with a constraint on the behaviour of the environment, such as ‘outputs are always enabled’, or ‘immediately after an input, output is enabled’ or that ‘the behaviour of the environment should be a mirrored image of the behaviour of the system’. Considering the interaction system between the system and its environment, it would suffice to state that *10* seconds after an input, an output is established ([8]).

2.3 Action Modelling

Behaviours of functional entities are defined in terms of actions and interactions and their relationships. Therefore it is important to investigate precisely what actions and interactions should represent and the objectives of defining them.

2.3.1 Objectives in the Design Process

We consider an activity as a possibly complex composition of actions, which has in its totality a certain final objective. In this definition of activity we deliberately abstract from the assignment of parts of this activity to functional entities.

Action modelling is a technique in which possibly complex activities of the real world are represented by more abstract actions. Suppose there is an activity in the real world from which all details are known. This activity can be represented by an abstract action that models the achievement of the final objective of the activity.

For example making coffee is an activity that involves actions such as buying coffee, taking a coffee machine, pouring coffee powder, putting water to boil, etc., and is only really finished when the final product, i.e. coffee, is obtained. The final objective of all these actions is to eventually obtain a couple cups of coffee. We can model this whole activity (making coffee) by a single abstract action, which represents the availability of coffee.

When considering only the achievement of the final objective of an activity we abstract from details of how this final objective has been achieved, which are considered as irrelevant in the definition of the abstract action. Therefore action modelling consists of selecting the most essential aspects of the final objectives of activities and assigning them to corresponding actions, allowing one to reason about these activities without the burden of their details. We can also say that an action is the most elementary form of activity at a certain abstraction level.

We may need to represent the making coffee activity in a more detailed way at lower abstraction levels, by making other less abstract actions than the action that represents the whole activity explicitly. However this more refined representation of the making coffee activity should still correspond to the single abstract action in which coffee is made available.

Figure 2.8 depicts examples of this approach on daily activities.

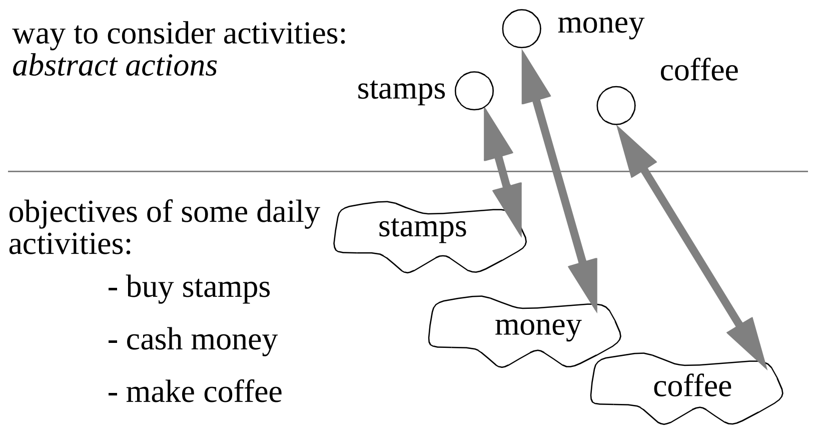


Figure 2.8: Daily activities as actions

Action modelling is a powerful technique for system analysis, since it allows one to evaluate the properties of systems composed of activities without having to modify these systems, for example to introduce diagnostic functions. In our design methodology, however, action modelling is used to support system development. This means that the activities do not yet exist in reality, but we want to implement them according to the actions that represent these activities. Reversing the reasoning, action modelling consists of representing the expectations on the objectives of the activities that shall compose the real system to be produced.

In system development an action should represent what happens, while the activities represent how things actually happen. Designers have a choice of activities to implement actions, determining what we call the design freedom. Design choices are made by designers, based on the knowledge of available or feasible activities, implications related to each specific choice with respect to price, complexity, etc., and some knowledge of which activities are correct implementations of which actions.

Since the knowledge on activities and implications of choices may vary from each instance of the design process and are normally difficult to capture in a generic framework, this work concentrates on the knowledge on which activities can be considered correct implementations of the more abstract actions.

Figure 2.9 depicts the application of action modelling in system development.

In addition to representing an activity by a single action, designers should also be able to represent complex compositions of activities in terms of compositions of abstract actions and relationships between these actions. A systematic way to address the conditions for action modelling in general is the following: (i) consider the condition in which an abstract action represents an activity and (ii) consider the condition in which a composition of abstract actions represents a composition of more complex activities.

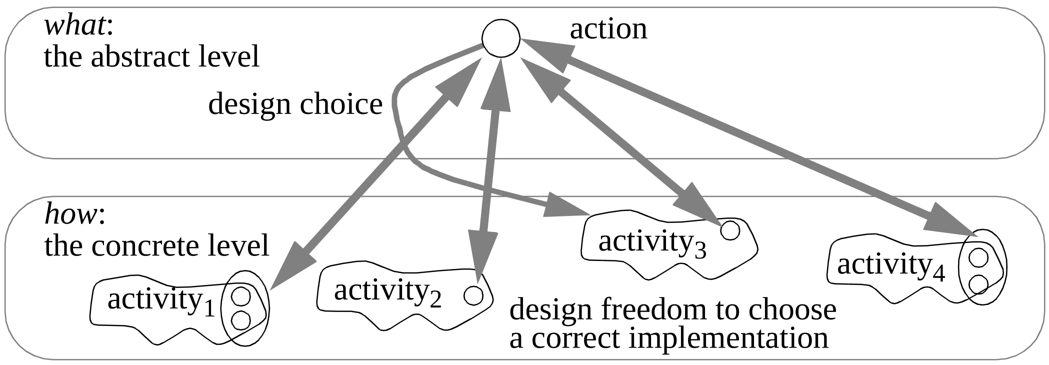


Figure 2.9: Action modelling in system development

Figure 2.10 depicts this strategy.

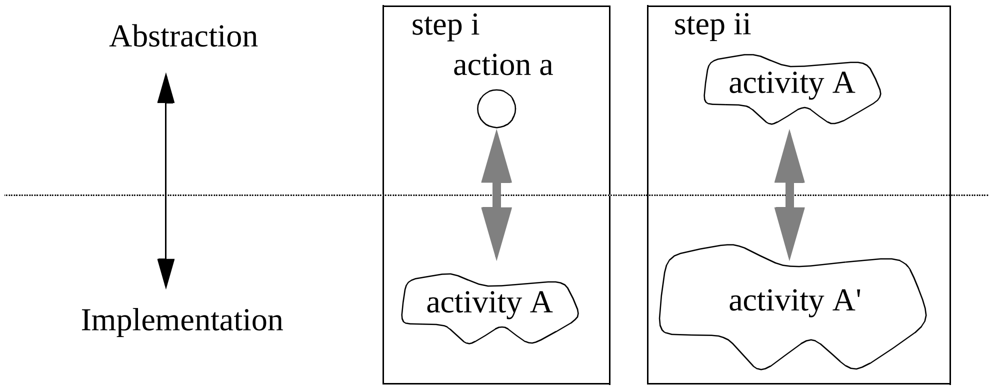


Figure 2.10: Conditions for implementation correctness

2.3.2 Activity Represented by an Action

We consider activities of the real world which aim at establishing values of information from now on. Activities of this kind should be obtained in the design process of distributed information systems. An action models an activity if it represents the essential aspects of the objective of this activity. These essential aspects for an activity in which values of information are established are:

- *all* the *values of information* that are made available by the activity;
- the *time* at which these values are made available;
- the *place* at which these values are made available.

The values of information established by an action are all the values of information made available at the end of the activity that this action models.

Different parts of the values of information that are made available in an activity may be established at different time moments, by different actions of this activity. Since we should be able to refer to all the values that are made available by an activity, the abstract action modelling the activity should represent the fact that all the actions of the activity in which these values are established have happened, which implies the completion of the activity. Consequently, the time of the occurrence of such an abstract action should be the time of occurrence of a certain *final action* of the activity, where all the values established in the activity are made available. The location of the abstract action should represent the locations of the actions of the activity.

The values of information established in the final action of an activity may be just part of the values made available by the whole activity. This means that it should be possible for the final action of an activity to make the rest of the values established in this activity available for reference by other activities. This can be done by considering that an action may have, apart from the values that are established in this action, some other retained values, which have been established before. Retained values are discussed in detail in Chapter 3.

Figure 2.11 illustrates the representation of an activity by an action.

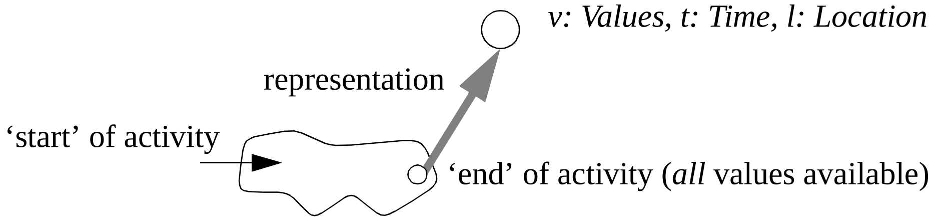

v: Values, t: Time, l: Location
representation
'start' of activity
'end' of activity (*all* values available)

Figure 2.11: Modelling of activity by an action

2.3.3 Activities Represented by Compositions of Actions

Complex designs involve complex compositions of activities, in which values of information are established. These activities may be related, such that some activities cause other activities or are excluded by other activities. In this case we can represent each activity by an abstract action, such that the complex composition of activities is represented by a more abstract composition of actions, under the following conditions:

- the values of information made available in the activity are the values of information established by the abstract action;
- the time when all the values of information of the activity are made available is the time of occurrence of the abstract action;
- the place where all the values of information of the activity are available is represented by the location of the abstract action;
- the role of an activity in its context, which defines how the activity is influenced by other activities and influence other activities, is represented by the role of the abstract action with respect to the actions that represent these other activities.

The role of an action in a behaviour is characterized in Chapter 3. The conditions for an action to be considered a proper abstraction of an activity are discussed in more detail in Chapter 7.

Figure 2.12 depicts the relationship between a composition of related activities and actions with arbitrary activities and actions.

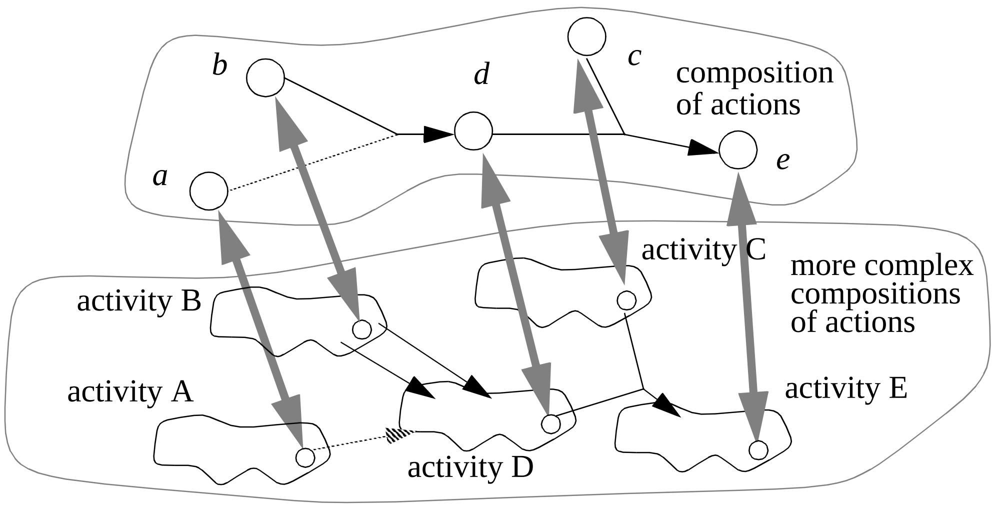


Figure 2.12: Activities and possible corresponding behaviour

The replacement of compositions of actions by more complex composition of actions can be applied recurrently in the design process. However, the design process must reach a final product, in which no more replacements are possible. At sufficiently low abstraction levels, actions should correspond to the actions available in the implementation environment. The use of a single framework to define compositions of actions at different abstraction levels makes it possible to determine whether activities are correct implementations of actions across these abstraction levels, covering the whole design process.

2.4 Application in Design Methodology

The different abstraction levels identified so far are only useful in the design process of a system, i.e. in its conception, design and implementation, if these levels can be ordered, determining global design objectives or milestones. The relative position of these abstraction levels of design can be used to define design steps and their systematic application.

2.4.1 Milestones of a Global Design Process

The design process should start with the most abstract representation of the system (the *what*), and includes details on the system construction (the *how*) in successive design steps. Therefore the design process starts with user-oriented descriptions, which should be simple and easy to understand. The most abstract description of a system is the definition of the *interaction system* between the system and its environment. This description should be formulated at the beginning of the design process.

The next step is to distribute responsibilities and constraints on the establishment of the interaction system behaviour to the system and to its environment. This leads to an *integrated system description*, in which actions of the integrated system are transformed into interactions between the system and its environment. Requirements for the behaviour of the environment are often implicitly generated in this design step.

The design process ends up in a possibly complex implementation-oriented description, where all information for the construction of the system is included ([3]). Therefore the integrated system description should be systematically decomposed in terms of a combination of functional entities. This determines a detailed description of the system structure, which is called a *distributed system description.*

Finally the behaviours of the functional entities that form the distributed system description should be refined, in order to be built out of available mechanisms. The resulting description is called an *interface refined distributed system description*. This description can be considered a blue-print for implementation. In order to obtain such a description some iterations of replacements of abstract actions by concrete activities may be necessary.

Figure 2.13 depicts the four milestones of a global design process, and their relative position. Intermediate abstraction levels between these milestones can be used in the definition of intermediate design steps, in which more specific design objectives and choices are considered.

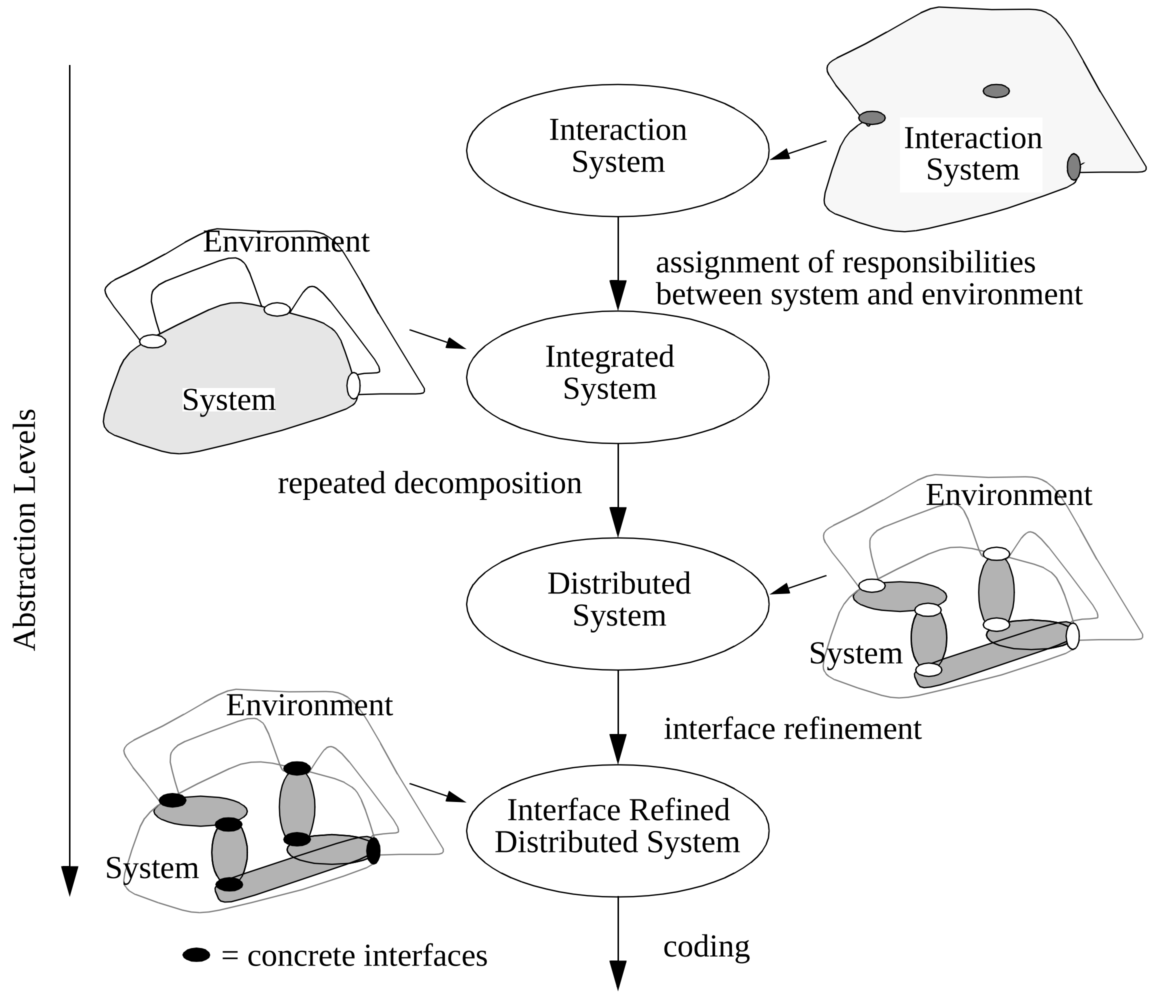


Figure 2.13: Milestones of a global design process

2.4.2 Roles of the Milestones

The behaviour of an interaction system defines the composite behaviour of the system and its environment, in terms of orderings of actions. The representation of an interaction system can be used to model the role of a system in its environment, as a requirement for the definition of the behaviour of the system itself. Therefore there is a relationship between this perspective and the objectives of the enterprise viewpoint, as defined in ODP. Although the interaction system perspective presented here is not defined in the same way as prescribed by the enterprise models, we believe that the interaction system perspective can provide the technical fulfilment that is still missing in most ODP enterprise models.

The behaviour of a system according to the integrated perspective is defined in terms of the orderings of interactions between the system and the environment, as observed (by participation) by the environment. This is traditionally called the observable behaviour of the system.

The behaviour of a system according to the distributed perspective is defined in terms of the orderings to interactions between system's parts, and between system's parts and the system's environment. This results in the definition of the observable behaviour of each system's part and the definition of the interconnection structure between parts.

The behaviour of the system according to an interface refined distributed perspective defines the most elementary interactions between system's parts or between system's parts and the system's environment, and their relationships. These most elementary interactions between system's parts or between system's parts and the system's environment define a set of concrete interfaces, which cater for the accomplishment of the abstract interactions defined in the distributed perspective.

2.4.3 Design Choices

The integrated perspective of the system determines a particular choice on the assignments of responsibilities for the system and for the environment. Similarly, the distributed perspective determines a particular choice of system's parts, and the interface refined distributed perspective determines a particular choice of more concrete interactions. This means that a design tree can be defined, which has an interaction system as the root node, and the other milestones as possible nodes.

Figure 2.14 depicts this design tree.

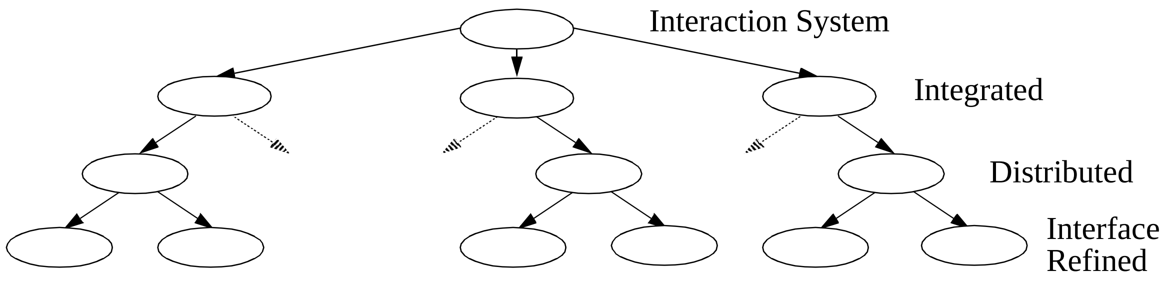


Figure 2.14: Design tree of milestones

However, it appears that for a manufacturer the definition of the system according to the integrated perspective should be made generic and constant, while different requirements and constraints on the users of the system (system’s environment) would determine a family of interaction systems. Since the integrated perspective is actually the first description of the system without considering its environment, it seems counter-productive to produce a multitude of such definitions, and therefore produce a multitude of systems, for different environment behaviours. This would modify the behaviour tree considerably.

Figure 2.15 depicts the influence of a generic integrated system definition in the design tree.

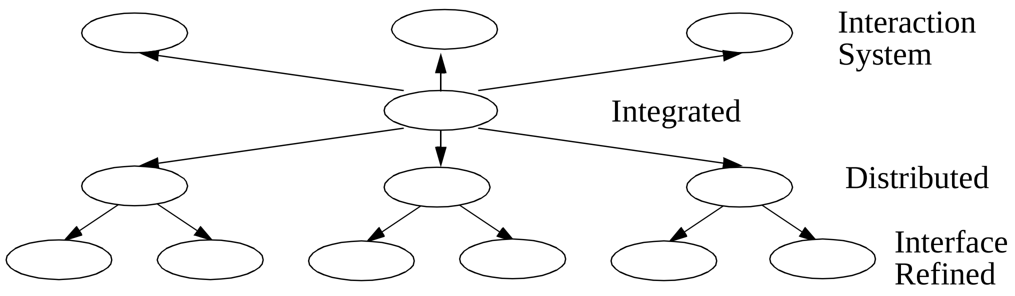


Figure 2.15: Generic integrated system

2.5 Examples

This section presents some examples, in which the technical applications of the concepts introduced so far, especially the design milestones, are illustrated.

2.5.1 OSI Service and Service Providers

The OSI RM defines a set of concepts which are necessary for the design of services and protocols. In the sequel we consider some possible correspondences between the abstraction levels identified in section 2.2 and some of these concepts.

Service and Service Provider

The Draft International Standard ISO 10731 ([5]) is the document where the concepts of service, service provider, service primitives, etc. are defined. The definitions of (OSI-)service and (OSI-) service provider are copied verbatim below:

> *OSI-service: the capability of an OSI-service-provider which is provided to OSI-service-users at the boundary between the OSI-service-provider and the OSI-service-users.*
>
> *OSI-service provider: an abstract representation of the totality of those entities which provide an OSI-service to the OSI-service-users.*

The questions that immediately arise are: is the service the same as the service provider? if not, what is then the difference between them? These question shall be answered below.

The service provider is defined as an abstract representation of a structure of entities which provide the service and therefore corresponds to the integrated system perspective presented in section 2.2.2. The service is defined as the capability of a provider, provided at the boundary with the service users. Since the interactions between service provider and service users only occur at their boundary, this capability can only be expressed by the definition of the interactions at this boundary and the relationships between these interactions. Furthermore [5] states that:

> *An OSI-service definition is the complete expression of the behaviour of an OSI-service-provider as seen by its service users'... 'To make proper use of an OSI-service, it is necessary for an OSI-service-user to reference the OSI-service definition. As a result, an OSI-service definition constrains the behaviour of the OSI-service-users. Nevertheless it is not the purpose of an OSI-service definition to express the complete behaviour of OSI-service-users.*

This means that the service definition is an expression of the behaviour of the service provider, although is also constraints the service users. Based on the roles of the design process milestones we can interpret a service as a representation of the *interaction system between the service provider and the service users*.

Other reasons for considering the service as an interaction system between the service provider and the service users are:

1. the service boundary crosses the implementation domains of the computer systems that implement the distributed OSI systems. Therefore designers and implementers of protocols should have the freedom to choose how to implement that boundary crossing. This means that this boundary crossing should be defined in the most abstract way, i.e. by means of an abstract local interface defined as an interaction system. Abstract local interfaces to the service could be later in the design process placed in the service users, in the service provider, or even in a third interface process, depending on specific design decisions;
2. a complete behaviour description of a service should express both functional and quality of service aspects (real time constraints, probability, etc.), which can only be done in a satisfactory way by considering it as an interaction system ([8]). Since the integrated perspective of a system defines the participation of the system in interactions (observable behaviour), it is impossible to represent real time and probability constraints that apply to the integrated interactions (actions) between service provider and service users using this perspective. These constraints could be naturally represented in the behaviour of the interaction system between service provider and service users.

Figure 2.16 depicts the service as an integrated interaction system, structured in terms of abstract local interfaces and remote constraints.

The astonishing consequence is that a service is not necessarily the same as the service provider. Furthermore, in most cases it is not even desirable to make the service and the service provider the same. The behaviour of a service, according to this view, is observed by an external observer, which is not a functional entity of the system, placed at the service boundary, i.e. observing all service access points simultaneously. This is also pictorially represented in Figure 2.16.

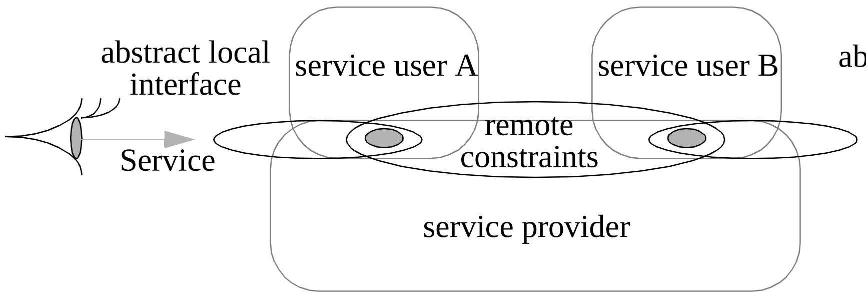


Figure 2.16: Service as an interaction system between service users and service provider

A service provider corresponds to the definition of the integrated system. At this point the designer may choose to place local constraints of the service in the service provider, in the service user, or in a third functional entity which performs the interfacing between service users and service provider. Remote constraints, i.e. those constraints that relate interactions at different interaction points, should be allocated to the service provider.

Figure 2.17 depicts the service provider in case the local constraints are assigned to the service users. The service users observe the behaviour of the service provider by interacting with it.

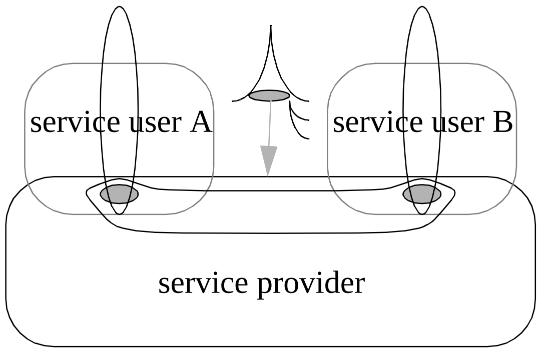


Figure 2.17: Service provider observed by the service users

The combined behaviour of the service users and the service provider must be an implementation of the behaviour of the original service.

A connectionless service, for example, may define that Data Request primitives at source service access points are possibly followed by Data Indication primitives at their corresponding destination service access points. Quality of service (QoS) parameters define aspects of the service such as transit delays, throughput, error rates, etc. The behaviour of the service could be defined in terms of its functionality (Data Request primitives possibly followed by Data Indication primitives) and its QoS (figures for maximum and average transit delays, throughput, error probabilities, etc.). The behaviour of the service provider may include some kind of backpressure on the acceptance of Data Request’s, but we may impose that service users must always accept Data Indication’s, to avoid jeopardizing for example the maximum transit delay of the service.

Protocol

A protocol corresponds to the distributed perspective of a system, in which inter-working aspects have been taken into consideration (amongst other aspects PDU syntax and semantics). A protocol is defined in terms of protocol entities and an underlying service provider.

Figure 2.18 depicts a protocol. The same reasoning we have used to view a service as an interaction system between service users and service provider can be repeated here for the underlying service. Figure 2.18 supposes that the local constraints of the underlying service are already assigned, either to the protocol entities or to the underlying service provider.

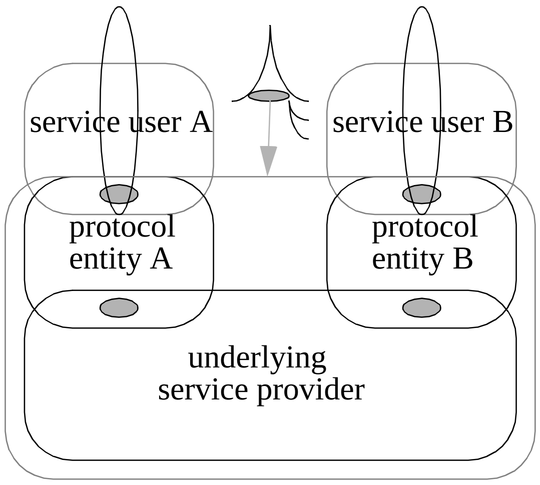


Figure 2.18: Protocol as a distributed representation of a service provider

The observable behaviour of the protocol must be an implementation of the observable behaviour of the service provider, from the point of view of the service users.

In some protocol specifications in LOTOS, as for example the OSI Transport and Session protocols, the local constraints of the underlying services are repeated in the definition of the behaviour of the protocol entities. These constraints can either be interpreted as something to be implemented (prescription of behaviour), or as a warning to indicate what the service provider allows or not. This repetition of constraints could be avoided if a clear assignment of constraints to the protocol entities and to the service provider had been made, or if LOTOS supported specifications in which specific assignments are ignored, i.e. the definition of the interaction system between the protocol entities and the underlying service provider.

2.5.2 Question Answer Service (revisited)

This section discusses the design of an instance of the *QA_Service* presented in [10]. The purpose of the system to be designed is the establishment of a question (*Q*) and an answer (*A*). This can be modelled using a single action, modelling the most abstract interaction system between the system and its environment. The specific values of *Q* and *A* are not constrained by this model, since the system must be kept rather general purpose. These values may depend on specific environments and situations in which this system operates.

Figure 2.19 depicts this view of the system.

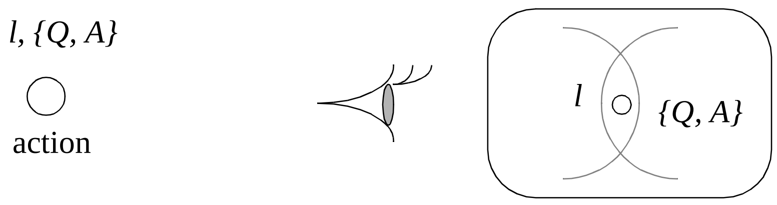


Figure 2.19: Integrated interaction system with single action

The system description can be modified to model the occurrence of two separate actions, namely the establishment of the question the establishment of the answer. This corresponds to a specific choice for the replacement of the abstract action of Figure 2.19 by two more concrete actions.

Figure 2.20 depicts this representation of the interaction system.

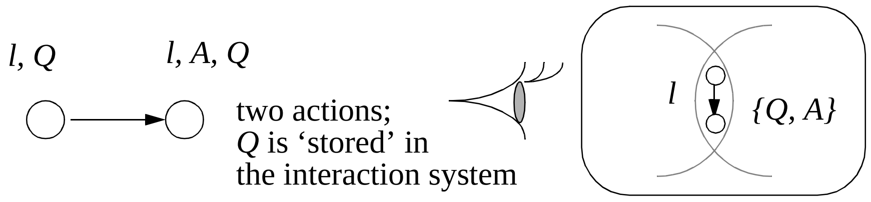


Figure 2.20: Integrated interaction system with two actions

The interaction system obtained in this way should be a correct refinement of the depicted in Figure 2.19. Intuitively we can assess that when the establishment of the answer has taken place, the values of information which have been established in the refined model are the same as the values of information established in the single abstract action, namely *{Q, A}*.

A system requirement that has not been considered so far is that the question and the answer are generated at distinct locations. Therefore we identify two action points l_1 and l_2, which should replace the action point *l*, and distribute the behaviour on these action points. In this way the question is originated in l_1 and the answer is originated in l_2, but both are made available at both action points. This interaction system is called the *QA Service*.

Figure 2.21 depicts the *QA Service*.

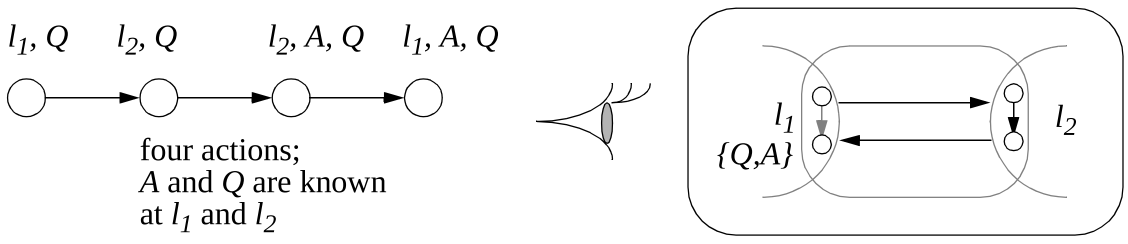


Figure 2.21: Question-answer service

The QA service of Figure 2.21 is a correct implementation of the interaction system of Figure 2.20, since the establishment to Q at l_2 corresponds to the establishment of Q at l, and the establishment of A at l_1 corresponds to the establishment of A at l.

We can now determine which functional entities are responsible for the different constraints of the interaction system. We consider that the environment must be decomposed in two service users, attached to the system through l_1 and l_2. The system obtained is the *QA service provider.*

We consider that the *QA service* provider takes care of the remote constraints of the *QA service*, i.e it transports a question of l_1 to l_2, and an answer of l_2 to l_1, independently. The service users are made responsible for the correct local ordering of interactions, namely that a question is followed by an answer. Actions are transformed into interactions, through synchronization and distribution of responsibilities.

Figure 2.22 depicts the decomposition of the original interaction system into system and environment.

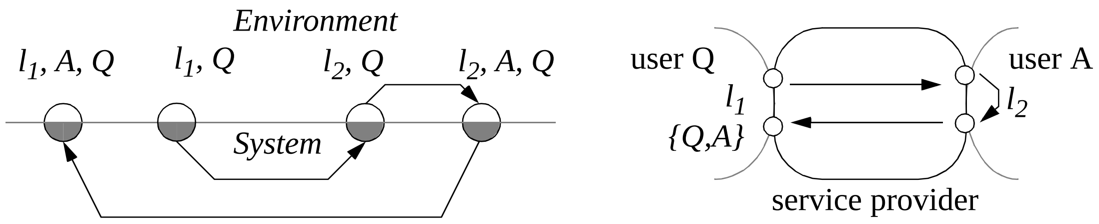


Figure 2.22: The QA service provider and its service users

A *QA protocol* that implements the service provider should now be defined. The function of this protocol is to map questions and answers data types onto generic data. The *QA protocol* is defined in terms of *QA protocol entities* (Q and A), and an underlying service provider, which transfers generic data.

Figure 2.23 depicts this distributed representation of the *QA service* provider.

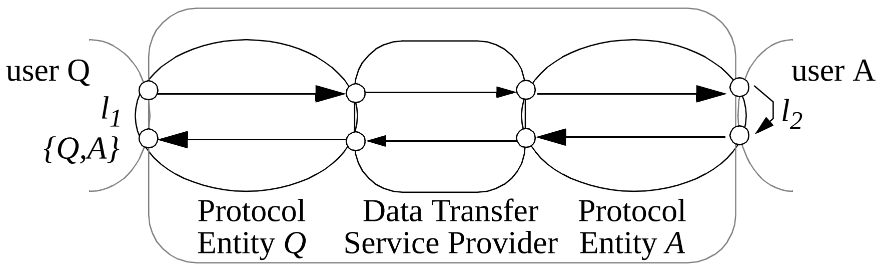


Figure 2.23: QA protocol (distributed perspective of the QA service provider)

2.5.3 Hospital Information System

This section considers the development of a Hospital Information System. In particular we show that some more specific design steps can be identified in between the design milestones of section 2.4, which can be used to define more concrete guidelines for designers.

The development of a Hospital Information System (HIS) should start with the definition of the interaction system between the HIS and its users. In this definition a set of sufficiently abstract actions are defined, and their relationships are established. Experience in the application of similar methodologies shows that the identification of abstract actions can be a difficult task, mainly for designers which are not used to action modelling. Actions should model the establishment of information values to be manipulated by the system; the procedures to obtain these values, for example, filling out fields of a user's interface, should not be modelled at this abstraction level.

Due to the variety of actions to be supported, and their different concerns, it is paramount that these actions are classified and organized according to the functions or user groups they support. For example, there may be functions that support the activity of nurses only, doctors only, management only, or combinations of these three user groups.

At this level we can also identify local and remote constraints, i.e constraints that can be locally supported and constraints that imply physical or logical distribution, respectively. These constraints should be further structured based on the classification of actions per function or user group support.

The next step could to determine the responsibility of the users and of the HIS in the establishment of the interactions defined in the interaction system. This results in a description of the HIS according to the integrated perspective (observable behaviour) and possibly a collection of constraints that should apply to the HIS users. These could also be used as the basis for the development of a user's manual guide.

The description of the HIS according to the integrated perspective is in the next design steps decomposed in abstract functional components, without taking into consideration any physical distribution of these functional components. These abstract functional components should represent the main functions of the HIS, and should be identified from a careful consideration of the actions, user groups and their relationships. For example we expect that some data base components containing information on patients will be identified. A collection of components that provide different views on these data bases for the different user groups might also be identified. There are many choices possible for the identification of these functional components. One of the most important criteria is that the composition of such functional components is a correct implementation of the observational behaviour as the original integrated HIS.

This design step has considered some more specific design objectives and choices than the ones presented in the definition of the distributed perspective milestone. The following step could be to distribute the functional components identified before, such that geographical distribution of users is taken into account. Some functional components will be decomposed and distributed in this way. This description generates a collection of application processes which are specific for supporting the different user groups.

The following step could be to identify a generic communication and possibly application infrastructure which could be used to support and implement the application processes, respectively. At this point some general and specific application functions can be identified. General application functions may be built out of available building blocks, while specific application functions may be further implemented.

In the last step the concrete interfaces to the communication infra-structure and general application functions are considered. A detailed definition of user interface functions and the interfaces of the specific application processes shall also be necessary.

Figure 2.24 depicts this design process, relating it to the design milestones identified before.

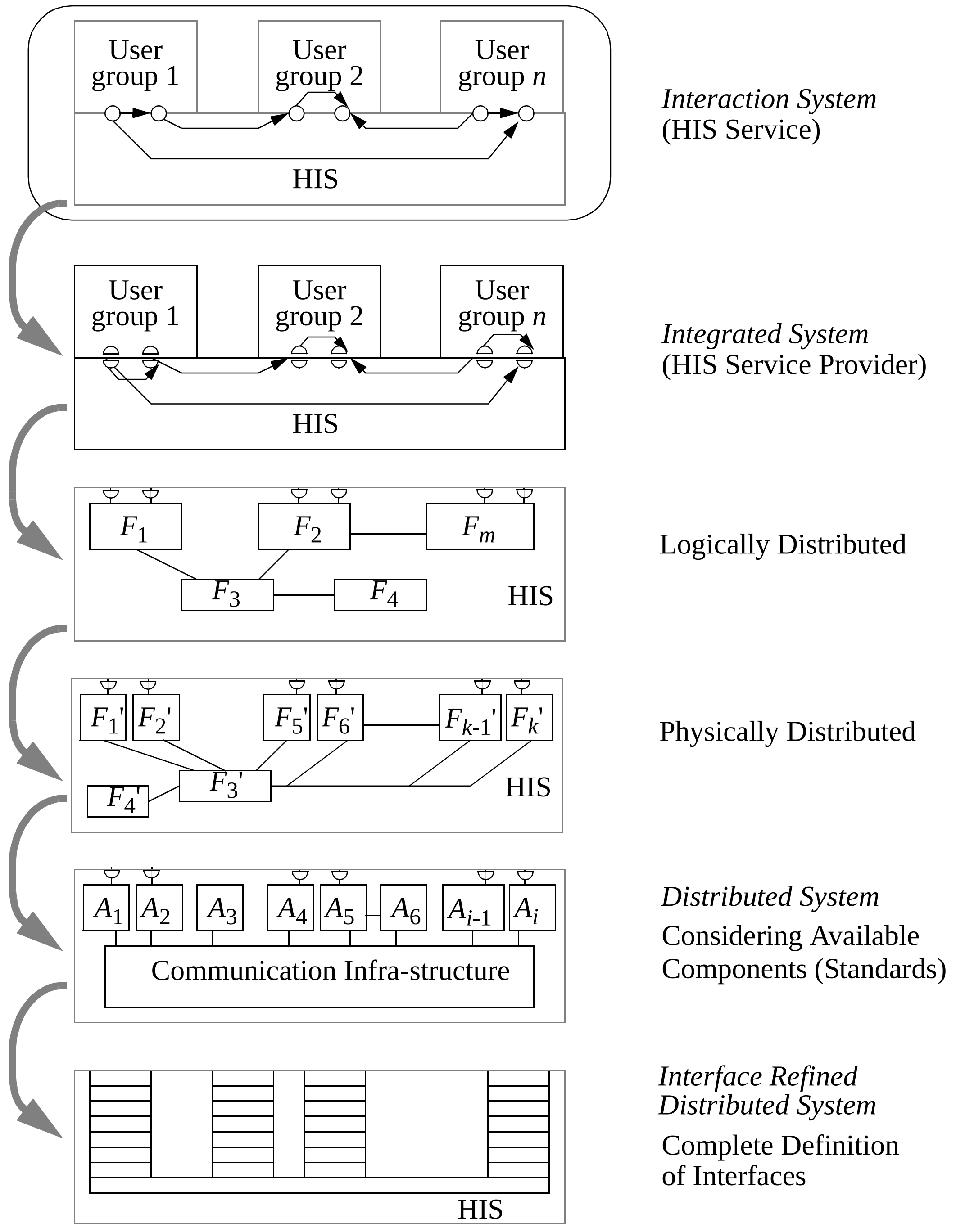


Figure 2.24: Possible design process for a Hospital Information System

The logically distributed and the physically distributed system representation are elaborated to accomplish design objectives that are more specific than the identification of system parts. Therefore we can consider that such design objectives have been introduced in between the generic design milestones, illustrating how more specific instances of the design process can be generated from our design methodology. The objectives of each system description and their relationships can be made more concrete, making them useful as concrete guidelines to designers.

2.6 References

[1] B. W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, pages 61–72, May 1988.

[2] A. M. Davis, E. H. Bersoff, and E. R. Comer. A strategy for comparing alternative software development life cycle models. *IEEE Transactions on Software Engineering*, 14(10):1453–1461, October 1988.

[3] L. Ferreira Pires and C. A. Vissers. Overview of the Lotosphere design methodology. In Commission of the European Communities, editor, *ESPRIT 1990, Conference Proceedings*, pages 371–387. Kluwer Academic Publishers, 1990.

[4] R. Gotzhein. *Open Distributed Systems. On concepts, methods, and design from a logical point of view*. Vieweg Advanced Studies in Computer Science. Vieweg, Wiesbaden, Germany, 1993.

[5] ISO/IEC JTC1/SC21. Revised text of CD 10731, Information Processing Systems - Open Systems Interconnection - Conventions for the Definition of OSI Services, May 1991.

[6] L. Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, January 1989.

[7] R. L. Probert and K. Saleh. Synthesis of communication protocols: Survey and assessment. *IEEE Transactions on Computers*, 40(4):468–475, April 1991.

[8] J. Schot. *The role of Architectural Semantics in the formal approach towards Distributed Systems Design*. PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[9] C. A. Vissers, L. Ferreira Pires, and D. A. Quartel. *The Design of Telematic Systems*. University of Twente, Enschede, the Netherlands, Nov. 1993. Lecture Notes.

[10] C. A. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma. Specification styles in distributed systems design and verification. *Theoretical Computer Science*, 89:179–206, 1991.

Chapter 3

Behaviour Definitions

This chapter introduces the design concepts necessary to define behaviours of functional entities, and shows the application of these concepts in the definition of behaviours. Causality plays an important role in behaviour definitions, since behaviours are defined in terms of causality relations involving actions and interactions. This chapter focuses on the definition of monolithic and finite behaviour, while more complex behaviour definitions are discussed in Chapters 4 and 5.

This chapter is organized as follows: section 3.1 introduces basic behaviour concepts, section 3.2 introduces the concept of causality relation, section 3.3 discusses causality relations involving conditions and constraints on action and interaction attributes, section 3.4 addresses the application of causality relations in the definition of finite behaviours and evaluates the expressive power of causality relations.

3.1 Basic Behaviour Concepts

The behaviour of a functional entity should be defined in terms of relationships involving actions and interactions of this functional entity. Actions and interactions are the basic concepts for behaviour definitions.

3.1.1 Actions and Interactions

We recall from [5] that an *interaction* is a *unit of common activity* of two or more functional entities, in which a value of information is established. An *integrated interaction,* is an interaction viewed in such a way that the individual responsibilities and constraints of the involved functional entities are ignored. An *action* represents an activity of a functional entity.

Since we have abstracted from the individual responsibilities of involved functional entities when defining an integrated interaction, an integrated interaction can be considered as an action of the interaction system between these functional entities. Although all integrated interactions are actions, some actions may not be integrated interactions, since one may define actions that are assigned to a single functional entity at a certain abstraction level, and are not distributed over multiple functional entities at lower abstraction levels.

Attributes of an action should reflect the essential aspects of the objectives of the activities modelled by this action, as discussed in Chapter 2. These essential aspects have determined the choice of the following action attributes:

- *location*: the logical or physical location where an action occurs. This attribute is often called *action point* or *interaction point*;
- *time*: the moment of time when all the values of information established in an action can be referred to by other actions;
- *action values*: the values of information that are established in an action;
- *retained values*: the values of information established in other actions that happened before, and kept by this action for further references;
- *probability*: the probability that an action occurs according to its definition (e.g. time and value conditions), once this action is enabled.

We say that an action is *enabled* if and only if all the conditions for the occurrence of this action are satisfied. An action only occurs at some moment if it is enabled at this moment.

Interactions have the same attributes as action, since they are simply different perspectives of the same concept. Interaction attributes, however, must be interpreted taking into consideration the distributed nature of an interaction. The term action is used to denote both actions and interactions below.

Actions are considered to be atomic, in the sense that they represent an *indivisible instance of activity at a certain abstraction level*. Direct reference to action attributes enable the definition of arbitrarily complex relationships between actions. Furthermore, some important design operations which are necessary in the design process imply manipulations of actions, their relationships and their attributes.

Example

Flipping a coin to decide the winner between two players is an example of an interaction that can be alternatively considered as an action. The act of choosing tail or head and the outcome of the throw is the mechanism that allows both players to interact. The result of the interaction has a different meaning for each player, i.e. for one of them it means defeat and to the other it means win. On the other hand, an unbiased observer sees a common action and a result indicating the winner and the loser. This kind of common action is the one meant here.

In this example, the time and place where the common action of determining the winner is established are the same, if we consider the observer view (common action) or the players view (interaction), once it happens. In general we observe that in the definition of a common action one defines “what happens”, while in the definition of an interaction one defines the constraints imposed by each participant, which is in some sense “how it happens”.

Time Modelling

When an action occurs, other actions are able to refer to all values of information of this action. An activity modelled by an action may have a certain duration. We abstract from duration of an activity by only considering the moment when the activity is completed as the time attribute value of the abstract action that models this activity.

Realistic timed models have to adhere to our intuitive notion of time. For example time always progresses infinitely, which implies that time can be modelled by an infinite totally ordered set. Time is also continuous, since between two moments of time one can always consider the existence of another moment of time. However, in case of simulation and implementation, time representation is discrete and bound to a certain precision. In these cases one should select the precision that is acceptable for the specific application area.

Time is modelled as continuous in this work. Time moments are represented by (positive) real numbers. Similarly to [4], we consider time to be relative, since we can always find a suitable time reference from which time moments can be related to.

Values of Information

Values of information established in an action can be of unlimited complexity. They can be a single value, a set of values, a list of values, or any other arbitrary data structure. In order to be manipulated effectively, only those characteristics of the values of information of an activity which are relevant at the abstraction level being considered must be represented. Furthermore values of information which are defined as complex data structures should be defined as a hierarchy of more elementary data structures. An example of technique that allows abstract representation and structuring of values of information is the *Abstract Data Type* (ADT) theory.

The retained values of an action consist of action values of other actions that have taken place before this action. Retained values are defined in order to allow actions to refer to these values, without having to be directly related to the actions in which these values are established. The occurrence of an action models, amongst others, that its retained values are available. In implementations, the functional entities participating in the actions should be able to access these values, for example from memory positions or variables in some implementation code. The action occurrence simply indicates to these functional entities that these values are available.

Roughly comparing, retained values are implicitly defined in programming or specification languages through the definition of scope rules, which define which language elements may refer to specific variables and values established earlier. The actual references are, in the case of programming and specification languages, a subset of the ones allowed by the scope rules, augmented with possible explicit references to variables and values outside a scope. In the definition of a design model we can afford to define retained values explicitly.

The *functionality* of an action is the complete set of values which are made available by this action. The functionality of an action consists of its action values and the retained values. Functionality plays an important role in the characterization of implementation relations between actions and activities in Chapter 7.

Figure 3.1 illustrates the relationship between action values, retained values and functionality.

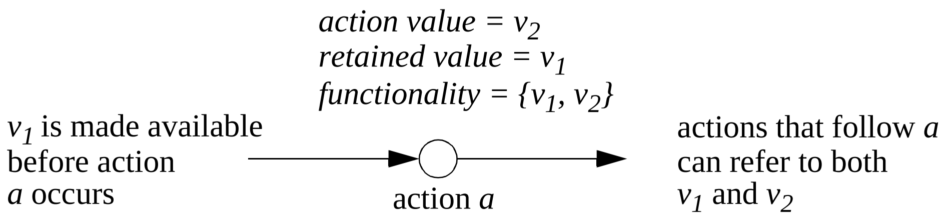

action value = v_2
retained value = v_1
functionality = $\{v_1, v_2\}$
v_1 is made available before action *a* occurs
action *a*
actions that follow *a* can refer to both v_1 and v_2

Figure 3.1: Values of information of an action

In Figure 3.1, a certain action *a* establishes value v_2 and retains value v_1, which had been established before the occurrence of *a*. The functionality of action *a* consists of both values v_1 and v_2. Actions that follow action *a* can refer to both values v_1 and v_2.

3.1.2 Behaviour Elements

The behaviour of a functional entity should define the possible ways this entity can function. The functioning of a functional entity can only be defined in terms of its actions. Therefore we define the behaviour of an interaction system in terms of a set of relationships and dependencies involving its actions that altogether determine the possible ways this functional entity can function.

We assume that actions in a behaviour can always be distinguished. Distinct actions are supposed to be unique, even if two or more distinct actions have all identical attributes. This implies that each distinct action of a behaviour can be identified by a unique *action identifier.* Action identification in a behaviour is arbitrary and must be guided by the technical needs to distinguish them. An action identifier represents that “something happens”, while action attributes precisely define “what happens”.

Suppose we have a behaviour consisting of two alternative occurrences: one in which the natural number value *0* is established, or another in which the natural number value *1* is established. These two occurrences can be modelled as a single action, where values *0* or *1* can be established, or as two distinct and conflicting actions, one which establishes *0* and another which establishes *1*. Actions should be distinguished according to some technical needs. In this example it may be preferable to model these two occurrences as a single action, for instance because they are expected to be implemented by a single mechanism.

Behaviour definitions should be able to represent the following elements: (i) initial actions, (ii) causality contexts of actions, and (iii) exit and termination conditions. Each of these elements is briefly discussed below:

- *initial actions*: each behaviour has actions which are allowed to occur independently of the occurrence of other actions of that behaviour. Such actions are called *initial action*s. Initial actions may refer to an initial set of attribute values (time reference, initial retained values, etc.), which have been established before the behaviour is activated. Initial actions may occur either spontaneously as the behaviour is instantiated (*start* actions), or may be enabled by other behaviours, characterizing the *entries* of this behaviour;
- *causality contexts*: the causality context of an action defines the role of this action in a behav-

iour. This means that a behaviour can be defined in terms of the causality contexts of all its actions. Possible time orderings of actions are implicitly defined in their causality contexts. Given the causality context of all actions in a behaviour, one should be able to determine possible time orderings of these actions;

- *exit and termination conditions*: a behaviour is said to *exit* if conditions of its behaviour enable initial actions of another behaviour. This enabled behaviour is then allowed to start, possibly receiving timing references, and values of information. When an exit of a behaviour occurs, this behaviour may still proceed, i.e. in case actions of this behaviour are enabled, they are still allowed to happen. An action is said to terminate in a behaviour if no more actions or other behaviours are enabled by it.

Behaviour entries and exits are discussed in detail in Chapter 4.

The causality context of an action of a behaviour *B* defines both (i) the necessary and sufficient conditions for it to occur, and (ii) the conditions of other actions of *B* in which it plays a role. This completely defines the role this action plays in behaviour *B*. The combination of the causality context of all actions of a behaviour completely defines this behaviour.

Figure 3.2 illustrates the relationship between causality contexts and behaviour.

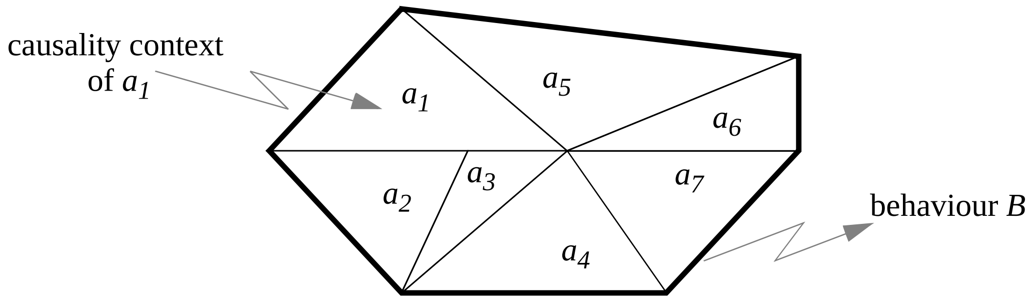


Figure 3.2: Causality contexts and behaviour

In Figure 3.2, behaviour *B* has distinct actions a_1, a_2, a_3, a_4, a_5, a_6 and a_7. The causality contexts of all these actions completely define *B*. A more precise characterization of the causality context of an action is presented in section 3.4.

3.2 Causality Relations

The conditions for the occurrence of an action of a behaviour *B* are defined in a *causality relation* between the other actions of *B* and this action. We say that a causality relation of an action defines the conditions for this action to occur, and the constraints on the attributes of this action. A causality relation is defined in terms of relationships involving actions and possibly their attributes. Causality relations can be used as elementary building blocks for the definition of arbitrarily complex behaviours.

In a causality relation, conditions for the occurrence of an action are defined in terms of the occurrence or non-occurrence of other actions. In a more general case, these conditions may also involve attribute values of these actions. Constraints on the allowed attribute values of an action in a causality relation may also refer to attribute values of other actions.

The identification of actions without referring to their actual attributes is used below as a short-hand notation to introduce the possible relationships involving actions without the burden of attribute details. However, relationships involving actions of practical use can only be represented by also considering the action attributes in these relationships. The role of action attributes in causality relations is discussed in section 3.3.

3.2.1 Interpretation Options

There are in principle two interpretation options for causality relations defined in terms of occurrence and non-occurrence of actions as conditions for another action to take place:

1. admit that all conditions have to be satisfied at the moment when an action takes place, and that after this action takes place these conditions do not have necessarily to stay valid;
2. admit that the conditions are or will ever be valid (either at the moment when the action takes place or sometime in future) for an action to take place, and that once these conditions are valid, they remain valid forever.

We consider an example to illustrate these two interpretation options: suppose two actions a_1 and a_2 are in the conditions of a_3, such that for a_3 to occur, a_1 must occur and a_2 should not occur.

According to option (1), only in case the conditions for occurrence of a_3 are satisfied, i.e. a_1 has happened and a_2 not, a_3 is allowed to happen. After a_3 happens, a_2 may even happen, depending solely on the conditions of a_2. Figure 3.3 depicts two possible action orderings option (1).

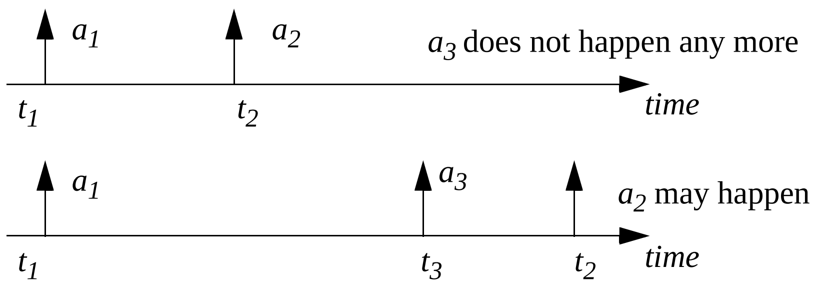


Figure 3.3: Possible orderings for interpretation option (1)

According to option (2), the conditions for occurrence of a_3 may not yet be satisfied, but if we know that they will be, a_3 is allowed to happen. Suppose then that a_1 has happened and a_2 has not. This means that, after a_3 happens, a_2 can not happen any more, since the condition must remain valid forever. According to this option, the conditions of a_3 at some moment also define the conditions for the occurrence of a_2. Figure 3.4 depicts two possible action orderings for option (2).

An advantage of option (1) is that all necessary and sufficient conditions for the occurrence of an action are stated in a single statement. This is not the case in option (2), because in this option causality relations in which the occurrence or non-occurrence of an action is a condition have to

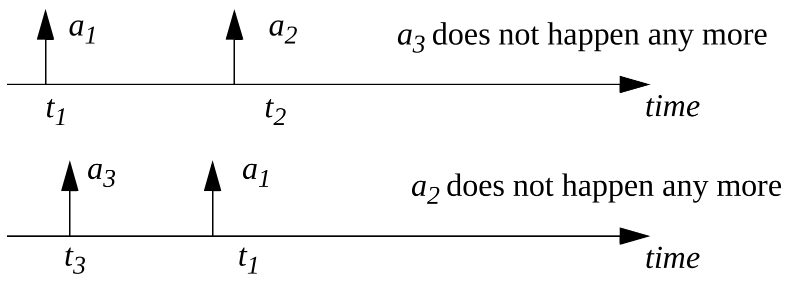

a_1
a_2
a_3 does not happen any more
t_1
t_2
time
a_3
a_1
a_2 does not happen any more
t_3
t_1
time

Figure 3.4: Possible orderings for interpretation option 2

be considered in combination with the causality relation that defines this action, in order to determine when the action is allowed to occur. Since we strive for having all the necessary and sufficient conditions for an being action defined in its causality relation, we choose for the first interpretation option.

3.2.2 Causality

Causality is often defined as the relation between a cause and its effect, such that the effect can not be caused by something that happens in the future. We bind causality to the past because we are only interested in building models that can be implemented. Causality to future events are impossible to build in practice.

We define the causality relation a_1 -> a_2 as:

> *the occurrence of the specified action a_1 is a condition for the occurrence of the specified action a_2. Only in case a_1 has occurred a_2 is allowed to occur.*

Actions are expected to take some time to occur in their implementations, although we do not represent this time explicitly. This means that if an action, say a_2, is only allowed to happen if another action a_1 has happened, characterizing a causal relationship, we expect that the implementation of a_2 needs some time to recognize the occurrence of a_1.

Furthermore, the purpose of defining causality relation is actually to allow actions to refer to attributes of its condition actions, which may also cost some time in implementations. Therefore if a_1 is a condition for a_2, it is reasonable to model it such that a_1 must have happened *before* a_2 happens. We can also say that whenever a_1 and a_2 happen in a run of the system, $t_1 < t_2$ where t_1 is the time of occurrence of a_1 and t_2 is the time of occurrence of a_2. This condition is always implicitly present in causality relations involving causality.

Since causality relations are supposed to be defined for each distinct action of the behaviour of B, a necessary condition for any action to occur is that it has not yet occurred. This condition is also implicitly present in causality relations with causality.

Causality relation a_1 -> a_2 says nothing about the possible occurrence of a_1. The possible occurrence of a_1 should be defined in the causality relation of a_1, which is not part of the causality relation of a_2, according to the interpretation option chosen for causality relations.

Causality relations determine possible behaviour orderings between actions. For example, the set of causality relations $\{a_1 \text{ -> } a_2, a_2 \text{ -> } a_3, a_3 \text{ -> } a_1\}$ describes an impossible behaviour, since a_2 can only happen if a_1 has happened, a_3 can only happen if a_2 has happened, and a_1 can only happen if a_3 has happened, which means that none of the conditions will ever be fulfilled. This kind of definition may be either tolerated and solved at semantic level (dynamic semantics), or disallowed from the beginning (static semantics). However, it is the explicit responsibility of the designer to recognize and eliminate impossible behaviour, such that only meaningful behaviour is specified.

We can say that if a_1 -> a_2:

- the occurrence of a_1 allows (enables) the occurrence of a_2, or
- the occurrence of a_2 depends on the occurrence of a_1, or
- the occurrence of a_1 is a necessary condition for the occurrence of a_2, or
- a_1 enables a_2.

We consider the causality relation a_1 -> a_2 as a definition of an elementary form of behaviour and as an element of the total behaviour of *B* that we want to define. This elementary behaviour defines a mechanism that is embedded in the functional entities that are engaged in the possible execution of a_1 and a_2. It links the possible execution of a_1 to the possible execution of a_2, such that if a_1 has occurred, a_2 is allowed to occur.

3.2.3 Exclusion

Similarly to having the occurrence of an action as a condition for another action, we consider that the non-occurrence of an action can also be a condition for another action. The purpose of defining this is to be able to define exclusion between actions, which is the situation where the occurrence of an action implies that another action is not allowed to happen.

We define exclusion as the relation $\neg\, a_1$ -> a_2, such that:

> *the non-occurrence of the specified action a_1 is a condition for the occurrence of a_2. As long as a_1 does not occur, a_2 is allowed to occur; if a_1 occurs and a_2 had not occurred before it, a_1 excludes the occurrence of a_2.*

Since we want to use exclusion as a condition which is true at the moment of the occurrence of an action, it is not enough to define that a_1 does not happen before a_2, but a_1 should not happen at the same time a_2 happens either.

Conflict between two actions, which is sometimes called *action choice* in the literature, means that at most one of these two actions happen in an instance of execution of a behaviour. The choice of defining exclusion as the non-occurrence of an action before or at the same time as the excluded action makes it possible to model conflict between actions in terms of a symmetric exclusion, which would have been impossible with another choice of implicit time conditions.

Exclusion is also often called *disabling* or *asymmetric conflict* ([3]) in the literature.

We say that if $\neg\, a_1$ -> a_2:

- the occurrence of a_1 prevents the occurrence of a_2, or
- the occurrence of a_2 depends on the non-occurrence of a_1, or
- the non-occurrence of a_1 is a necessary condition for the occurrence of a_2, or
- a_1 excludes a_2.

Whenever both a_1 and a_2 happen in an instance of execution of a behaviour containing this causality relation, $t_1 > t_2$ where t_1 is the time of occurrence of a_1 and t_2 is the time of occurrence of a_2. This condition is always implicitly present in causality relations with exclusion.

We consider the expression $\neg\, a_1$ -> a_2 as a definition of an elementary form of behaviour and an element of the total behaviour of *B* that we want to define. This elementary behaviour defines a mechanism that is embedded in the functional entities that are engaged in the possible execution of a_1 and a_2. It links the possible execution of a_1 to the possible execution of a_2, making it sure that if a_1 occurs before a_2, a_2 is not allowed to happen any more.

Although the definition of exclusion naturally follows from the requirements of our model, it introduces an interesting mirrored symmetry with respect to causality. This mirrored symmetry occurs in case both a_1 and a_2 happen in a run of the system: in case a_1 enables a_2, a_1 should happen before a_2, and in case a_1 excludes a_2, a_1 should happen after a_2.

3.2.4 Generic Causality Relations

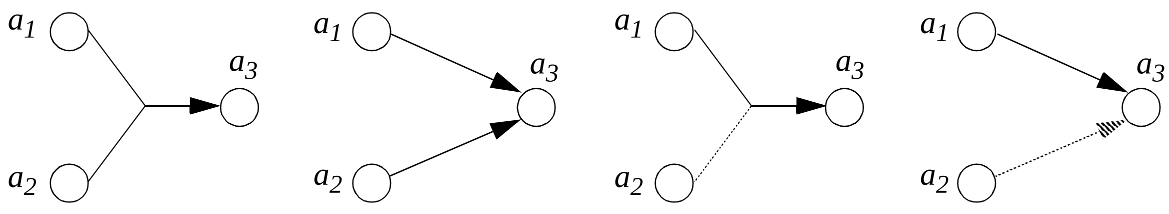
Elementary relations based on causality and exclusion are used to compose more complex causality relations. We combine these elementary conditions using *and* ($\wedge$) and *or* ($\vee$) logical operators, and we interpret these combinations according to the rules of boolean logic. Some examples of causality relations are:

- $a_1 \wedge a_2$ -> a_3: the occurrences of both a_1 and a_2 are conditions for the occurrence of a_3. In this case both a_1 and a_2 must have happened before a_3 is allowed to happen;
- $a_1 \vee a_2$ -> a_3: the occurrence of a_1 or a_2 is a condition for the occurrence of a_3. In this case a_1 and a_2 may both happen, but the occurrence of one of them is already sufficient for the occurrence of a_3. In case both a_1 and a_2 occur before a_3, there is a *non-deterministic choice* on which of these actions have caused a_3. In case a_3 refers to attributes of a_1 or a_2, this choice determines which attributes are used, such that a_3 may refer to the attributes of a_1 or to the attributes of a_2, but not to the attributes of both;
- $a_1 \wedge \neg\, a_2$ -> a_3: the occurrence of a_1 and the non-occurrence of a_2 are both conditions for the occurrence of a_3;
- $a_1 \vee \neg\, a_2$ -> a_3: the occurrence of a_1 or the non-occurrence of a_2 or both are conditions for the occurrence of a_3.

The examples above only define the conditions for a_3, stating nothing about the conditions for the occurrence of a_1 and a_2. These conditions should be stated in the causality relations of a_1 and a_2 respectively.

Causality relations can also be represented using a graphical notation. This graphical notation is expected to be useful for helping understanding and analysing causality relations, especially later on when more complex forms of behaviour structuring are investigated. In this graphical notation actions are depicted as circles, and causality relations are depicted as arrows.

Figure 3.5 depicts the examples presented above using our graphical representation. This notation is consistently applied throughout this work.



(a) $a_1 \wedge a_2$ -> a_3 (b) $a_1 \vee a_2$ -> a_3 (c) $a_1 \wedge \neg a_2$ -> a_3 (d) $a_1 \vee \neg a_2$ -> a_3

Figure 3.5: Graphical representation of causality relations

Arbitrarily complex causality relations can be composed by combining action occurrence, non-occurrence and the logical operators $\wedge$ and $\vee$, such as in the following example:

- *$(a_1 \wedge a_2 \wedge \neg a_3) \vee (\neg a_1 \wedge a_3)$ -> a_4*: the occurrence of a_1, the occurrence of a_2 and the non-occurrence of a_3, or the non-occurrence of a_1 and the occurrence of a_3, or both of these conditions, are conditions for the occurrence of a_4.

We call the left hand side of a causality relation the *(action) conditions*. The symbol -> is the *causality operator.* The right hand side of a causality relation is called the *result* or *resulting action*.

Consistently with the option adopted for the interpretation of the basic causality relations, all the prescribed conditions have to be fulfilled at the moment when an action occurs. Generalizing, a causality relation in a behaviour *B* has the form *F(A)* -> a_j, $A \subseteq A_B - \{a_j\}$, where A_B is the set of actions of behaviour *B*, and *F* is a formula using $\vee$, $\wedge$ and elementary conditions of the form a_k and $\neg\, a_k$, representing the occurrence and non-occurrence of a_k, respectively, where $a_k \in A$. *F(A)* has to be true at the moment a_j occurs.

3.2.5 Elements for an Algebra of Causality

Since occurrence and non-occurrence of actions are combined using logical operators in the definition of conditions in causality relations, one could expect that traditional boolean laws were applicable to reduce and prove properties of causality relations. However, due to the mirrored symmetry of causality and exclusion with respect to implicit time conditions, traditional boolean laws cannot be directly applied in this case.

Some elements for an algebra of causality are briefly discussed below:

- *enabled action*:
  true -> a_x

This means that a_x is enabled and can happen at any time, without being subject to any condition.

- *uniqueness of actions*:
  $a_x \wedge a_x$ -> a_y is equivalent to a_x -> a_y
  $a_x \vee a_x$ -> a_y is equivalent to a_x -> a_y
  $\neg\, a_x$ -> a_x is an ill-formed causality relation, since a_x should not happen at the same time as itself (a_x can never happen)
  a_x -> a_x is an ill-formed causality relation, since it is against the uniqueness of actions (a_x can never happen)
- *negation properties*:
  $a_x \wedge \neg\, a_x$ -> a_y defines a condition that can never satisfied (a_x can never happen)
  $a_x \vee \neg\, a_x$ -> a_y is *not* equivalent to *true* -> a_y

For example the behaviour {$a_x \vee \neg\, a_x$ -> a_y, $a_y \vee \neg\, a_y$ -> a_x} actually means that a_x and a_y are *interleaved*, since either a_x happens before a_y or a_x does not happen (meaning actually that if it happens, it happens after a_y), and vice-versa. This means that either a_x happens and then a_y, or a_y happens and then a_x, but they will never happen 'at the same time'.

Until now we have defined conditions in a causality relation without actually concerning whether these conditions are sufficient for an action to occur, but rather taking them as necessary conditions. In this way one could also assign a boolean *true* or *false* to a causality relation itself, meaning that the causality relation expresses a necessary condition or not, respectively. In this way the following could hold:

- *transitivity of necessary conditions*: a_x -> a_y and a_y -> a_z implies a_x -> a_z. This law states that considering a_y as a necessary condition for a_z implicitly makes the necessary conditions of a_y (in this case a_x) also necessary conditions for a_z.

References to action attributes impose a more rigorous constraint on the valid forms for replacement by applying transitivity. For example if a_z refers to some values of a_x, these values should be forwarded to a_z by using retained values attribute of a_y.

Some of the elements discussed above are useful in manipulations of behaviour definitions, for example in the definition of methods for assessing the correctness of design operations in which behaviours and actions are refined, in Chapters 6 and 7.

3.2.6 Choice of Causality

The choice of using causality for the definition of behaviours has been inspired by the objectives of our design model. The semantics of a design model should allow one to distinguish those designs that are considered to be different. In the specific case of behaviour definitions, we would like to distinguish the concept of independence from the concept of interleaving. This is necessary, since in most technical systems of interest, independence and interleaving are indeed distinct relationships between actions that have to be explicitly represented in the behaviour of these systems. Furthermore we want to use behaviour specifications as prescriptions for the construction of the functional entities that execute these behaviours, such that design information concerning independence or interleaving between actions is maintained in the course of the design process.

The distinction between independency and interleaving is not possible in case we consider a design model for behaviours based on arbitrary interleaving, such as the design model underlying design languages based on process algebras.

We illustrate the need for distinguishing independence from interleaving by an example of a bi-directional data buffer, with capacity one in each direction of communication ([1]). In order to simplify the example, we consider only one instance of communication per direction. We also consider that actions *in* and *out*, which represent insertion and retrieval of data, respectively, may occur at action points *a* and *b*.

Figure 3.6 depicts the structure of the bi-directional buffer example.

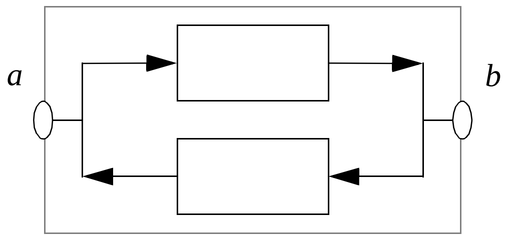


Figure 3.6: Bi-directional buffer example

Suppose the buffer behaviour is informally defined by the following requirements: (1) in case data is introduced at *a* through an action *in*, it can be retrieved at *b* through an action *out*; (2) in case data is introduced at *b* through an action *in*, it can be retrieved at *a* through an action *out;* (3) all interactions at *a* are interleaved and (4) all interactions at *b* are interleaved.

This behaviour could be described in LOTOS in the following way:

```
process BiBuff [a, b] :noexit :=
Buff [a, b] ||| Buff [b, a]
where
 process Buff [x, y] :noexit :=
 x !in ?v: Data; y !out !v ; stop
 endproc (* Buff *)
endproc (* BiBuff *)
```

The formal semantics (formal interpretation) of a LOTOS specification can be represented in terms of a behaviour tree. Figure 3.7 depicts the behaviour tree of this example.

By inspecting the behaviour tree in Figure 3.7 we notice that the formal interpretation of the specification has introduced an extra property to the behaviour, namely that some actions that were supposed to be independent have become interleaved. The consequence is that a designer cannot infer from the specification if the actions must be interleaved, which is the case for actions happening at *a* or *b*, i.e. the pairs <`a !in ?v` , `a !out ?v`> and <`b !in ?v`, `b !out ?v`>, or if the actions must be independent of each other, which is the case for the pairs < `a !in ?v`, `b !in ?v`> and < `a !out ?v`, `b !out ?v`>.

In case the final implementation of a specification is mapped onto sequential processes that do not support parallelism anyway, interleaving semantics does not present a drawback. However in the design of complex distributed systems, which is our area of concern, the introduction of extra

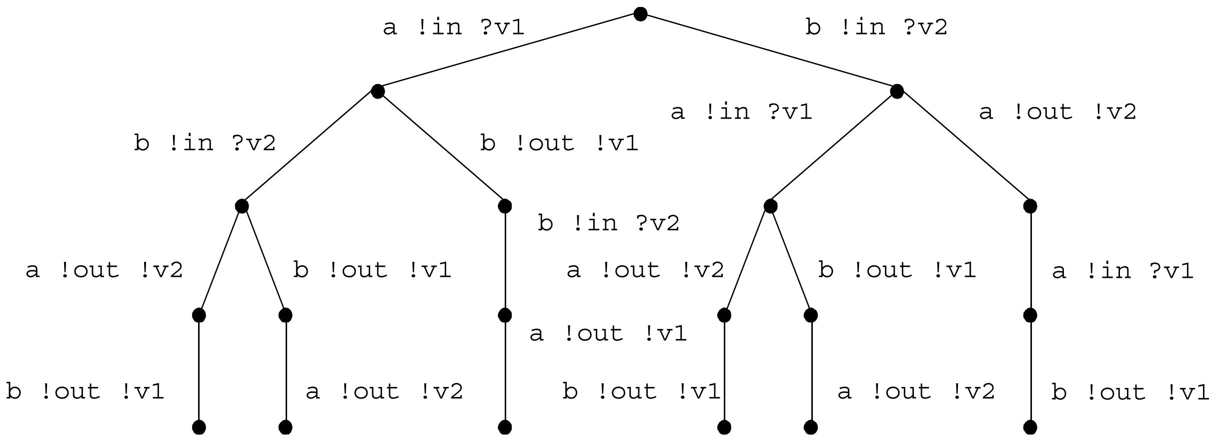


Figure 3.7: Behaviour tree of the bi-directional buffer example

constraints due to interleaving semantics is very undesirable, and it is in practice either informally relaxed or unnecessarily built. In the former it corrupts the objective of unambiguous interpretation of using a formal language, and in the latter it may generate low quality solutions (for example, with bad performance).

Behaviour definitions using causality and exclusion allows one the distinguish between independent and interleaved actions. Section 3.4.2 discusses this in more detail and revisits this example.

3.2.7 Comparison with LOTOS

Below we compare some causality relations with possible ‘equivalent’ LOTOS specifications. This comparison is an approximation, since LOTOS has an interleaving semantics and has actually been designed to represent interactions, not actions[1].

The causality relation $a_1 \wedge a_2$ -> a_3, imposing that a_1 and a_2 are independent of each other, could be expressed in the following way in LOTOS:

```
( a1; exit ||| a2; exit ) >> a3; stop
```

The causality relation $a_1 \vee a_2$ -> a_3, imposing that a_1 and a_2 are independent of each other, could be expressed in the following way in LOTOS:

```
hide sync in
( a1; sync; stop ||| a2; sync; stop ) |[sync]| sync; a3; stop
```

A LOTOS specification of this causality relation in a rather structured way such as above requires the introduction of some form of internal communication through internal gates. Recalling that the use of internal gates characterize a resource-oriented style (see [6]), which should be used to represent internal behaviour structure and possibly a certain structure of functional entities, we conclude that this internal gate obscures the interpretation of the specification.

1. Actions can correspond to internal events in LOTOS in some cases. We ignore this possibility in this comparison.

Another possibility would be to define this behaviour in terms of all the traces which result in the desired condition. In this case the structure of the original requirements (a_3 depending on a_1 and a_2) would have been completely ruined.

The causality relation $a_1 \wedge \neg a_2$ -> a_3, imposing that a_1 and a_2 are independent of each other, could be expressed in the following way in LOTOS:

```
( a1; a3 ; stop )|[a3]| ( a3 ; stop [> a2; stop )
```

The causality relation $a_1 \vee \neg a_2$ -> a_3, imposing that a_1 and a_2 are independent of each other, could be expressed in terms of the possible traces in the following way in LOTOS:

```
a1; ( a2; stop ||| a3; stop )
[] a2; a1 ; a3; stop
[] a3 ; ( a1; stop ||| a2; stop)
```

A more structured specification does not seem to be possible in this case.

The impossibility of representing these rather appealing behaviour patterns without using internal gates, exit constructs or possible traces, and the unpleasant consequences of these constructs shows that the design model of LOTOS is unsuitable for a straightforward representation of causality relations. In some cases the essence of a causality relation is obscured in the LOTOS ‘equivalent’ behaviour expression by complex language constructs, which could even be incorrectly interpreted as implementation prescriptions. This is especially the case in the representation of *or*-conditions, which is only possible with the introduction of hidden gates or using traces.

3.3 Attributes in Causality Relations

Action attributes in causality relations enable the representation of more complex relationships between actions. Action attributes can play two distinct roles, depending whether an action is a condition or a resulting action in a causality relation:

1. for actions in a condition of a causality relation, specific attribute values define the conditions for the resulting action to occur. These attributes are not constrained in this causality relation, but can be referred to by the resulting action;
2. for a result action of a causality relation, its attributes may be constrained such that only specific values are allowed. These constraints may involve references to attribute values of the actions in the condition.

The use of action attributes as in item 1 above is comparable to the LOTOS guard, since it defines conditions that have to satisfied prior to the occurrence of an action. The use of action attributes as in item 2 above is comparable to the LOTOS selection predicate, since it defines constraints that apply to the action being established.

We discuss below, by means of examples, some possible ways action attributes can be used to define conditions and constraints in causality relations.

3.3.1 Values of Information

The occurrence of actions with specific action values can be used as conditions for the occurrence of an action. Consider the following causality relation:

$$a_1\ (v_1\text{: } Nat)\ [v_1 < 10]\ \text{->}\ a_2$$

This causality relation states that only in case a_1 happens with value v_1*: Nat* (of type natural number) smaller than *10*, a_2 is allowed to happen. In case a_1 happens with v_1 greater or equal to *10*, the condition for a_2 is not true, and a_2 is not allowed to happen.

Attributes of an action can be also constrained in its causality relation, defining in this way a set of allowed action values. Consider the following causality relation:

$$a_1\ (v_1\text{: } Nat)\ \text{->}\ a_2\ (v_2\text{: } Nat)\ [v_2 = v_1 + 2]$$

In this case a_2 can only happen for v_2 equal to $v_1 + 2$. Another example could be:

$$a_1\ (v_1\text{: } Nat)\ \text{->}\ a_2\ (v_2\text{: } Nat)\ [v_1 < v_2 < 100]$$

In this example a_2 can only happen for v_2 bigger than v_1 and smaller than *100*. Action a_2 does not a value of v_2 that does not comply to these conditions.

Attributes of more than one action can be used to define the conditions and constraints for an action to occur. Consider the following example:

$$a_1\ (v_1\text{: } Nat) \wedge a_2\ (v_2\text{: } Nat)\ [v_1 > v_2]\ \text{->}\ a_3\ (v_3\text{: } Nat)\ [v_1 > v_3 > v_2]$$
$$a_1\ (v_1\text{: } Nat) \wedge a_2\ (v_2\text{: } Nat)\ [v_1 < v_2]\ \text{->}\ a_4\ (v_4\text{: } Nat)\ [v_1 < v_4 < v_2]$$

In this example, actions a_3 and a_4 depend on the values of v_1 and v_2 established in a_1 and a_2, respectively. In case $v_1 = v_2$ neither a_3 nor a_4 happen. The conditions for the occurrence of a_3 and a_4, and the constraints on the values v_3 and v_4 depend on values of information established in more than one action (a_1 and a_2). This can be generalized to an arbitrary number of actions.

In causality relations of the form $a_1 \vee a_2$ -> a_3 we may need to refer to different values and in different ways in case the actual cause of a_3 is a_1 or a_2. Consider the following causality relation:

$$\begin{aligned}&a_1\ (v_1\text{: } Nat) \vee a_2\ (v_2\text{: } Nat)\ \text{->}\ a_3\ (v_3\text{: } Nat)\\ &\qquad [\text{if } a_1 \text{ then } v_3 = f(v_1)\\ &\qquad \text{if } a_2 \text{ then } v_3 = g(v_2)]\end{aligned}$$

In this example, if a_1 is the cause of a_3, v_3 is a function *f* of v_1, and if a_2 is the cause of a_3, v_3 is a function *g* of v_2. Since a_3 cannot be caused by both a_1 and a_2, value v_3 cannot refer to both values v_1 and v_2 at the same time.

Value Conditions in Exclusion

Often we have to express the condition that the non-occurrence of some specific action with some attribute values is a condition for the occurrence of another action. Consider the more concrete example of an action a_2, which can only happen if another action a_1 has not happened with value $v_1 < 10$. We could have defined this condition such as $\neg\, a_1\ (v_1\!: Nat)\ [v_1 < 10]$, but the problem is that we would be defining conditions on an action that possibly does not happen, which is not appealing to a designer's intuition.

We have solved this problem by imposing the restriction that conditions on and references to attributes of actions appearing in exclusion in causality relations are not allowed. This restriction makes the reference to action attributes systematic, without decreasing the expressive power of our design model. In order to comply to this restriction the condition in the example above should be indicated in the following way:

$$a_1\ (v_1\!: Nat)\ [v_1 \geq 10] \vee \neg\, a_1 \text{ -> } a_2$$

This means that if (i) a_1 happens with value v_1 greater than or equal to 10 or (ii) a_1 does not happen at all, a_2 is allowed to happen. The statement above also means that if both a_1 with $v_1 < 10$ and a_2 ever happen in a run of the system, then a_1 with $v_1 < 10$ happens after a_2.

Retained Values

Retained values are used to allow an action to refer to values of information established in other actions that do not cause this action directly. Consider the following example:

$$a_1\ (v_1\!: Nat) \wedge a_2\ (v_2\!: Nat)\ [v_1 > v_2] \text{ -> } a_3\ (v_3\!: Nat, v_1\!: Nat)\ [v_1 > v_3 > v_2]$$
$$a_3\ (v_3\!: Nat, v_1\!: Nat) \text{ -> } a_4\ (v_4\!: Nat)\ [v_1 < v_4 < v_3]$$

In this example, action a_4 refers to the value v_1 established in a_1, but it does not depend directly on a_1. However this reference is possible since a_3 is capable of keeping the value of v_1 in its retained values for later reference by a_4. Value v_1 is not established by a_3, but the availability of v_1 is indicated by a_3. Conditions on retained values are also possible.

3.3.2 Time

In many instances of design, timing requirements are critical and have to be explicitly represented. Examples of these requirements are orderings between actions, intervals of time when actions can occur, given points in time when actions can occur, or combinations of these requirements ([2]). Orderings between actions are implicitly defined in causality relations, since causality and exclusion impose specific orderings between actions. Time intervals and given points of time when actions can occur can be defined using constraints on time attributes.

In contrast to models in which timing requirements are not considered, it is not enough to implement causality relations by producing the prescribed values and behaviour patterns. These values and patterns must also be produced according to the timing requirements. This adds another dimension to the criteria that determine whether a certain behaviour is a correct implementation of an abstract one.

Time intervals and given points of time are two forms of timing requirements; we call them asynchronous and synchronous timing requirements, respectively. We consider the latter as a special case of the former in this work.

Asynchronous Timing Requirements

Designers should be able to define the time interval in which actions are supposed to happen, without having to consider the specific mechanisms that implement these actions and the timing characteristics of these mechanisms. This can be done by defining intervals in the timing constraints of an action, which determine the periods of time in which an action can happen. Since the time attribute of an action defines the moment of occurrence of this action, timing constraints actually define the possible values of the time attribute of the action.

Consider the following causality relation:

$$a_1\ (t_1\text{: Time}) \text{ -> } a_2\ (t_2\text{: Time})\ [t_2 < t_1 + 10]$$

This causality relation defines that a_2 is allowed to happen after a_1 has happened (at t_1), but before $t_1 + 10$. The implicit timing conditions of this causality relation determine that $t_2 > t_1$. Therefore this timing constraint defines the interval $]\ t_1, t_1 + 10[$, into which a_2 is allowed to happen. This requirement determines that the mechanism that implements this causality relation may delay to a maximum of *10* time units, but not more.

Another example of asynchronous timing requirement is the following:

$$a_1\ (t_1\text{: Time}) \text{ -> } a_2\ (t_2\text{: Time})\ [t_2 > t_1 + 10]$$

This causality requirement defines that a_2 is allowed to happen in case a_1 happens (at t_1), but from $t_1 + 10$ on. This timing constraint defines the interval $]\ t_1 + 10, \infty[$, into which a_2 is allowed to happen. This requirement determines that the mechanism that implements this causality relation must have a minimum delay of *10* time units, after which a_2 is allowed to happen.

Timing requirements can be defined such that arbitrary intervals in which actions are allowed to happen are indicated. The following causality relation illustrates the definition of an arbitrary time interval for the occurrence of an action:

$$a_1\ (t_1\text{: Time}) \text{ -> } a_2\ (t_2\text{: Time})\ [t_1 + 5 < t_2 < t_1 + 10]$$

This causality relation states that a_2 can occur in the interval $]\ t_1 + 5, t_1 + 10[$, in case a_1 happens. Figure 3.8 depicts the timing diagrams of the examples above.

Implementations that conform to the constraint $[t_1 + 5 < t_2 < t_1 + 10]$ also conform to the constraint $[t_1 < t_2 < t_1 + 10]$, such that the former can be considered as a refinement of the latter. The timing constraint $[t_1 < t_2 < t_1 + 10]$ may have been defined in an early stage in the design process, in which specific knowledge of implementation mechanisms for the causality relation

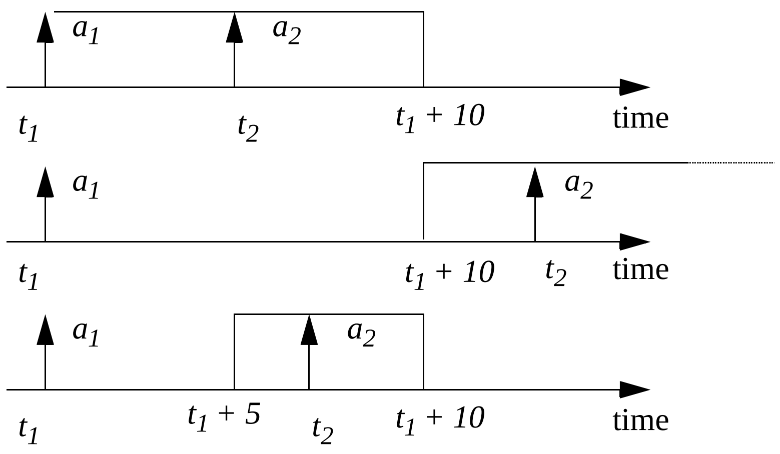


Figure 3.8: Examples of asynchronous timing constraints

between a_1 and a_2 was not available. The timing constraint $[t_1 + 5 < t_2 < t_1 + 10]$ may replace the former timing constraint, after a refinement step in which the implementation knowledge that a minimum of 5 time units is necessary to implement the causality relation is introduced. In this case, requirements and consideration of implementation characteristics have been combined in a single statement.

The examples above show that some time moments have to be considered in the definition and interpretation of asynchronous timing constraints: (i) the moment in which the conditions for the occurrence of an action become true, t_c, (ii) the beginning of the time period in which the action can occur, t_l, and (iii) the end of this time period, t_u. Actually more than one period in which the action can occur can be defined, which means that more instances of (ii) and (iii) may be defined.

Consider the following causality relation:

$$a_1\ (t_1\text{: Time}) \wedge a_2\ (t_2\text{: Time}) \text{ -> } a_3\ (t_3\text{: Time})$$
$$[\max\ (t_1, t_2) + 2 < t_3 < \max\ (t_1, t_2) + 5]$$

In this example, t_c is the moment in which the latest of a_1 or a_2 happen, or both a_1 and a_2 if they happen at the same time, t_l is the next time moment bigger[2] than $\max\ (t_1, t_2) + 2$ and t_u is the next time moment smaller[2] than $\max\ (t_1, t_2) + 5$.

Figure 3.9 depicts the timing pattern of asynchronous timing requirements and its application on the example above.

Synchronous Timing Requirements

Designers may need to define specific time moments at which actions can occur. Consider the following causality relation:

2. Since we have been considering a dense time domain (positive real numbers), t_l and t_u theoretically tend to $\max\ (t_1, t_2) + 1$ and $\max\ (t_1, t_2) + 5$, respectively. This should not be a problem, considering that our time observations are bound to some precision anyway.

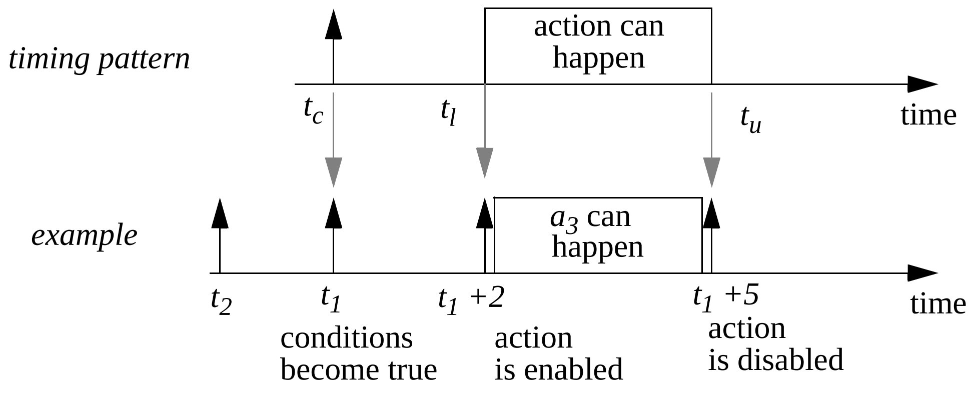


Figure 3.9: Timing pattern for the interpretation of timing constraints

a_1 $(t_1$: *Time*$)$ -> a_2 $(t_2$: *Time*$)$ $[t_2 = t_1 + 10]$

This causality relation defines that a_2 is allowed to happen ‘exactly’ *10* time units after a_1 happens. The degree of precision assigned to this timing requirement may depend on the specific application. In implementations of this causality relation, a_2 is actually expected to happen in an interval $t_1 + 10 - \delta$, $t_1 + 10 + \delta$, where δ defines the precision. Since we want to be able to represent this condition for different applications, we abstract from timing precision in this text.

Comparing synchronous timing requirements to asynchronous timing requirements, we notice that the former is a special instance of the latter, in which the interval in which an action can happen has been reduced to a single time moment, i.e. $t_l = t_u$ ($= t_1 + 10$ in the example). In this case we say that there is *synchronization*, meaning that a_2 is supposed to happen synchronously at $t_2 = t_1 + 10$.

Figure 3.10 depicts the relationship between synchronous and asynchronous timing requirements.

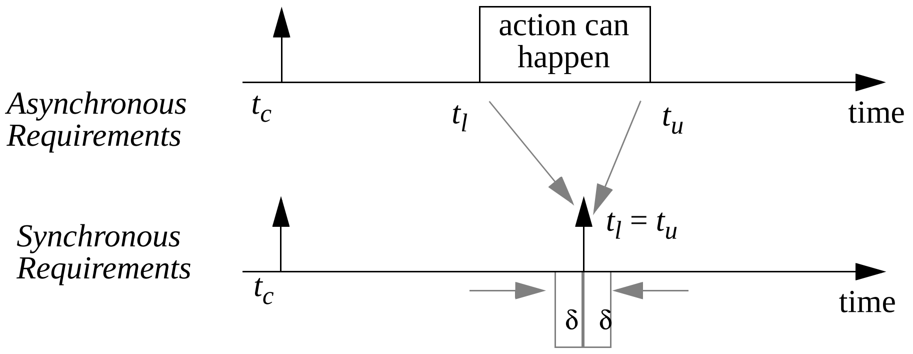


Figure 3.10: Asynchronous and synchronous timing requirements

By generalizing the conditions above we can define causality relations that allow a large number of real-time requirements to be expressed, for example by constraining actions to occur at a collection of intervals, at a collection of time moments, etc.

3.3.3 Probability or Modality Requirements

Another important aspect of the definition of an action is whether this action may or must occur in case it is enabled. Until now we have considered that an action can only occur in case it is enabled, which does not imply that actions that are enabled must necessarily happen. However, in many realistic instances of design one should be able to distinguish between actions that must surely happen or that may happen, and possibly even assign probabilities to the occurrences of actions. The probability attribute is defined for this purpose.

Consider the following causality relation:

$$a_1 \text{ -> } a_2\ (p_2\text{: Prob})\ [p_2 = 100\%]$$

In this example, in case a_1 happens, a_2 must happen sometime after a_1. Actually such requirement is of limited use in realistic system design, since theoretically a_2 may happen much later (years, centuries, etc.) than a_1. A more practical example would then be the following:

$$a_1\ (t_1\text{: Time}) \text{ -> } a_2\ (t_2\text{: Time}, p_2\text{: Prob})\ [t_2 < t_1 + 10, p_2 = 100\%]$$

This causality relation states that in case a_1 happens (at time t_1), a_2 must happen before $t_1 + 10$. Supposing this causality relation represents the behaviour of an instance of a connectionless service, this corresponds to a reliable service with a maximum delay of *10* time units.

Consider the following causality relation:

$$a_1\ (t_1\text{: Time}) \text{ -> } a_2\ (t_2\text{: Time}, p_2\text{: Prob})\ [t_2 = t_1 + 10, p_2 = 100\%]$$

This causality relation states that in case a_1 happens (at time t_1), a_2 *must* happen at time moment $t_1 + 10$.

So far we have assumed that in case we do not mention the probability attribute value of an action explicitly, this action may happen if it is enabled. However this is the same as defining that this action has a certain probability attribute value smaller than *100%*. Defining explicit values for the probability attribute of an action, we can indicate the probability of an action to occur in case its conditions are fulfilled. Consider the following example:

$$a_1\ (t_1\text{: Time}) \text{ -> } a_2\ (t_2\text{: Time}, p_2\text{: Prob})\ [t_2 < t_1 + 10, p_2 = 90\%]$$

This causality relation states that in case a_1 happens (at time t_1), a_2 may happen before $t_1 + 10$, with probability *90%*. If a_2 does not happen before $t_1 + 10$ it does not happen at all. This means that in some instances of execution of this causality relation, the occurrence of a_1 does not imply the occurrence of a_2. Supposing this causality relation represents the behaviour of an instance of a connectionless service, this corresponds to an unreliable service with a maximum delay of *10* time units and a loss probability of *10%*.

The probability attribute of an action can be defined in terms of the classical definition of probability: the ratio between the instances of execution in which the action occurs given that its conditions are fulfilled and the instances of execution in which the action does not occur given that its conditions are fulfilled tends to the value of the probability attribute of the action, in case we would be able to execute the same behaviour infinitely many times. Although it can be defined in terms of infinite repetitions, the probability attribute of an action is a characteristic of a single causality relation, and it applies to a single instance of the resulting action.

In case the probability attribute of an action is *100%*, which characterizes a *must* causality relation, this action certainly occurs according to its constraints if its conditions are fulfilled. In case the probability attribute of an action is smaller than *100%*, which characterizes a *may* causality relation, there is no guarantee that this action happens if it is conditions are fulfilled. However in a may causality relation an action happens according to its constraints, or it does not happen at all. Therefore the probability attribute of an action is a conditional probability, since it represents the probability that an action happens in case its conditions are fulfilled. The distinction between must causality relations, and may causality relations without specific probability values is called *modality.*

A may causality relation expresses some form of *non-determinism*, since it represents uncertainty about the occurrence of an action. This non-determinism can be used to conceal uncertainty about implementation components such as memory availability, bandwidth, delays, reliability, etc.

Probability attributes also allow a neater representation of non-determinism, if we compare causality relations to process algebras. For example, the loss of a message in a connectionless service can be defined without relating it to any kind of obscure internal event, as it is necessary in process algebra models of behaviour. With the use of probability attributes the loss of a message can be represented by the probability that the delivery of this message does not happen.

Probability requirements can get much more complex than in the examples presented before. For example in $a_1 \vee a_2 \text{ -> } a_3$ we could have different probabilities for a_3, depending on its actual cause (a_1 or a_2). We may also have different probabilities depending on the values of information, timing, etc. We refrain from discussing these complications for the time being.

Relationship with Other Attributes

By inspecting the examples above one can conclude that it is possible to define and manipulate probability requirements separately from timing requirements. In a more detailed model of action attributes one could consider the relationship between probability and timing requirements, by defining the distribution of the probability that a certain action happens before a certain period of time. In this thesis we do not address the distribution of probability in time, since these distributions are normally known to existing systems, but have to guessed for systems being designed. Furthermore it involves techniques that fall outside the scope of this thesis, and is left for further study.

Conditions and constraints on values of information are not effected by probability requirements. It is fair to say that in most causality relations probability requirements can be superposed to requirements on the other attributes, without disturbing these requirements.

3.3.4 General Comments

This section discusses the definition of conditions or constraints on more than one attribute type in a single causality relation, implementability aspects of causality relations and non-determinism in causality relations.

Generic Causality Relations

Location attributes are treated as a specific value sort. We suppose the existence of a data sort, which represents a location, and the existence of a set of location values, which represent the physical or logical locations in a design. The location attribute of an action is a location value of this sort and from this set, which can be assigned to an action as a constraint on its occurrence.

In general, conditions and constraints on different action attributes can be combined in a single causality relation. Consider the following example:

$$\begin{aligned}&a_1\ (l_1\text{: } Loc,\ v_1\text{: } Nat,\ t_1\text{: } Time)\ [v_1 > 10] \to\\ &\quad a_2\ (l_2\text{: } Loc,\ v_2\text{: } Nat,\ t_2\text{: } Time)\\ &\qquad [l_2 = a \wedge v_2 = v_1 - 10 \wedge t_2 < t_1 + 10 \wedge p_2 = 90\%]\end{aligned}$$

In this causality relation, a_2 can only happen if a_1 happens with $v_1 > 10$. Furthermore v_2 can only be equal to v_1 - 10 and a_2 happens at a moment t_2 smaller than $t_1 + 10$ at location a and with probability 90%. Action a_2 does not happen with time and value attributes that do not comply to these constraints.

Summarizing, for an action to occur, its conditions, as defined in its causality relation, must be fulfilled. Furthermore an action occurs at the time and location indicated by its time and location attribute, respectively, and according to their constraints, with the probability defined by probability attribute and making the values indicated by its functionality (action values and retained values) available according to their constraints.

Implementability

Since we are interested in specifying only those systems that can be built, conditions and constraints on attributes have to be evaluated in terms of their implementability. For example the following causality relations do not satisfy the implementability criterion:

$$a_1\ (v_1\text{: } Nat) \to a_2\ (v_2\text{: } Nat)\ [v_1 +10 < v_2 < v_1 +10]\ \text{(* impossible value *)}$$

$$\begin{aligned}&a_1\ (t_1\text{: } Time) \to a_2\ (t_2\text{: } Time)\ [t_1 + 15 < t_2 < t_1 + 10]\\ &\qquad\qquad \text{(* empty time interval *)}\end{aligned}$$

$$\begin{aligned}&a_1\ (t_1\text{: } Time) \to a_2\ (t_2\text{: } Time)\ [t_1 = t_2]\\ &\qquad\qquad \text{(* violates implicit condition that } t_2 > t_1 \text{ *)}\end{aligned}$$

It is the explicit responsibility of the designer to define conditions that can be met by available implementation components, possibly after consulting the implementers. For example if designers define that $a_1\ (t_1\text{: Time}) \text{ -> } a_2\ (t_2\text{: Time})\ [t_2 < t_1 + 10\ \mu s]$, they must know that it is feasible to build a mechanism that performs a_2 within $10\ \mu s$ after a_1 happens. Therefore timing requirements should be considered as a prescription to the mechanism that implements this causality relation, influencing and constraining the implementer's choices.

Non-determinism versus Choice of Implementation

Causality relations define the conditions which allow an action to occur and its allowed attribute values. However, during the design process, causality relations in which multiple alternative attribute values are defined may be interpreted in two different ways by implementers:

1. all the alternative attribute values can possibly be established by the implementation of the causality relation;
2. the implementer may select attribute values, such that some of the alternative values defined in the causality relation will never be established by its implementation.

We illustrate these two interpretations with the following example of causality relation:

$$a_1 \text{ -> } a_2\ (v_2\text{: Nat})\ [3 < v_2 < 9]$$

In this example, according to interpretation (1) an implementation in which we know that v_2 can only get value *4* would not be considered correct, although it complies to the constraint $3 < v_2 < 9$. However, the same implementation is acceptable according to interpretation (2).

Another way of tackling these alternative interpretations is to relate them to (1) non-determinism and (2) choice of implementation, respectively. According to interpretation (1) this causality relation would define a non-deterministic choice at the action a_2 on a value the complies to $3 < v_2 < 9$. According to interpretation (2) this causality relation defines a set of options, where sub-sets of the allowed values are established, such that the implementer may choose, for instance, for an option in which value *4* is established, or for another one in which values *4* or *5* can be established. Alternative (1) defines a single relation between a_1 and a_2, while alternative (2) defines a set of optional relations, from which the implementer has to select one.

Ideally one could think of a notational way to distinguish statements that denote a single behaviour from statements that denote alternative sets of behaviours early in the design process, but this may be impractical, since it may make the interpretation of causality relations too strict. One way of coping with this problem is to define implementation relations for the refinements of such causality relations that reflect these interpretations. For example by not accepting or accepting restrictions on the number of possible attribute values one can give the interpretation of non-determinism or choices of implementation to causality relations, respectively. Such implementation relations are discussed in Chapter 6.

3.4 Finite Behaviour Definitions

The behaviour of a functional entity defines possible orderings of its actions. However, since we are interested in the relationships between the actions of a behaviour and in how these relationships determine the possible orderings between these actions, we define behaviour in terms of these relationships. These relationships correspond to the causality relations.

3.4.1 Representation of Behaviour Elements

Finite behaviour can be represented by a set of causality relations, one relation per action of the behaviour. The representation of each of the behaviour elements identified in section 3.1.2 is discussed in the sequel.

Initial Actions

Some actions may be enabled from the beginning of the behaviour (initial actions), while some others may depend on actions of the behaviour in order to be enabled. We denote the condition of being enabled from the beginning of a behaviour by *start*, which corresponds to a *true* condition.

Consider the following example:

$$\begin{aligned} B := \{\ & start \rightarrow a_0, \\ & start \wedge \neg a_0 \rightarrow a_1, \\ & a_1 \vee a_0 \rightarrow a_2, \\ & a_1 \wedge \neg a_2 \rightarrow a_3, \\ & a_2 \rightarrow a_4\ \} \end{aligned}$$

$start \rightarrow a_0$ states that a_0 is enabled from the beginning of the behaviour, such that it does not have to wait for other actions in order to be enabled. $start \wedge \neg a_0 \rightarrow a_1$ implies that while a_0 does not happen a_1 is allowed to happen, therefore at the beginning of the behaviour a_1 is also enabled. This means that both a_0 and a_1 are initial actions of B.

Entry points in behaviours are discussed in Chapter 4.

Causality Contexts

The behaviour definition B in terms of causality relations determines the causality context of all its actions. Although Figure 3.2 may give the impression that the causality contexts of actions in a behaviour are disjoint, this is not necessarily the case.

Figure 3.11 depicts the example above and the causality contexts of a_1 and a_2.

In Figure 3.11, a_2 is the resulting action in its causality relation, and appears in the causality relations of a_3 and a_4. Therefore the causality context of a_2 is $<a_1 \vee a_0 \rightarrow a_2, \{a_1 \wedge \neg a_2 \rightarrow a_3, a_2 \rightarrow a_4\}>$, defining the conditions for a_2 to occur, and the actions that are influenced by a_2, which defines the role of a_2 in behaviour B. The causality context of a_1 is $<start \wedge \neg a_0 \rightarrow a_1, \{a_1 \vee a_0 \rightarrow a_2, a_1 \wedge \neg a_2 \rightarrow a_3\}>$, which means that the causality contexts of a_1 and a_2 overlap.

In this form of behaviour definition, all the conditions for the occurrence of an action are stated in a single statement. All necessary and sufficient conditions for a certain action to take place are stated once and for all in its causality relation. This form of behaviour definition is comparable to

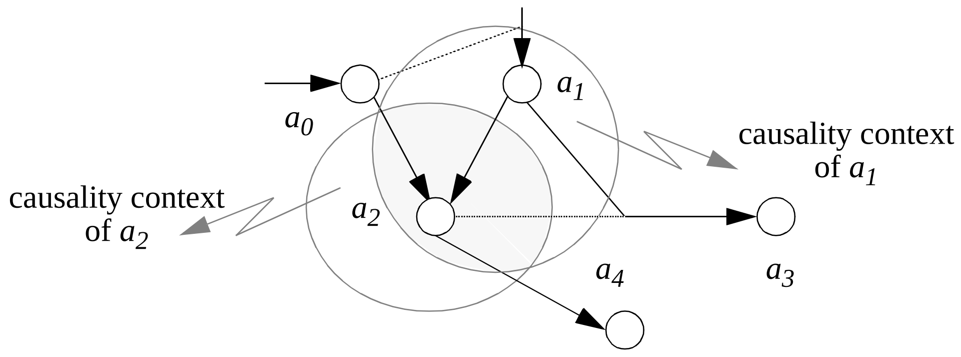


Figure 3.11: Overlap in causality contexts

the *monolithic specification style* of [6], in which a behaviour is described as a monolithic whole, without any structuring.

Causality relations are a compact and parsimonious notation to define relationships between actions, but lack structuring and may become difficult to comprehend and manipulate. Furthermore this notation is restricted to finite behaviours.

Termination Actions

Termination actions are those actions after which no behaviour follows. Termination actions can be explicitly defined, although this should not be mandatory. The explicit definition of these conditions could be used to enhance readability. In Figure 3.11, for example, we could have included the statements a_3 -> *stop* and a_4 -> *stop*, to indicate that no behaviour follows from a_3 and a_4.

Behaviour exits are discussed in Chapter 4.

3.4.2 Expressive Power

The capability of representing behaviour patterns that can be intuitively understood or that can be represented in available formal description techniques, can give some indication on the expressive power of causality relations. Below we identify and represent some behaviour patterns using causality relations, aiming at evaluating the suitability of causality relations to express these behaviour patterns, and the degree of generality that can be obtained by using causality relations.

Sequential Ordering

This behaviour pattern represents the situation in which actions occur one after another, in a specific order. Consider the following behaviour definition:

$$\begin{aligned} B := \{ \; & start \to a_0, \\ & a_0 \to a_1, \\ & a_1 \to a_2 \; \} \end{aligned}$$

This behaviour defines that after a_0 happens, a_1 is allowed to happen and that after a_1 has happened a_2 is allowed to happen. Nothing else happens after a_2. The statement a_1 -> a_2 alone does not imply that the *first* action that happens after a_1 is a_2. This is a consequence of the uniqueness of actions and that a_0, a_1 and a_2 are the only defined actions of B.

Choice

This behaviour pattern represents that at some point in time there is a choice between actions that may occur, such that once one of the actions involved in the choice occur, the others are not allowed to occur any more. We illustrate this behaviour pattern with the following behaviour definition:

$$\begin{aligned} B := \{\ & start \text{ -> } a_0, \\ & a_0 \wedge \neg\, a_1 \text{ -> } a_2, \\ & a_0 \wedge \neg\, a_2 \text{ -> } a_1\ \} \end{aligned}$$

The interpretation of $a_0 \wedge \neg\, a_1$ -> a_2 is that a condition for a_2 to occur at t_2 is that a_0 happens and a_1 does not happen before or at time t_2. This causality relation does not say anything about the conditions for the occurrence of a_1, which are stated in $a_0 \wedge \neg\, a_2$ -> a_1.

Arbitrary Interleaving

This behaviour pattern defines that some actions are not allowed to occur at the same time. The following behaviour definition illustrates this behaviour pattern for two actions:

$$\begin{aligned} B := \{\ & start \text{ -> } a_0, \\ & a_0 \wedge (a_1 \vee \neg\, a_1) \text{ -> } a_2, \\ & a_0 \wedge (a_2 \vee \neg\, a_2) \text{ -> } a_1\ \} \end{aligned}$$

The interpretation of $a_0 \wedge (a_1 \vee \neg\, a_1)$ -> a_2 is that a condition for a_2 to occur is that either (i) a_1 happens before a_2 or (ii) a_1 does not happen before nor at the same time as a_2. This causality relation does not state anything about the conditions for a_1, which are defined in $a_0 \wedge (a_2 \vee \neg\, a_2)$ -> a_1.

This example shows that arbitrary interleaving corresponds to a sort of mutual causality on occurrence, namely if two actions a_1 and a_2 are interleaved, either a_1 is a cause of a_2 or vice-versa. In some cases we could even expect that arbitrarily interleaved actions would refer to each others functionality. A causality relation such as $a_0 \wedge (a_2 \vee \neg\, a_2)$ -> a_1 allows a_1 to refer to an attributes of a_2, in case a_2 occurs before a_1.

A more concrete example of reference to attribute values in interleaved actions is the selection of connection identifiers in connection-oriented services. In this case the connect actions in which connection identifiers are selected have to be interleaved, allowing them to refer to each others functionality, namely the set of available connection identifiers and the selected value, in order to select connection identifiers that have not been selected yet.

Causality relation $a_0 \wedge (a_1 \vee \neg\, a_1)$ -> a_2 implies that a_1 and a_2 will never occur at the same time, even if this is allowed in the definition of a_1. Therefore the extra condition $(a_2 \vee \neg\, a_2)$ in the causality relation of a_1 could be considered an over-specification. However, omitting this condition would give a false impression that this constraint does not apply.

Disabling

This behaviour pattern defines that the occurrence of some actions disables the occurrence of some other actions. We illustrate this behaviour pattern with the following behaviour definition:

$$\begin{aligned} B := \{\ & start \text{ -> } a_0, \\ & a_0 \wedge \neg\, a_1 \text{ -> } a_2, \\ & a_0 \text{ -> } a_1\ \} \end{aligned}$$

The interpretation of $a_0 \wedge \neg\, a_1$ -> a_2 is that a condition for a_2 to occur is that a_1 has not occurred. In case a_1 happens before a_2 has happened, a_2 is not allowed to happen any more (one condition of a_2 becomes false). a_1 is allowed to happen at any time after a_0, which is stated in a_0 -> a_1.

Independence

This behaviour pattern defines that there is no relation whatsoever between certain actions, so that each of these actions can occur at any time, independently of the others. We illustrate this behaviour pattern involving two actions, with the following behaviour definition:

$$\begin{aligned} B := \{\ & start \text{ -> } a_0, \\ & a_0 \text{ -> } a_1, \\ & a_0 \text{ -> } a_2\ \} \end{aligned}$$

In this behaviour definition, a_1 and a_2 have no relation with each other, since a_2 is not mentioned in the causality relation of a_1 and vice-versa. Therefore a_1 and a_2 can happen independently of each other.

Summary

We recognize that sequential composition on one hand, and independence, disabling and choice on the other hand, form two different categories of behaviour patterns. Sequential composition is directly related to causality, while disabling and choice are related to exclusion and independence is based on the absence of causality or exclusion. In the case of sequential composition, reference to value attributes is possible, while in the other behaviour patterns reference to value attributes is not possible. Arbitrary interleaving combines causality and exclusion.

Independence, arbitrary interleaving, disabling and choice impose, in this order, increasingly severe constraints on the occurrence of a_1 and a_2: no restriction, not at the same time, not before nor at the same time, never if the other happens. Furthermore these restrictions have been formulated here as special cases of a single causality relation mechanism, in the same way value checking, value passing and value generation are expressed as special cases of a single interaction mechanism in [5].

Summarizing:

- *independence*: a_1 and a_2 are completely unrelated (independent of each other) and can both occur at any time;
- *arbitrary interleaving*: a_1 and a_2 are not allowed to occur at the same time moment, but both can still occur;
- *disabling*: a_2 is only allowed to occur if a_1 does not occur before it nor at the same time;
- *choice*: a_1 or a_2 can occur, but once one of them has happened, the other is not allowed to occur any more.

Figure 3.12 depicts these behaviour patterns using our graphical notation. In Figure 3.12 the conditions $a_0 \wedge (a_1 \vee \neg a_1)$ and $a_0 \wedge (a_2 \vee \neg a_2)$ are represented in terms of their equivalent forms $(a_0 \wedge a_1) \vee (a_0 \wedge \neg a_1)$ and $(a_0 \wedge a_2) \vee (a_0 \wedge \neg a_2)$, respectively.

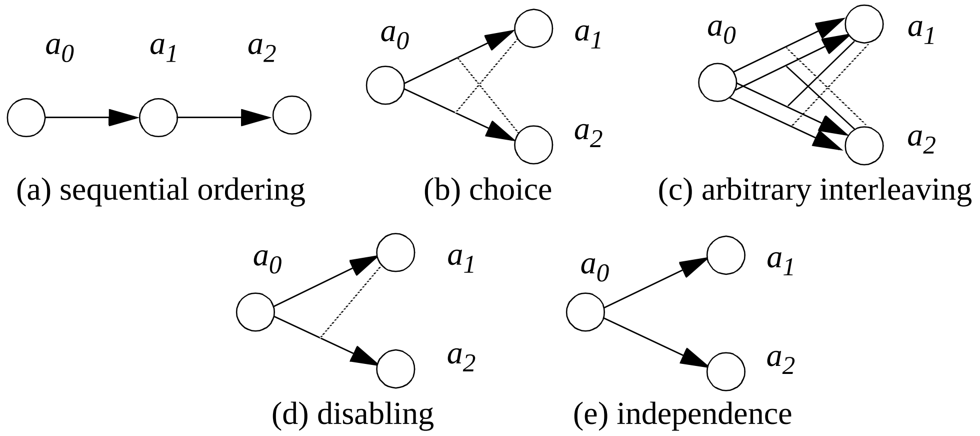


Figure 3.12: Some behaviour patterns

We conclude that some well-known behaviour patterns can be represented using causality relations. Furthermore causality relations allows one to explicitly distinguish between arbitrary interleaving and independence.

Bi-directional Buffer Example (Revisited)

The behaviour bi-directional buffer example of Figure 3.6 can be defined using causality relations in the following way, where in_a and in_b represent the input of data at locations *a* and *b* respectively, and out_a and out_b represent the output of data at locations *a* and *b* respectively, and *Loc* and *Val* represent the location and the value sorts of these actions, respectively.

$$
\begin{aligned}
&BiBuff := \{\ start \wedge (\neg\, out_a \vee out_a\,(l_1\!: Loc, v_1\!: Val)) \;\text{->}\\
&\qquad in_a\,(l_a\!: Loc, v_a\!: Val)\ [l_a = a],\\
&\quad start \wedge (\neg\, out_b \vee out_b\,(l_1\!: Loc, v_1\!: Val))\\
&\qquad \text{->}\ in_b\,(l_b\!: Loc, v_b\!: Val)\ [l_b = b],\\
&\quad in_a\,(l_a\!: Loc, v_a\!: Val) \wedge (\neg\, in_b \vee in_b\,(l_1\!: Loc, v_1\!: Val))\\
&\qquad \text{->}\ out_b\,(l_b\!: Loc, v_b\!: Val)\ [(l_b = b) \wedge (v_b = v_a)]\ ,\\
&\quad in_b\,(l_b\!: Loc, v_b\!: Val) \wedge (\neg\, in_a \vee in_a\,(l_1\!: Loc, v_1\!: Val))\\
&\qquad \text{->}\ out_a\,(l_a\!: Loc, v_a\!: Val)\ [(l_a = a) \wedge (v_a = v_b)]\ \}
\end{aligned}
$$

Figure 3.13 shows the graphical representation of this behaviour, ignoring the attribute values.

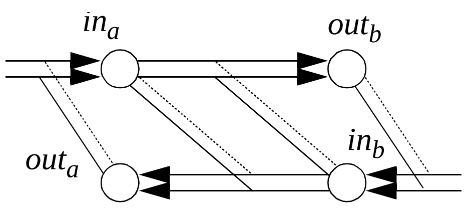


Figure 3.13: Behaviour of the bi-directional buffer in terms of causality relations

In Figure 3.13, actions in_a and out_a (in_b and out_b) are interleaved, while actions in_a and in_b (out_a and out_b) are independent. These two difference relations between these actions could not be identified in the behaviour tree of Figure 3.7.

3.4.3 Comparison with LOTOS

We consider behaviour patterns of Figure 3.12, and compare these patterns with their possible corresponding LOTOS behaviour expressions, considering the differences of design model already pointed out in section 3.2.7.

The behaviour pattern of Figure 3.12 (a) corresponds in LOTOS to the behaviour expression `B := a0; a1; a2; stop`. The behaviour pattern of Figure 3.12 (b) corresponds in LOTOS to the behaviour expression `B := a0; (a1; stop [] a2; stop)`.

Causality relations allow the explicit representation of arbitrary interleaving, by defining explicitly which actions are not supposed to happen at the same time. Considering the parallel semantics induced by causality relations, thus, the behaviour pattern of Figure 3.12 (c) corresponds in LOTOS to the behaviour expression `B := a0; (a1; stop ||| a2; stop)`.

The behaviour pattern of Figure 3.12 (d) corresponds in LOTOS to the behaviour expression `B := a0; (a2 ; stop [> a1; stop)`, which is equivalent to `a0; (a2 ; a1 ; stop [] a1; stop)`. The condition $\neg\, a_1$ -> a_2 is also comparable to the asymmetric conflict relation introduced in [3] for extended bundle event structures. The introduction of asymmetric conflict in [3] followed, amongst others, from the difficulties in defining the formal semantics of the LOTOS disable operator in a bundle event structure formalism.

This behaviour pattern of Figure 3.12 (e) cannot be formally represented in LOTOS, since LOTOS has an interleaving semantics. A common artifice is the use of arbitrary interleaving to represent independence (in the example `B := a0; (a1; stop ||| a2; stop)`). This cannot be avoided, since in LOTOS one cannot distinguish between arbitrary interleaving and independence at the semantics level. The major drawback of such an artifice is that it may not be clear for an implementer whether a_1 and a_2 are arbitrarily interleaved or independent. Sometimes it is necessary to have this distinction explicitly made, which requires that (normally informal) statements have to be added to the specification.

Modality

Another interesting aspect of our design model is the possibility of defining probability and modality. Consider the following causality relation:

$$a_1\ (t_1\text{: Time}) \text{ -> } a_2\ (t_2\text{: Time}, p_2\text{: Prob})\ [t_2 < t_1 + 10, p_2 < 100\%]$$

Assuming that a_1 is enabled by a start and that no other actions than a_1 and a_2 are defined, this behaviour can be specified in LOTOS in the following way:

```
a1 ; (i (* 1 *) ; a2 ; stop [] i (* 2 *) ; stop )
```

The internal events represent non-determinism in this behaviour. In case internal event marked with (* 1 *) happens, `a2` becomes possible, if the environment wants to perform it. The internal event marked with (* 2 *) models that the system has internally chosen not to make the occurrence of `a2` possible. The introduction of internal events is the only way this behaviour can be defined in LOTOS, since in LOTOS events that are enabled by all participating entities eventually happen. However, internal events obscure the relationship between actions a_1 and a_2, by relating the possibility that a_2 does not happen, i.e. the *non-occurrence* of a_2, to the *occurrence* of an internal event.

3.5 References

[1] L. Ferreira Pires, C. A. Vissers, and M. van Sinderen. Advances in architectural concepts to support distributed systems design. In *11o Simpósio Brasileiro de Redes de Computadores. Tutoriais e Minicursos*, Campinas, Brazil, 1993. Departamento de Engenharia de Computação. Faculdade de Engenharia Elétrica. Universidade Estadual de Campinas. also available as: Memorandum Informatica 93-17 (TIOS 93-09), University of Twente, Enschede, the Netherlands.

[2] B. Hoogeboom and W. A. Halang. The concept of time in the specification of real-time systems. In M. Schiebe and S. Pferrer, editors, *Real-time systems engineering and applications*, The Kluwer International Series in Engineering and Computer Science, chapter 2, pages 11–40. Kluwer Academic Publishers, USA, 1992.

[3] R. Langerak. *Transformations and Semantics for LOTOS*. PhD thesis, University of Twente, Enschede, Netherlands, 1992.

[4] J. Schot. *The role of Architectural Semantics in the formal approach towards Distributed Systems Design*. PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[5] C. A. Vissers, L. Ferreira Pires, and D. A. Quartel. *The Design of Telematic Systems*. University of Twente, Enschede, the Netherlands, Nov. 1993. Lecture Notes.

[6] C. A. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma. Specification styles in distributed systems design and verification. *Theoretical Computer Science*, 89:179–206, 1991.

Chapter 4

Causality-oriented Behaviour Composition

This chapter introduces the *causality-oriented behaviour composition* structuring technique. This structuring technique is introduced in order to allow complex behaviours to be structured, making them understandable for designers, and to allow behaviours to be defined in terms of compositions of general purpose (sub-)behaviours that can be re-used in the definition of various different behaviours.

The chapter is structured as follows: section 4.1 introduces entries and exits, which are notational mechanisms to compose behaviours, section 4.2 generalizes the use of these mechanisms by allowing behaviours to be instantiated inside other behaviours, section 4.3 discusses the representation of some well-known behaviour patterns using causality-oriented behaviour composition, and section 4.4 compares causality-oriented behaviour composition with the LOTOS sequential composition.

4.1 Entries and Exits

Design models for behaviour definition should allow designers to structure not only actions and their relationships, but also behaviours and their relationships. Causality relations involving actions, which have been discussed so far, correspond to a microscopic view on behaviour definitions. Causality relations in terms of relationships involving behaviours shall enhance the expressive power of our design model to a macroscopic view on behaviour definitions, allowing designers to define complex behaviours as compositions of more simple ones.

Causality relations between actions can be generalized, by also allowing the definition of causality and exclusion between behaviours. By using these more general causality relations, behaviours of functional entities can be structured in terms of relationships between actions or relationships between behaviours, or as a combination of both, such that behaviour hierarchies can be defined. Furthermore behaviours can be defined in terms of pre-defined sub-behaviours, which also allows some degree of reusability through instantiation. Repetitive and infinite behaviour can also be expressed using this structuring technique, through recursive behaviour instantiation.

In order to generalize causality relations to allow causality and exclusion between behaviours, we have to find a proper matching between elements of a causality relation and elements of behaviour definitions. A causality relation contains conditions and a result action possibly with constrained attribute values. Analogously, causality-oriented behaviour composition can be used to express that conditions inside a certain behaviour enable and constrain result actions of other behaviours.

Behaviour entries and exits have been defined to support causality-oriented behaviour composition. An *entry* defines a point from which actions in a behaviour can be enabled and an *exit* defines conditions that can be used to enable actions of other behaviours. This allows one to connect behaviours to each other by connecting behaviour entries and behaviour exits.

4.1.1 Single Entry and Exit

Suppose B_1 and B_2 are behaviours defined in terms of causality relations, and that B_1 has one exit and B_2 has one entry. A sequential composition between B_1 and B_2 can be defined by combining the conditions of the exit of B_1 and the action(s) of the entry of B_2, such that the conditions of the exit of B_1 become conditions to the action(s) of the entry of B_2.

The following example illustrates causality-oriented behaviour composition using a single entry and a single exit:

$$\begin{aligned} B_1 := \{ & start \text{ -> } a_1, start \text{ -> } a_2, \\ & a_1 \wedge a_2 \text{ -> } a_3, \\ & a_3 \text{ -> } a_4, a_3 \text{ -> } a_5, \\ & a_4 \wedge a_5 \text{ -> } exit \} \end{aligned}$$

$$\begin{aligned} B_2 := \{ & entry \text{ -> } a_6, entry \text{ -> } a_7, \\ & a_6 \wedge a_7 \text{ -> } a_8 \} \end{aligned}$$

The statements *entry* -> a_6, *entry* -> a_7 allow a_6 and a_7 to be enabled by coupling a condition to this *entry*. The statement $a_4 \wedge a_5$ -> *exit* allows to make the occurrence of both a_4 and a_5 a condition for the occurrence of other actions. A causality-oriented behaviour composition of B_1 and B_2 can be defined in the following way:

$$B := \{ B_1\,(exit) \text{ -> } B_2\,(entry) \}$$

When the *exit* condition of B_1 becomes true, the actions that follow the *entry* of B_2 are enabled. This means that B_1 is coupled to B_2, such that the conditions of the *exit* of B_1 become the conditions for the actions of the *entry* of B_2. This composition corresponds to the following monolithic behaviour:

$$\begin{aligned} B := \{ & start \text{ -> } a_1, start \text{ -> } a_2, \\ & a_1 \wedge a_2 \text{ -> } a_3, \\ & a_3 \text{ -> } a_4, a_3 \text{ -> } a_5, \\ & a_4 \wedge a_5 \text{ -> } a_6, a_4 \wedge a_5 \text{ -> } a_7, \\ & a_6 \wedge a_7 \text{ -> } a_8 \} \end{aligned}$$

Figure 4.1 depicts the sequential composition of B_1 and B_2.

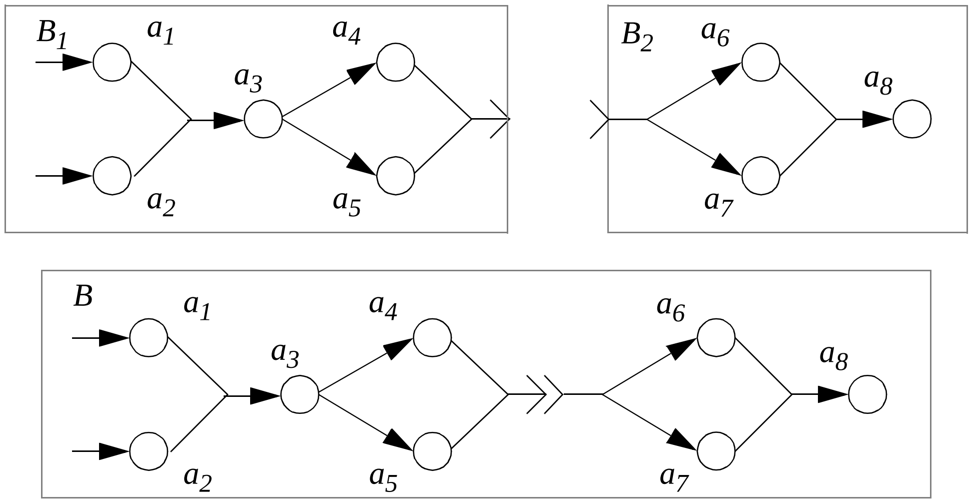


Figure 4.1: Causality-oriented behaviour composition: one entry and one exit

Figure 4.1 shows that we have actually connected the conditions of the *exit* of B_1 to the entry of B_2. This turns the conditions of the *exit* of B_1 into conditions of a_6 and a_7.

4.1.2 Negative Exit Conditions

The basic conditions in causality relations involving actions are causality and exclusion. In order to have similar basic conditions in causality relations involving behaviours we define not only (positive) exit conditions, such as the ones defined so far, but also negative exit conditions.

Negative exit conditions can also be used as conditions to enable actions in other behaviours. However, due to the implicit time constraints of causality and exclusion we cannot simply consider an exclusion as the negation of causality. For example if a_1 is a condition for a_2, a_1 happens before a_2, and the negation of this condition is naturally a_1 does not happen before a_2. In case $\neg a_1$ is a condition for a_2, a_1 should not happen before or at the same time as a_2, which is a different condition than the negation of a_1.

In order to avoid problems with implicit time constraints, we introduce a negation operator – on conditions, which converts causality to exclusion and vice-versa. The negation operator – has the following interpretations when applied to conditions that do not contain explicit references to attribute values:

- $-(C_1 \wedge C_2) = -C_1 \vee -C_2$ for conditions C_1 and C_2
- $-(C_1 \vee C_2) = -C_1 \wedge -C_2$ for conditions C_1 and C_2
- $-a_i = \neg a_i$ for action a_i
- $-(\neg a_i) = a_i$ for action a_i

The negation operator on conditions is not so trivial as it seems at first sight. For generic conditions it works as an ordinary boolean negation, but for terminals (a_i and $\neg a_i$) it transforms causality into exclusion, and vice-versa.

Negative exit conditions can be defined in causality relations by using the negation operator. Consider the following example:

B_1 := { *start* -> a_1, *start* -> a_2, $a_1 \vee a_2$ -> *exit* }
B_2 := { *entry* -> a_3 }

Behaviour *B*:= { *start* $\wedge$ – B_1 *(exit)* -> B_2 *(entry)* } defines that *start* $\wedge$ -($a_1 \vee a_2$) is the condition for the entry of B_2. Applying the interpretation rules above, we conclude that a_1 and a_2 exclude a_3, which makes this behaviour correspond to the following monolithic behaviour definition:

B' := { *start* -> a_1, *start* -> a_2,
start $\wedge \neg a_1 \wedge \neg a_2$ -> a_3 }

The negation operator is not essential for defining behaviour. It can be seen as a notational artifact for defining behaviours as compositions of sub-behaviours with structuring capabilities similar to the capabilities for defining behaviours as compositions of actions. For example, behaviour *B* could be defined using aa (positive) exit condition in the following way:

B:= {B_1 *(exit)* -> B_2 *(entry)*,
where
B_1 := { *start* -> a_1, *start* -> a_2, *start* $\wedge$ ($\neg a_1 \vee \neg a_2$) -> *exit* }
B_2 := { *entry* -> a_3 } }

The negation operator must be carefully used. Behaviours that could be considered as equivalent may generate non-equivalent behaviours when they are combined with other behaviours through negative exit conditions. An example is a behaviour containing a condition $a_1 \vee (\neg a_2 \wedge a_2)$. This condition could be considered as equivalent to the condition a_1, because it is impossible that a_2 happens and does not happen at the same time. However the application of the negation operator on $a_1 \vee (\neg a_2 \wedge a_2)$ results in $\neg a_1 \wedge (a_2 \vee \neg a_2)$, which corresponds to exclusion by a_1 and interleaving with a_2, while the application of the negation operator on a_1 results in $\neg a_1$. We have stated in Chapter 3 that designers should never define conditions which are impossible. In this example, the apparently harmless condition $\neg a_2 \wedge a_2$ brings serious consequences when the negation operator is used. Since we are only interested in the discussing this problem from a designer's point of view, we simply consider the condition $a_1 \vee (\neg a_2 \wedge a_2)$ as irregular. Tools to detect such conditions and eliminate them could be developed. Further work on a formal semantics for causality relations should try to solve this problem.

4.1.3 Multiple Entries and Exits

Entries and exits can be generalized, such that a behaviour may have more than one entry, more than one exit, or both. Proper compositions of behaviours with multiple entries and exits, however, are only possible if we are able to distinguish these multiple entries and exits, which implies that each of these entries and exits must be uniquely identified.

Consider the following example:

$$\begin{aligned}B_1 := \{ & start \text{ -> } a_1, start \text{ -> } a_2, start \text{ -> } a_3,\\ & a_1 \wedge a_2 \text{ -> } exit_1\\ & a_3 \text{ -> } exit_2 \}\end{aligned}$$

$$\begin{aligned}B_2 := \{ & entry_1 \text{ -> } a_4,\\ & entry_2 \text{ -> } a_5,\\ & a_4 \wedge a_5 \text{ -> } a_6 \}\end{aligned}$$

Behaviour *B:= { B_1 (exit$_1$) -> B_2 (entry$_1$), B_1* (*exit$_2$) -> B_2 (entry$_2$) }* implies that *exit$_1$* of *B_1* is connected to *entry$_1$* of *B_2*, and that *exit$_2$* of *B_1* is connected to *entry$_2$* of *B_2*. We assume that *B_1 (exit$_1$)* and *B_1 (exit$_2$)*, and *B_2 (entry$_1$*) and *B_2 (entry$_2$)* refer to the same instances of *B_1* and *B_2*, respectively.

Figure 4.2 depicts behaviour *B* above in terms of its sub-behaviours *B_1* and *B_2*.

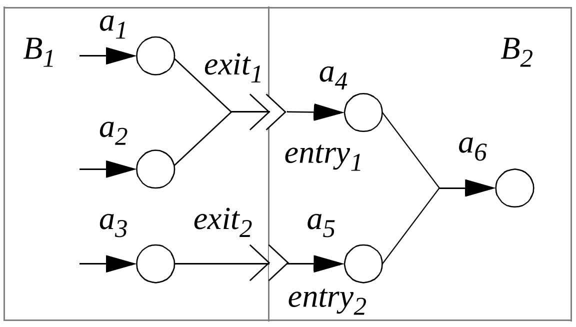


Figure 4.2: Behaviours with multiple exits and entries

Summarizing, entries and exit constructs can be defined and used in the following ways:

1. an *exit* can be declared in the result of a causality relation. For each *exit* we can assign a condition *C(B(exit)*), which is the condition of the causality relation in which this *exit* is declared;

2. an *exit* can be used to define causality-oriented behaviour composition, to indicate that *C(B(exit)*) is a condition for actions of entries of other behaviours. This can be done by explicitly referring to this *exit* in the condition of a causality relation (such as in *B(exit)*);

3. an *exit* can be used to define causality-oriented behaviour composition, to indicate that the negation of *C(B(exit)*) is a condition for actions of entries of other behaviours. This can be done by explicitly referring to the negation of this *exit* condition of a causality relation (such as in - *B(exit)*);

4. an *entry* can be declared in the condition of one or more causality relations. For each *entry* we assign a set of actions, namely the actions which are result in the causality relations where this *entry* is declared. We call this set *A(B(entry))*;

5. an *entry* can be used to define causality-oriented behaviour composition, to indicate that for each action of *A(B(entry)*) the conditions of a certain *exit* apply. This is done by referring to the *entry* in the result of a causality relation (such as in *B(entry)*);

In Figure 4.1, for example, we can infer that $A(B_2(entry)) = \{a_6, a_7\}$ and $C(B_1(exit)) = a_4 \wedge a_5$. Behaviour $B := \{ B_1 (exit) \text{ -> } B_2 (entry) \}$ states that for each action $a_i \in A(B_2(entry))$, there is a causality relation of type $C(B_1(exit)) \text{ -> } a_i$.

The definition of entries and exits in a behaviour corresponds to the definition of the interfaces of this behaviour with respect to causality-oriented behaviour composition, since it defines the possible ways this behaviour can be combined with other behaviours using this structuring technique.

4.1.4 Attributes in Entries and Exits

Entry and exit constructs can be used to pass information from the *exit*ing behaviour to the resulting behaviour. Considering two behaviours B_1 and B_2, we can use entries and exits to relate attributes of actions of exits of B_1 to the attributes of actions of entries of B_2. A requirement for the coupling of two behaviours using entries and exits is that there must be a matching between the values defined in the exits and the values expected by the entries to which these exists are connected.

Consider the following example of single exit and single entry:

$B_1 := \{\ldots\ a_4 \wedge a_5 \text{ -> } exit\ (v_1, v_2)\ \}$
where v_1 and v_2 are values of sort s_1 and s_2 respectively

$B_2 := \{ entry\ (a: s_1, b: s_2) \text{ -> } a_6, entry\ (a: s_1, b: s_2) \text{ -> } a_7, \ldots \}$

In this example, behaviour $B := \{ B_1 (exit) \text{ -> } B_2 (entry) \}$ can be defined, since there is a matching between the list of parameters in the *exit* of B_1 and the list of parameters in the *entry* of B_2. This means that B_2 is expecting the values that B_1 has to offer.

Timing information can also be passed to resulting behaviour through entries and exits. Consider the following example:

$B_1 := \{\ldots\ a_4\ (t_4\!: Time) \wedge a_5\ (t_5\!: Time) \text{ -> } exit\ (t_4, t_5)\ \}$
$B_2 := \{ entry\ (t_a\!: Time, t_b\!: Time) \text{ -> }$
$\qquad a_6\ (t_6\!: Time)\ [\ t_6 > max\ (t_a +10, t_b +10)]$
$\quad \ldots \}$

The statement $B_1 (exit) \text{ -> } B_2 (entry)$ allows timing information from B_1 to be made available in B_2. Timing information is treated in this example as data values of sort *Time*. These values are used to define constraints on the time attribute value of a_6.

Negation Operator

The negation operator has to be reconsidered for conditions which contain explicit references to attribute values. In Chapter 3 we have stated that exclusion conditions should not contain references to attribute values. The consequence was that the negation of a causality condition such as, for example, $a_1\ (v_1\!: Nat)\ [v_1 < 10]$ is $a_1\ (v_1\!: Nat)\ [v_1 \geq 10] \vee \neg\, a_1$, which means that either a_1

does not happen (before or at the same time), or a_1 with values that do not agree with the condition.

The negation operator on causality with explicit action value attribute conditions should be defined as a generalization of the example above, which results in the following interpretation rule, for a condition action a_i that establishes value v_i of sort S_i with value condition $c_i\ (v_i)$:

$$-\ (\ a_i\ (v_i\!: S_i)\ [c_i\ (v_i)])\ =\ a_i\ (v_i\!: S_i)\ [\neg c_i\ (v_i)] \vee \neg\ a_i$$

In this case the negation operator applied to an action condition with conditions on the established values results in (i) either the occurrence of the condition action with the negation of the original condition, or (ii) the non-occurrence of the condition action.

4.2 More General Behaviour Definitions

So far behaviours have been either described in terms of causality relations involving actions, or in terms of causality relations involving entries and exits of behaviours. This section combines these two forms of behaviour definition, allowing behaviour to be defined in terms of causality relations involving actions, entries and exits.

4.2.1 Actions as Conditions for Entries

Conditions for a behaviour entry can be defined inside another behaviour, which allows one to directly create instances of behaviours.

Consider the following example:

```
B := { start -> $a_1$,
       $a_1$ -> $a_2$,
       $a_1$ -> $B_1$ (entry) }
where
       $B_1$ := {entry -> $a_3$,
                 entry -> $a_4$ } }
```

In this example all actions of $A(B_1\ (entry))$ (a_3 and a_4) are enabled by the occurrence of a_1. The occurrence of a_1 becomes a condition for the actions of $A(B_1\ (entry))$. Since a_2 is independent of a_3 and a_4, it is allowed to happen even if B_1 starts, which is even if a_3 or a_4 take place. This means that this behaviour structuring technique allows a behaviour to enable another behaviour and yet continue its operation in principle indefinitely.

Figure 4.3 shows behaviour *B* defined above.

Behaviour instantiation can be used to form a hierarchy of behaviours, in which behaviours are defined as a composition of sub-behaviours and so on. Such constructs are useful to represent repetitive, possibly infinite, behaviour.

Consider the following behaviour definition:

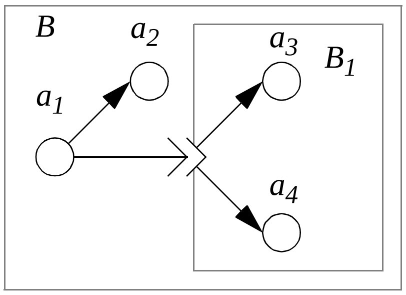
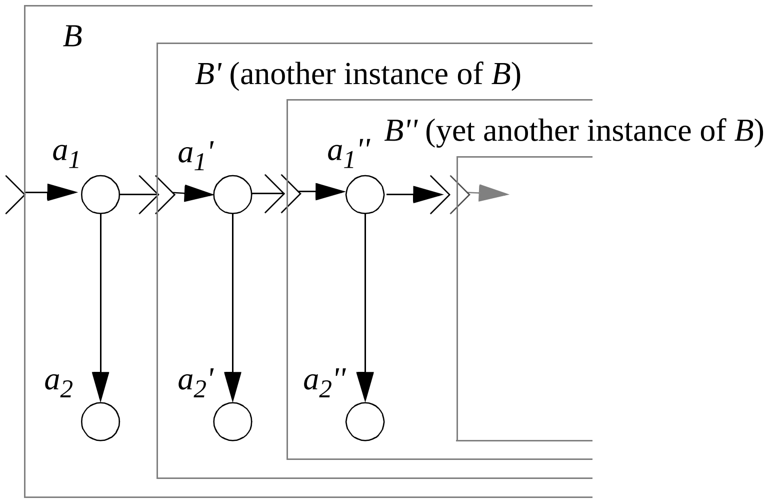


Figure 4.3: Behaviour instantiation

$B := \{\ entry \text{ -> } a_1,$
$\qquad a_1 \text{ -> } a_2,$
$\qquad a_1 \text{ -> } B\ (entry)\ \}$

In this behaviour definition a_1 is not only used to enable a_2, but it also enables another instance of B. This means that the behaviour defined in B is an infinite succession of a_1 followed by a_2, where each new instance of a_1 (a_1', a_1'', etc.) is caused by its predecessor, but the different instances of a_2 (a_2', a_2'', etc.) are independent of each other.

Figure 4.4 depicts behaviour B defined above.



B
B' (another instance of B)
B'' (yet another instance of B)
a_1
a_1'
a_1''
a_2
a_2'
a_2''

Figure 4.4: Repetitive behaviour

In Chapter 3 we have assumed that each action of a behaviour is unique and is uniquely identified. In the case of repetitive and infinite behaviours it is not feasible to define an infinite number of actions, such that one has to consider the interpretation (semantics) of the causality relations as the occasion in which new action identifiers are generated. This has been shown in Figure 4.4, where new identifiers a_i', a_i'', etc. are generated for each of the instances of action a_i.

4.2.2 Conditional Behaviour Instantiation

Behaviour instantiations can be made dependent of specific conditions involving action attribute values. Timing, action values, retained values and probability conditions can be defined in causality relations involving actions and behaviour entries and exits, similarly to the causality relations involving actions.

Consider the following behaviour definition:

$$
\begin{aligned}
&B := \{\ start \text{ -> } a_1\ (v_1\text{: } Nat),\\
&\qquad a_1\ (v_1\text{: } Nat) \text{ -> } a_2,\\
&\qquad a_1\ (v_1\text{: } Nat)\ [v_1 < 10] \text{ -> } B_1\ (entry\ (v_1))\ \}\\
&where\\
&\qquad B_1 := \{\ entry\ (v_1\text{: } Nat) \text{ -> } a_3\ (v_3\text{: } Nat)\ [v_3 = 2*v_1],\\
&\qquad\qquad entry\ (v_1\text{: } Nat) \text{ -> } a_4\ (v_4\text{: } Nat)\ [v_4 < v_1]\ \}\ \}
\end{aligned}
$$

In this behaviour definition the occurrence of a_1 with $v_1 < 10$ is a necessary and sufficient condition for the instantiation of B_1. Actions a_3 and a_4 are only allowed to occur if a_1 occurs with $v_1 < 10$. This example also shows that the entry of B_1 can be used to make reference to values established in the behaviour where B_1 is instantiated. In this case v_3 should be equal to $2*v_1$ and v_4 should be smaller than v_1.

4.2.3 Actions Enabled by Behaviour Exits

Exit conditions of a behaviour and their negation can be directly used as conditions in causality relations of actions in other behaviours. Consider the following example:

$$
\begin{aligned}
&B := \{\ start \text{ -> } B_1\ (entry),\\
&\qquad B_1\ (exit) \text{ -> } a_3\\
&where\\
&\qquad B_1 := \{\ entry \text{ -> } a_1,\\
&\qquad\qquad entry \text{ -> } a_2,\\
&\qquad\qquad a_1 \wedge a_2 \text{ -> } exit\ \}\ \}
\end{aligned}
$$

In this example the condition of the exit of B_1, namely $a_1 \wedge a_2$, becomes a condition for action a_3. B_1 is also instantiated in B, by the statement *start -> B_1 (entry).*

Figure 4.5 depicts this example.

An *exit* can be used to allow the explicit reference to action attribute values between different behaviours, such that value and time constraints on actions can be defined in terms of attribute values made accessible by behaviour exits.

Consider the following behaviour definition:

$$
\begin{aligned}
&B := \{\ start \text{ -> } B_1,\\
&\qquad B_1\ (exit\ (v_x, t_x)) \text{ -> } a_3\ (v_3\text{: } Nat, t_3\text{: } Time)\ [v_x < v_3 < 20, t_3 < t_x + 10]\\
&where
\end{aligned}
$$

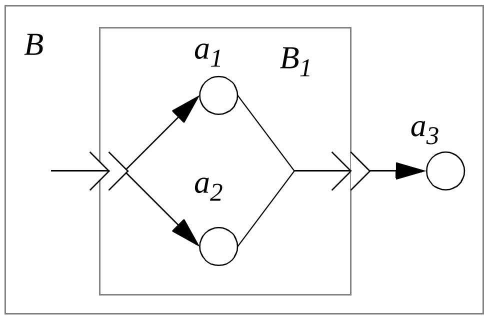


Figure 4.5: Exit as condition for an action

B_1 := { *entry* -> a_1 (t_1: *Time*),
entry -> a_2 (v_2: *Nat*, t_2: *Time*),
a_1 (t_1: *Time*) $\wedge$ a_2 (v_2: *Nat*, t_2: *Time*) [$v_2 > 10$]
-> *exit* (v_2, *max* (t_2, t_1)) } }

In this behaviour the exit of B_1 becomes a condition for a_3, and attribute values of a_1 and a_2 are used in the constraints of a_3. An equivalent causality relation for a_3 in monolithic form would be:

a_1 (t_1: *Time*)$\wedge$ a_2 (v_2: *Nat*, t_2: *Time*) [$v_2 > 10$] ->
a_3 (v_3: *Nat*, t_3: *Time*) [$v_2 < v_3 < 20$, $t_3 <$ *max* (t_2, t_1) +10]

4.2.4 Different Behaviour Instances

Causality-oriented behaviour composition allows one to define multiple entries in a behaviour. The consequence is that one should be able to determine, each time a certain entry is mentioned, whether this entry belongs to an existing or to a new behaviour instance.

Consider the following behaviour:

B := { *start* -> a_1, a_1 -> a_2, a_2 -> a_3, a_3 -> a_4,
a_1 -> B_1 (*entry*$_1$),
a_2 -> B_1 (*entry*$_2$),
a_3 -> B_1 (*entry*$_1$),
a_4 -> B_1 (*entry*$_2$) }
where
B_1 := { *entry*$_1$ -> a_5,
entry$_2$ -> a_6, ... } }

In this behaviour actions a_1, a_2, a_3 and a_4 altogether instantiate behaviour B_1 twice. Since each action or entry should have only one causality relation, the repetitions of the causality relations of B_1 (*entry*$_1$) and B_1 (*entry*$_2$) should be interpreted as the creation of a new instance of B_1. In order to simplify our notation, we make use of the order in which the causality relations involving entries are defined in order to associate conditions to behaviour instances. We assume that the first reference to an entry of a behaviour as a result in a causality relation instantiates this behaviour. Subsequent references to entries of this behaviour as a result in a causality relation refer to the existing behaviour instance. When an entry which has been already used as a result in a causality relation is used again, we suppose that a new instance of behaviour is created. References to entries are assigned to behaviour instances in the order that the instances have been created.

A more complex notation for distinguishing different instances of a certain behaviour may be necessary in case more complex behaviour structures are defined, but it falls outside the scope of this work.

Figure 4.6 depicts the graphical interpretation of the behaviour defined above.

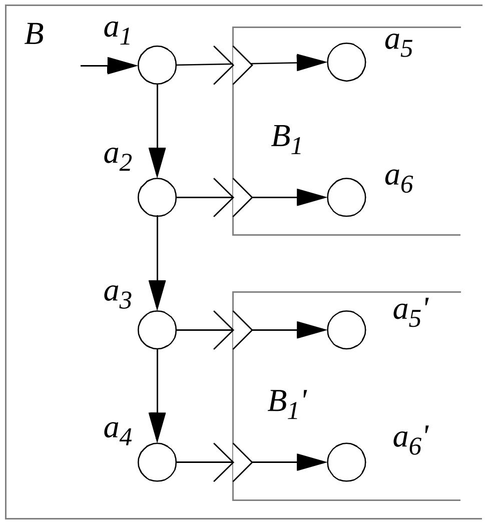


Figure 4.6:Two instances of a behaviour

4.2.5 Graphical Interpretation

Figure 4.7 illustrates the effect of the causality-oriented behaviour composition on a generic behaviour composed of four arbitrary sub-behaviours B_1, B_2, B_3 and B_4.

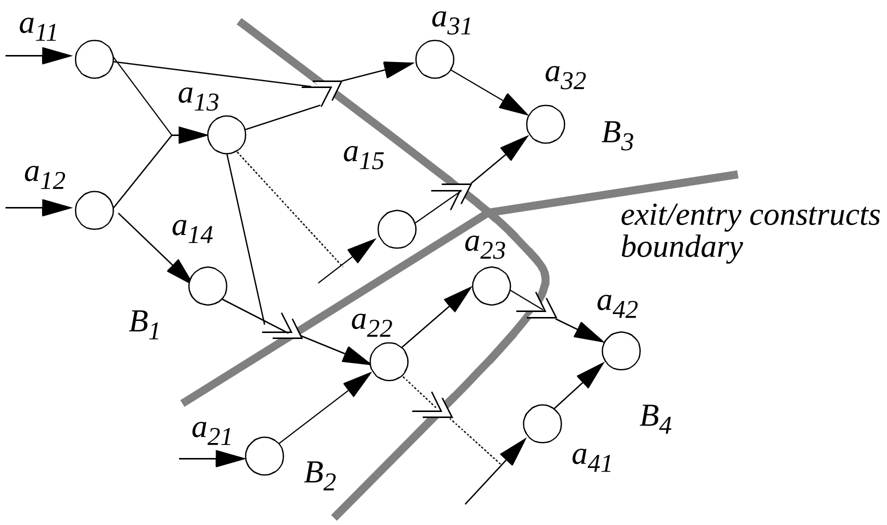


Figure 4.7: Causality-oriented behaviour composition for arbitrary behaviours

In Figure 4.7, entries and exits define a boundary that delimits the behaviours B_1, B_2, B_3 and B_4, by decomposing the causality relations of the monolithic behaviour. This mechanism allows designers to structure a monolithic behaviour in terms of sub-behaviours in a rather flexible way.

4.3 Expressive Power

The expressive power of the causality-oriented behaviour composition can be evaluated by investigating the representation of some well-known behaviour patterns using this structuring technique. This section addresses some of these behaviour patterns, by considering that behaviour definitions already contain their interfaces in terms of their exits and entries. This implies that a behaviour should be defined such that it can be combined with other behaviours in the desired ways.

Behaviour patterns are represented in this section in a similar form as their corresponding behaviour patterns involving actions discussed in section 3.4.2.

4.3.1 Sequential Ordering

Sequential ordering is a behaviour pattern in which the completion of one behaviour enables another behaviour. The condition that determines the completion of a behaviour must have been associated to an *exit*, while the initial actions of the other behaviour must have been associated with an *entry* in this case.

Consider the following example:

```
B := { start -> $B_1$ (entry)
       $B_1$ (exit) -> $B_2$ (entry)
       where
       $B_1$:= { entry -> $a_1$, entry -> $a_2$,
                 $a_1 \wedge a_2$ -> exit }
       $B_2$:= { entry -> $a_3$ } }
```

In this example the condition that determines the completion of B_1, in this case the execution of both actions a_1 and a_2, enable the actions of the entry of B_2, in this case the action a_3.

Figure 4.8 depicts behaviour B in terms of its sub-behaviours.

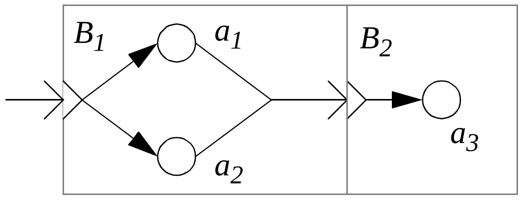


Figure 4.8: Example of sequential composition

4.3.2 Choice

Considering two behaviours B_1 and B_2, a choice between these behaviours means that in case an action of B_1 happens, no action of B_2 is allowed any more, and vice-versa. In the most general form any action of B_1 excludes any action B_2 and vice-versa. However, it is sufficient to have initial actions of B_1 and B_2 excluding each other.

In order to define a choice between two behaviours B_1 and B_2, we have to define these behaviours such that the conditions indicating that any initial action has occurred in B_1 and B_2 are assigned to exits of these behaviours.

Consider the following behaviour definition:

```
B := { start ∧ – B₂ (exit) -> B₁ (entry),
       start ∧ – B₁ (exit) -> B₂ (entry)
       where
       B₁ := { entry ∧ ¬ a₂ -> a₁, entry ∧ ¬ a₁ -> a₂,
               a₁ ∧ a₂ -> a₃, a₁ ∨ a₂ -> exit }
       B₂ := { entry -> a₄, entry -> a₅,
               a₄ ∧ ¬ a₅ -> a₆, a₄ ∨ a₅ -> exit } }
```

In this example, the condition that any initial action of B_1 occurs is defined as $a_1 \vee a_2$, which means that a_1 or a_2 occur, and the condition that any initial action of B_2 is defined as $a_4 \vee a_5$, which means that a_4 or a_5 occur. These conditions are assigned to B_1 *(exit)* and B_2 *(exit)*, respectively. The negation of B_1 *(exit)* and the negation of B_2 *(exit)* are conditions for the occurrence of the actions of the entry of B_2 and of the entry of B_1, respectively, which defines a choice between behaviours B_1 and B_2.

Figure 4.9 depicts behaviour B in terms of its sub-behaviours.

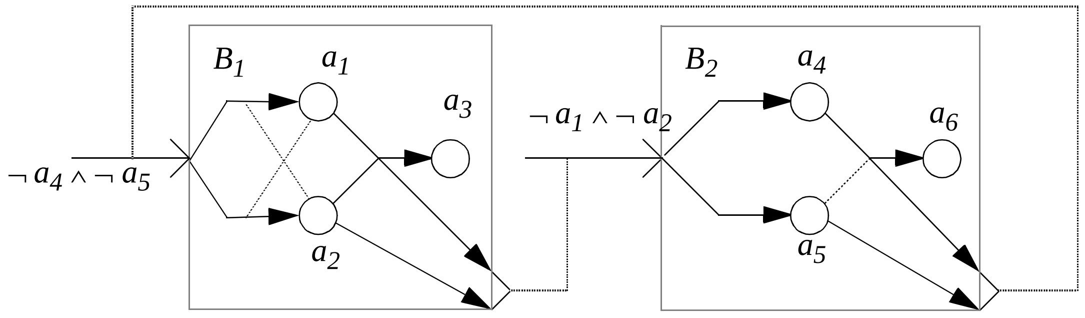


Figure 4.9: Example of behaviour choice

The interpretation of *start* $\wedge$ – B_2*(exit)* -> B_1*(entry)* is that a condition for the actions of B_1*(entry)* to occur is that the condition $C(B_2(exit))$ is not true, according to the rules of the negation operator. The same applies to *start* $\wedge$ – B_1*(exit)* -> B_2*(entry)*. In this example we have defined the occurrence of any initial actions of B_1 and B_2, in B_1 *(exit)* and B_2 *(exit)* respectively, in order to be able to represent the choice between B_1 and B_2.

Figure 4.10 depicts the monolithic behaviour that corresponds to behaviour B above.

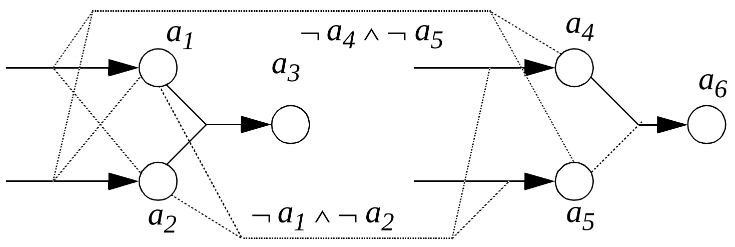


Figure 4.10: Monolithic representation of the choice example

4.3.3 Disabling

Disabling is a behaviour pattern in which actions of a certain behaviour B_1 are not allowed to be executed any more in case any action of another behaviour B_2 occurs. In the most general form any action of B_2 excludes all actions of B_1. However, it is sufficient to have initial actions of B_2 excluding all actions of B_1.

In order to define that B_2 disables B_1, we have to define B_2 such that a condition indicating that any of its initial action has occurred is assigned to an *exit*. The negation of this *exit* should be a condition for all actions of B_1.

Consider the following behaviour definition:

B := { start -> B_2 (entry),
start $\wedge$ – B_2 (exit) -> B_1 (entry),
where
B_1 := { entry -> a_1, entry $\wedge$ a_1 -> a_2,
entry $\wedge$ a_2 -> a_3 }
B_2 := { entry -> a_4, entry -> a_5,
$a_4 \wedge \neg a_5$ -> a_6, $a_4 \vee a_5$ -> exit } }

The occurrence of any initial action of B_2 is defined as $a_4 \vee a_5$, and it is assigned to the exit of B_2 An *entry* is defined in the causality relation of all actions of B_1, which is necessary because all actions of B_1 should be disabled if any initial action of B_2 happens.

Figure 4.11 depicts this example.

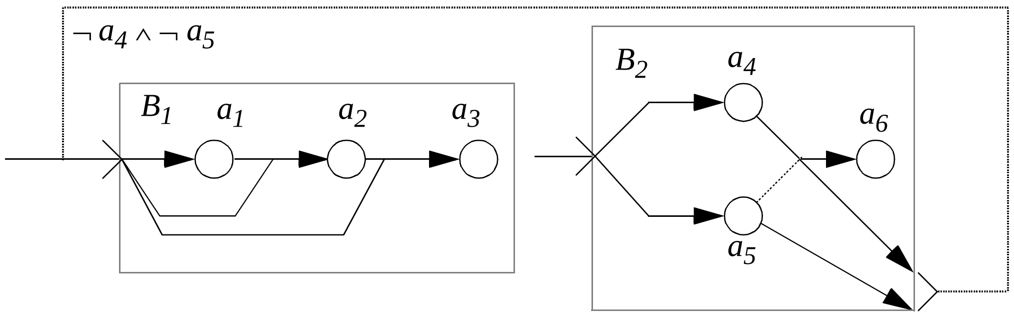


Figure 4.11: Example of disabling

The statement *start* $\wedge$ - *B_2 (exit) -> B_1 (entry)* defines that a condition for any action of *$A(B_1$ (entry))* to occur is that the condition of *B_2 (exit)* is not true, according to the rules of the negation operator. In case the condition of *B_2 (exit)* becomes true (a_4 or a_5 happen) no action of the entry of B_1 is allowed to happen. Actions of *B_2 (entry)* are allowed to happen at any time, which is stated in *start -> B_2 (entry)*.

Figure 4.12 depicts the monolithic behaviour that corresponds to the example above.

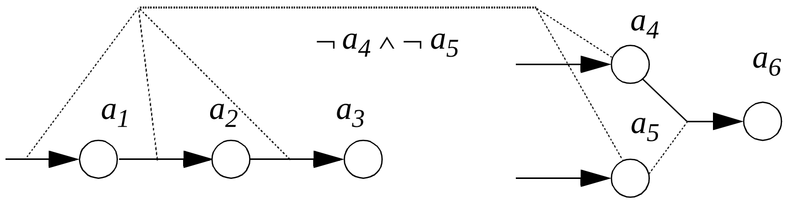


Figure 4.12: Monolithic representation of the disabling example

4.3.4 Independence

Behaviours are said to be independent if actions of one behaviour are allowed to happen independently of the actions of the other behaviour.

Consider the following behaviour definition:

$$
\begin{array}{l}
B := \{\ start \rightarrow B_1\ (entry), \\
\qquad start \rightarrow B_2\ (entry) \\
\qquad where \\
\qquad B_1 := \{\ entry \rightarrow a_1, \\
\qquad\qquad a_1 \rightarrow a_2,\ \ a_2 \rightarrow a_3\ \} \\
\qquad B_2 := \{\ entry \rightarrow a_4,\ entry \rightarrow a_5, \\
\qquad\qquad a_4 \wedge \neg a_5 \rightarrow a_6\ \}\ \}
\end{array}
$$

In the behaviour definition above, the actions of B_1 are independent of the actions of B_2 such that these two behaviours can be performed in parallel.

Figure 4.13 shows two independent behaviours.

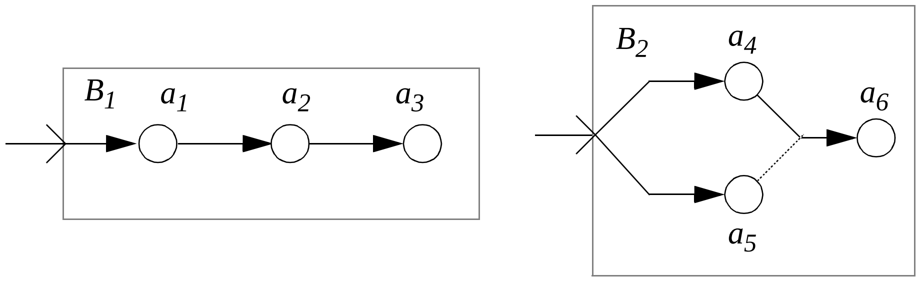


Figure 4.13: Example of independent behaviours

4.3.5 Arbitrary Interleaving

Arbitrary interleaving is a behaviour pattern in which actions are not allowed to occur at the same time. We limit this discussion to the interleaving between the initial actions of two behaviours B_1 and B_2, defined such that any initial action of B_1 is interleaved with any initial action of B_2 and vice-versa. In order to define interleaving in this way we must define B_1 and B_2 such that conditions that indicate that any initial actions of B_1 and B_2 have occurred are assigned to an exit in B_1 *and* B_2, respectively. These exits of B_1 and B_2 are used in the definition of an interleaving condition with the entry of B_2 and B_1, respectively.

Consider the following behaviour definition:

$$\begin{aligned}
B := \{ & start \wedge (-B_2\,(exit) \vee B_2\,(exit)) \text{ -> } B_1\,(entry),\\
& start \wedge (-B_1\,(exit) \vee B_1\,(exit)) \text{ -> } B_2\,(entry)\\
& where\\
& B_1 := \{ entry \text{ -> } a_1, entry \text{ -> } a_2,\\
& \qquad a_1 \wedge a_2 \text{ -> } a_3, a_1 \vee a_2 \text{ -> } exit \}\\
& B_2 := \{ entry \text{ -> } a_4, entry \text{ -> } a_5,\\
& \qquad a_4 \wedge \neg a_5 \text{ -> } a_6,\\
& \qquad a_4 \vee a_5 \text{ -> } exit \} \}
\end{aligned}$$

The condition that any initial action occurs is defined as $a_1 \vee a_2$ for B_1, and $a_4 \vee a_5$ for B_2. The statement *start* $\wedge$ *(- B_2 (exit)* $\vee$ *B_2 (exit)) -> B_1 (entry)* defines that a condition for actions of the entry of B_1 to happen is that the condition of the exit of B_2 is either true or not true, considering the rules of the negation operator. Since this condition is the occurrence of a_4 or a_5, this statement defines that either a_4, a_5 or none of these two actions, are conditions to the entry of B_1. This means that a_4 and a_5 are interleaved with all actions of the entry of B_1, in this case a_1 and a_2, such that a_4 and a_5 do not happen at the same time a_1 and a_2 happen.

Similarly to the case of interleaving between actions discussed in section 3.4.2, the statement *start* $\wedge$ *(– B_1 (exit)* $\vee$ *B_1 (exit)) -> B_2 (entry)* is not actually necessary, but it is used to emphasize the symmetry of interleaving.

Figure 4.14 depicts the behaviour definition above.

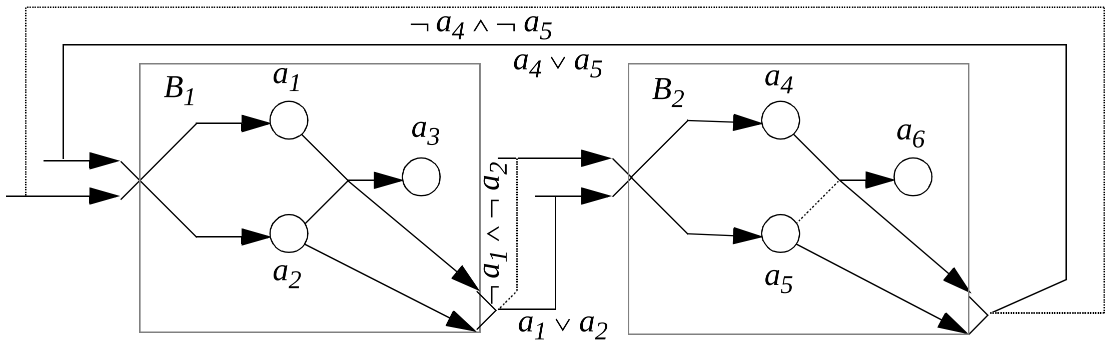


Figure 4.14: Example of interleaving

Figure 4.15 depicts the monolithic representation of this behaviour.

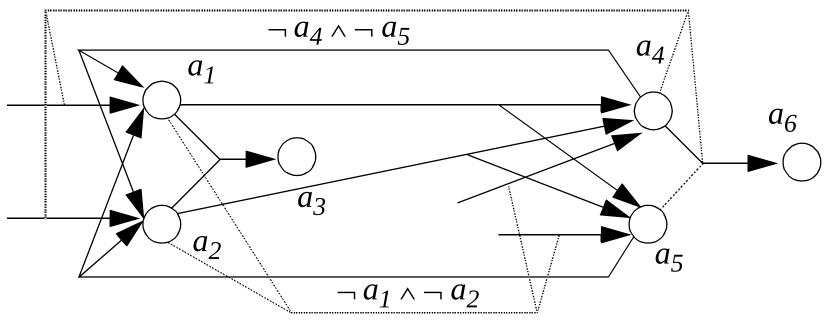


Figure 4.15: Monolithic representation of the interleaving example

4.3.6 Summary

The representation of these behaviour patterns using causality-oriented behaviour composition shows that sequential ordering is directly related to causality, disabling and choice are related to exclusion, interleaving is related to both causality and exclusion, and independence is related to the absence of causality relations.

In case of sequential composition and interleaving, explicit reference to action attributes is possible, while in the other behaviour patterns reference to attributes is not possible.

We also recognize that independence, arbitrary interleaving, disabling and choice impose increasingly severe constraints on the occurrence of actions of the entries of B_1 and B_2 (no restriction, not at the same time, not before or at the same time, never if the other happens). These restrictions have been again formulated as special cases of a single mechanism, analogously to the case of relations involving actions discussed in section 3.4.2.

Causality relations have been conceived to represent concurrence and causality, which implies that interleaving has to be explicitly represented whenever it is desired. In case all actions are expected to be interleaved independently of the behaviour they belong to, we have to represent this explicitly by relating each pair of actions in an interleaved way. The representation of interleaving using causality relation may result in a complex behaviour and can be a boring task.

Summarizing:

- *independence*: actions of B_1 and B_2 are completely unrelated (independent of each other) and can occur at any time;
- *arbitrary interleaving*: interleaved actions of B_1 and B_2 are not allowed to occur at the same time moment, but they can still occur;
- *disabling*: actions of B_1 are only allowed to occur as long as no initial action of B_2 occurs;
- *choice*: initial actions of B_1 and B_2 exclude each other, such that in case an initial action of B_1 happens, no initial action of B_2 can happen and vice-versa.

We conclude that some well-known behaviour patterns can be represented using causality relations. Some of these patterns, especially choice, disabling and interleaving, may generate complex relationships between actions. Notational artifacts to simply the definition of choice, disabling and interleaving between behaviours could be useful, but are outside the scope of this work.

4.4 Comparison with LOTOS

The causality-oriented behaviour composition is a generalization of the LOTOS concept of sequential composition. In LOTOS a behaviour can terminate successfully and enable another behaviour through the enable operator. Values of information can be passed between behaviours in LOTOS, using an accept construct.

The LOTOS behaviour examples used in this section are written in a simplified form, where the keywords `process` and `endproc`, behaviour functionality and gate lists are omitted.

4.4.1 Pseudo Event

In LOTOS, the execution of an `exit` corresponds to a pseudo event δ. This δ event executed in conjunction with the enable operator, results in an internal event, according to the LOTOS formal semantics.

Consider for example the following LOTOS process definition:

```
B := B1 >> accept x: Nat in B2 (x)
where
B1 := a ?y: Nat ; exit (y)
B2 (z: Nat) := b !z ; stop
```

According to the formal semantics of LOTOS, behaviour `B` executes `a ?y: Nat`, and an internal event before `b !z` in behaviour `B2` is allowed to be executed. The internal event between `a ?y: Nat` and `b !z` has absolutely no architectural purpose. The pseudo event δ has only been introduced to allow the synchronization of successfully terminating processes.

Using causality-oriented behaviour composition we would define this behaviour as follows:

```
B:= {  B1 (exit (x: Nat)) -> B2 (entry (x))
       start -> B1 (entry)
       where
       B1 := { entry -> a (y: Nat),
               a (y: Nat) -> exit (y) }
       B2 := {entry (x: Nat) -> b (x)
               .... } }
```

In this behaviour, action *a*, being the condition assigned to the *exit* of B_1, is directly a condition for action *b* in B_2, without the need for any sort of pseudo or internal event.

4.4.2 Multiple Entries and Exits

Causality-oriented behaviour composition allows the definition of one or more entries and exits. Using causality-oriented behaviour composition one can also combine behaviours in such a way that a behaviour does not have to terminate in order to enable actions in other behaviours, which makes this mechanism much more flexible than the LOTOS enabling.

We take the example of the behaviour of a FIFO queue. In [2] we can see that the concurrent semantics of this behaviour corresponds to a kind tile structure, in which input actions are interleaved with each other, and output actions are interleaved with each other and dependent of former inputs.

Quality design principles dictate that designs should be structured in terms of combination of simpler components. This is even more evident in the case of repetitive behaviours, such as in the example in Figure 4.17. The FIFO queue behaviour is actually a repetition of cells in which each cell double enables the next one, which means that each input of one cell enables the input of the next cell, and each output of one cell enables the output of the next cell.

This behaviour can only be represented in LOTOS using hidden gates for synchronization between cells. A LOTOS specification of a FIFO queue behaviour can be the following (literally taken from [2]):

```
FIFO := hide start, ok in
(start; exit ||| ok ;exit )
|[start, ok]| Cells
where
Cells := hide start’, ok’ in
   Cell |[start’, ok’]| Cells [start’/start, ok’/ok]
Cell := start; input ; ( start’; exit ||| ok; output ; ok’; exit)
```

In this specification `Cells [start’/start, ok’/ok]` means that process `Cells` is instantiated with gates `start` and `ok` being renamed to `start’` and `ok’`, respectively.

Figure 4.16 depicts the structure of the dynamic evolution of this behaviour.

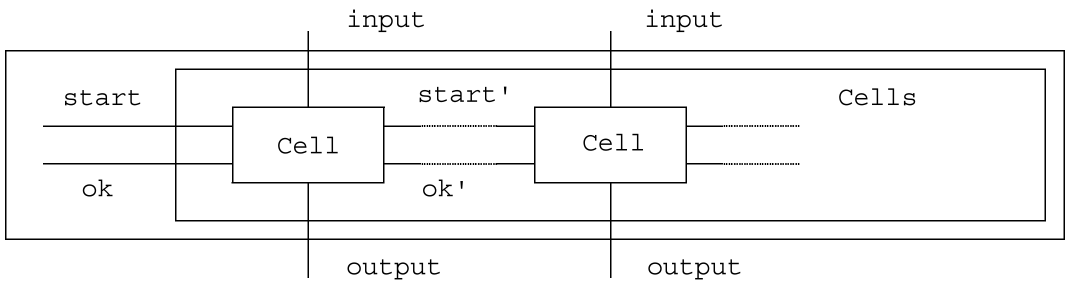


Figure 4.16: Structure of FIFO queue specification in LOTOS

The resource-oriented specification style has introduced for LOTOS in [3] to express compositions of functional entities, mainly because LOTOS does not support the explicit representation of functional entities and their interconnection. In this way internal (hidden) gates and processes are interpreted as interaction points and functional entities. This implies that by looking at the style of the specification above any sensible designer would get in trouble. Especially gates `start` and `ok`, and their successive renamed instantiations, would bring some doubts on whether they have to be implemented in this way or not.

Using causality relations and causality-oriented behaviour composition we can express this behaviour as follows:

$$
\begin{array}{l}
\textit{FIFO} := \{\ \textit{start} \text{ -> } \textit{Cells}\ (\textit{entry}_1),\ \textit{start} \text{ -> } \textit{Cells}\ (\textit{entry}_2), \\
\quad \textit{where} \\
\quad \textit{Cells} := \{ \\
\qquad \textit{entry}_1 \text{ -> } \textit{in},\ \textit{entry}_2 \wedge \textit{in} \text{ -> } \textit{out}, \\
\qquad \textit{in} \text{ -> } \textit{Cells}\ (\textit{entry}_1),\ \textit{out} \text{ -> } \textit{Cells}\ (\textit{entry}_2)\ \}\ \}
\end{array}
$$

In this behaviour, each cell of the FIFO queue double-enables the next one, which conforms more to our intuition about the behaviour of a FIFO queue than the LOTOS specification. The instance of *Cell*s enabled by actions *in* and *out* should be the same, as we have discussed before in section 4.2.4.

Figure 4.17 depicts this behaviour using causality relations.

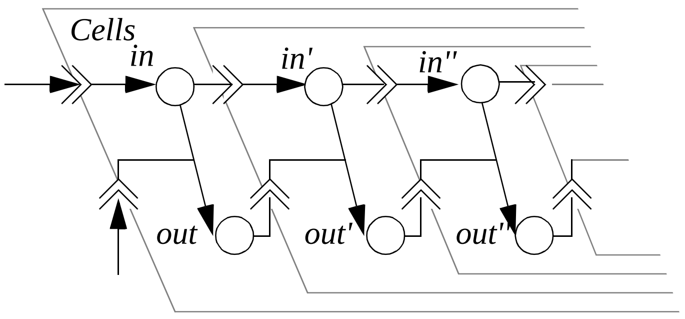


Figure 4.17: Behaviour of a FIFO queue

4.4.3 Flexible Exit Combinations

Using the LOTOS enabling one has to follow the functionality rules in order to determine which conditions of a behaviour enable another behaviour. These conditions determine synchronization on the pseudo event δ. We illustrate this with a behaviour example:

```
B: = (B1 ||| B2 ||| B3 ) >> B4
where ...
```

In this example, behaviours `B1`, `B2` and `B3` have to synchronize on the pseudo event δ in order to enable `B4`, such that all of these behaviours have to successfully terminate. There is no other possibility for the structured composition of these behaviours, such as for example that `B1` and `B2` successfully terminate or `B3` successfully terminates. This limitation has been also pointed out in [1].

Causality-oriented behaviour composition allows behaviour exits to be combined in different ways for enabling other behaviours. The LOTOS example given above is comparable to the following behaviour, in which the successful termination condition is assigned to the exits of B_1, B_2 and B_3:

$$B{:}= \{\ B_1\ (exit) \wedge B_2\ (exit) \wedge B_3\ (exit)\ \text{->}\ B_4\ (entry)\ \ldots\ \}$$

The only possible combination of exits for enabling behaviours in LOTOS corresponds to just one specific case of *exit* combination in causality-oriented behaviour composition, namely the conjunction of exit conditions. Other possible combinations of exit conditions, for example, `B1` and `B2` successfully terminate or `B3` successfully terminates, can be defined as follows:

$$B{:}= \{\ (B_1\ (exit) \wedge B_2\ (exit)) \vee B_3\ (exit)\ \text{->}\ B_4\ (entry)\ \ldots\ \}$$

4.5 References

[1] A. Azcorra. *Formal Modeling of Synchronous Systems*. PhD thesis, Universidad Politécnica de Madrid, Spain, Nov. 1990.

[2] R. Langerak. *Transformations and Semantics for LOTOS*. PhD thesis, University of Twente, Enschede, Netherlands, 1992.

[3] C. A. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma. Specification styles in distributed systems design and verification. *Theoretical Computer Science*, 89:179–206, 1991.

Chapter 5

Constraint-oriented Behaviour Composition

This chapter introduces the *constraint-oriented behaviour composition* structuring technique. This structuring technique consists of representing behaviours in terms of compositions of conditions and constraints on the occurrence of actions. These conditions and constraints are represented in (sub-)behaviours, which are superposed to each other, resulting in the original behaviour. Until now, conditions and constraints of each action could only be defined in a single causality relation. This structuring technique allows actions to be distributed aomngst sub-behaviours, such that each sub-behaviour defines part of the causality relations of these distributed actions.

This chapter is structured as follows: section 5.1 motivates constraint-oriented behaviour composition, section 5.2 evaluates the design freedom for choosing specific decomposition of conditions into behaviours, section 5.3 addresses the distribution of attribute conditions and constraints, section 5.4 defines rules for behaviour composition in constraints and section 5.5 discusses the expressive power gained with the introduction of this structuring technique.

5.1 Motivation

The behaviour definition and structuring techniques presented so far are not sufficient for structuring behaviours in which conditions and constraints on actions are complex. Causality relations are a monolithic form of behaviour representation. Causality-oriented behaviour compositions only allow the distribution of condition and result actions of causality relations over different sub-behaviours, while the actions are kept in a monolithic form. Therefore a structuring technique which supports the decomposing and structuring of conditions and constraints is also necessary.

In most instances of design some alternatives are possible for structuring a behaviour in terms of sub-behaviours that represent parts of the conditions and constraints on actions. These alternatives arise from the following choices:

1. each individual action can be assigned to just one sub-behaviour or it can be shared by more than one sub-behaviour;

2. in case an action is shared by sub-behaviours, its conditions and constraints, which are defined in a single causality relation in a monolithic behaviour definition, can be again distributed over these sub-behaviours in different ways.

These choices determine the design freedom designers have, and should be selected according to specific design objectives and quality principles.

The assignment of actions and parts of conditions and constraints on actions to sub-behaviours actually *define* these sub-behaviours. Therefore, according to the constraint-oriented behaviour composition, for each distributed action a causality relation is defined in each sub-behaviour that share this action. Conversely, the composition of these sub-behaviours should correspond to an certain intended monolithic causality relation.

5.1.1 Example

We illustrate constraint-oriented behaviour composition with an example. Figure 5.1 depicts an arbitrary behaviour defined in terms of causality relations involving actions that must be structured according to the constraint-oriented behaviour composition.

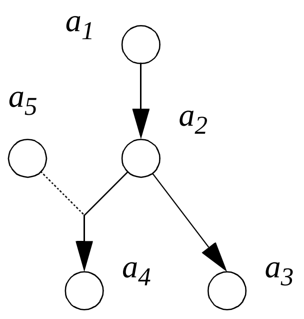


Figure 5.1: Arbitrary monolithic behaviour definition

Firstly we have to determine which actions will be distributed over which sub-behaviours. We suppose that two sub-behaviours B_1 and B_2 will be defined, and that actions a_2, a_3 and a_4 are shared by these sub-behaviours. Consequently actions a_1 and a_5 are not distributed, and are assigned to B_1 and B_2, respectively. Secondly we have to determine how the conditions and constraints on actions a_2, a_3 and a_4 are distributed over B_1 and B_2, The composition of these sub-behaviours implies that conditions and constraints of both B_1 and B_2 apply. The specific design objectives that originated these distribution choices are irrelevant in this discussion.

There are in principle multiple alternatives for the distribution of the conditions and constraints on actions a_2, a_3 and a_4 over B_1 and B_2. Figure 5.2 depicts a possible distribution.

Monolithic actions, which are those actions that are not distributed, are represented in Figure 5.2 as circles, while actions that are distributed are represented as circle segments. This allows one to distinguish distributed and monolithic actions from their graphical representation by inspection. This graphical notation is used consistently throughout this work. In the textual representation we indicate distributed actions with underline, such as $\underline{a}_2$, $\underline{a}_3$ and $\underline{a}_4$ in Figure 5.2.

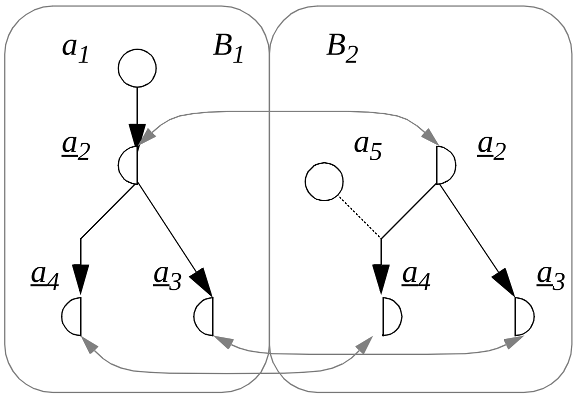


Figure 5.2: Alternative decomposition 1

A certain condition may be placed in more than one sub-behaviour. In Figure 5.2 the condition a_2 enables a_3 is placed in both B_1 and B_2.

Figure 5.3 depicts another alternative decomposition of the original behaviour in sub-behaviours.

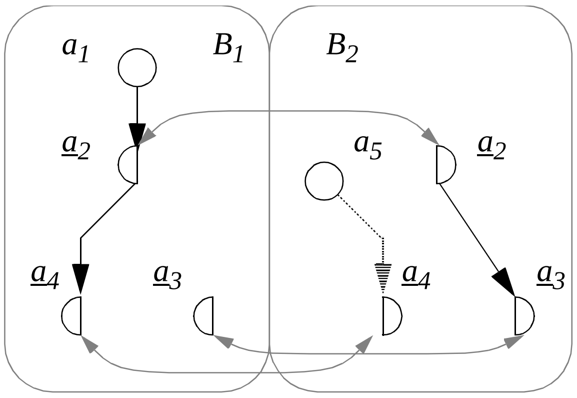


Figure 5.3: Alternative decomposition 2

Figure 5.3 shows that the conditions for a_4, namely the occurrence of a_2 and exclusion by a_5, can be distributed, such that occurrence of a_2 is guaranteed by B_1 and the exclusion by a_5 is guaranteed by B_2.

In some circumstances designers may have no choice of constraint assignment to behaviours. In this example, the condition a_1 enables a_2, considering the distribution of actions to behaviours given, can only be placed in B_1. This can be seen in both Figures 5.2 and 5.3.

5.1.2 Relationship with Entity Domain

Constraint-oriented behaviour composition is a structuring technique defined in the behaviour domain. The assignment of behaviours to functional entities is in principle independent of the way these behaviours are structured. In case different sub-behaviours are assigned to different functional entities, actions are transformed into interactions and a whole structure of interacting functional entities can be defined.

Therefore the constraint-oriented behaviour composition can also be considered as a preparation to functional entity decomposition, since it allows the decomposition of actions into interactions. The decomposition of a behaviour in constraints can be used, for example, to support the decomposition of an interaction system into a collection of interacting functional entities.

For an interaction system between a system and its environment, the assignment of specific constraints to the system or to the environment allows one to obtain the observable behaviour of the system.

Figure 5.4 depicts the possible assignments of behaviour structuring to an interaction system and to the system and its environment.

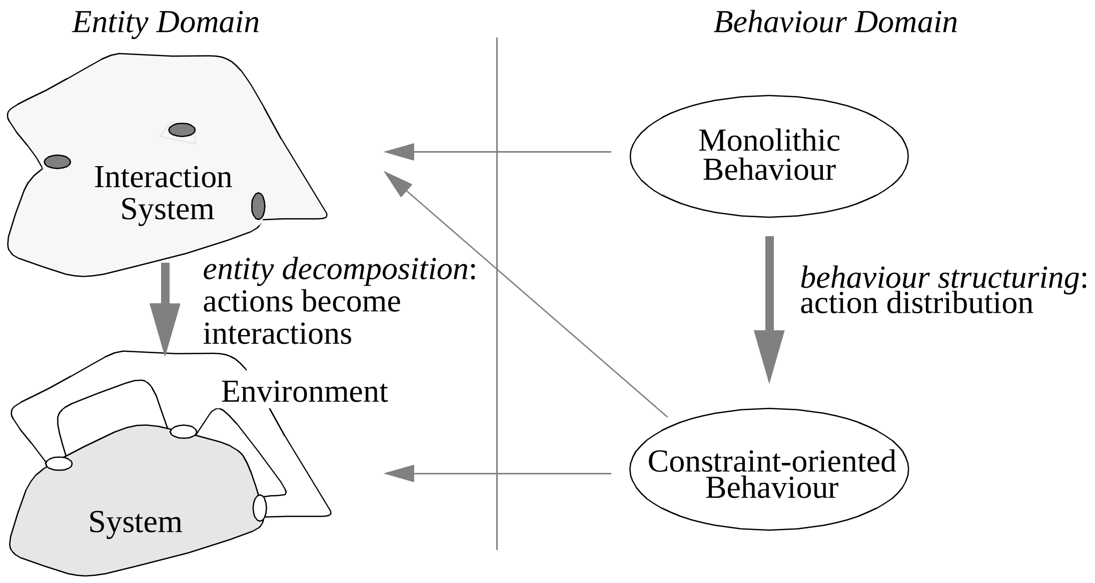

Entity Domain
Behaviour Domain
Interaction System
Monolithic Behaviour
entity decomposition: actions become interactions
behaviour structuring: action distribution
Environment
System
Constraint-oriented Behaviour

Figure 5.4: Assignment of behaviours to entities

Figure 5.5 illustrates the application of constraint-oriented behaviour composition on functional entity decomposition using the example of section 5.1.1.

Figure 5.5 considers that the behaviour of Figure 5.1 is originally assigned to a functional entity F. B_1 and B_2 represent sub-behaviours obtained from the decomposition alternative depicted in Figure 5.3. Sub-behaviours B_1 and B_2 are assigned to functional entities F_1 and F_2, respectively. We suppose that F_1 and F_2 interact through interaction points ip_1 and ip_2. The decomposition is complete when we assign interaction points to the location attributes of original actions a_2, a_3 and a_4, turning them into interactions. We consider for example that a_2 and a_3 happen at ip_1 and that a_4 happens at ip_2.

We conclude that a proper choice of sub-behaviours in a constraint-oriented behaviour enables functional entity decomposition. Separation of concerns can be used for the definition of sub-behaviours that are concerned with orthogonal aspects of the total behaviour. Chapter 6 addresses entity decomposition from the point of view of behaviour refinements that may be necessary for defining compositions of sub-behaviours that conform to a monolithic behaviour.

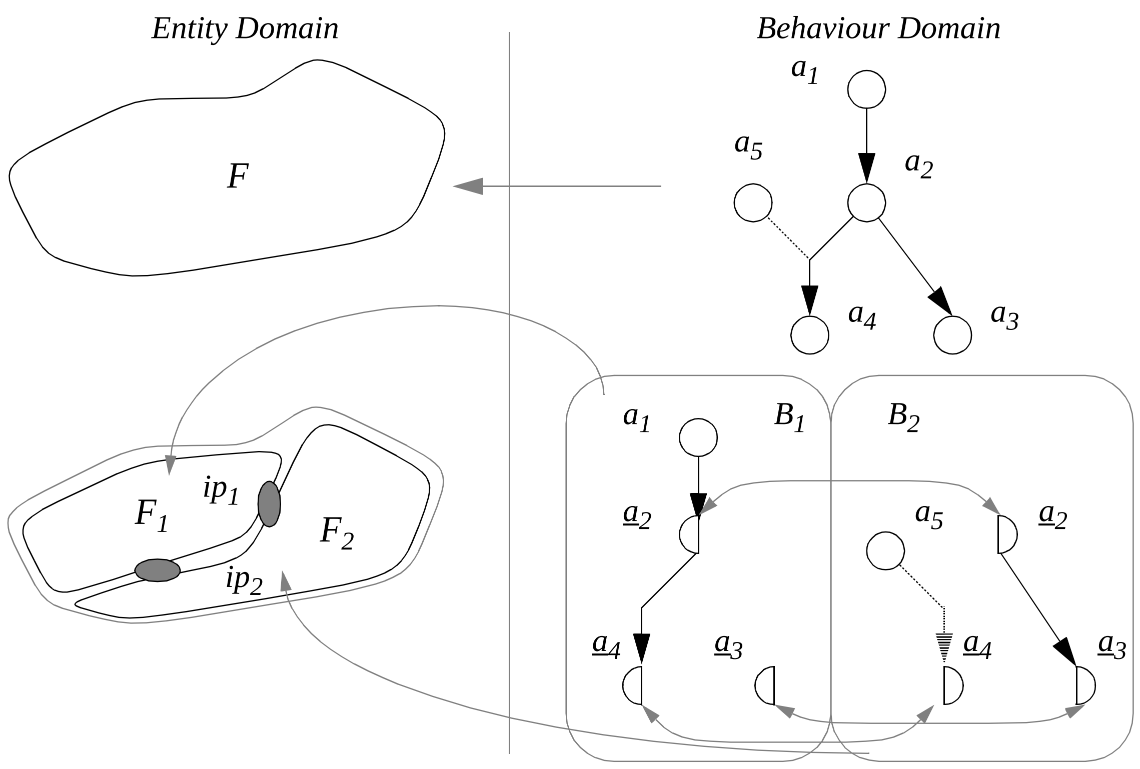


Figure 5.5: Constraint-oriented behaviour composition and entity decomposition

5.2 Degrees of Freedom

This section systematically discusses all the possible ways causality relations can be structured in terms of constraint-oriented behaviour compositions. This exercise shall determine the degrees of freedom that designers have when structuring behaviours using this technique, resulting in a set of general rules for structuring monolithic behaviours in terms of sub-behaviours representing constraints.

Throughout this section we consider that a certain behaviour B is decomposed in two sub-behaviours B_1 and B_2. Actions that are not distributed are represented with *italics* (e.g. a, b and c), while actions that are distributed are represented in <u>*underlined italics*</u> (e.g. $\underline{a}$, $\underline{b}$, and $\underline{c}$).

5.2.1 Decomposition of Conjunction (*and* Conditions)

We consider that a certain behaviour B contains a causality relation $a \wedge b \rightarrow c$. We also suppose that a, b and c may be or may be not distributed over behaviours B_1 and B_2. This analysis concentrates exclusively on the causality relation of c.

Possible decompositions are presented in terms of groups, which correspond to specific distributions of actions over behaviours B_1 and B_2. These groups are systematically presented in the sequel.

Group I: Distribution of One Condition

This group covers the distribution of one condition action, *a* or *b*, but not the result action *c*. Due to the fact that action *c* is not distributed, the condition $a \wedge b$ cannot be decomposed, i.e. the condition must be defined in the same behaviour where *c* is defined.

Figure 5.6 depicts a possible decomposition of the monolithic causality relation. The distribution of *b* in place of *a* and the mirror images in which B_1 and B_2 are exchanged would cover all the possible decompositions for this group.

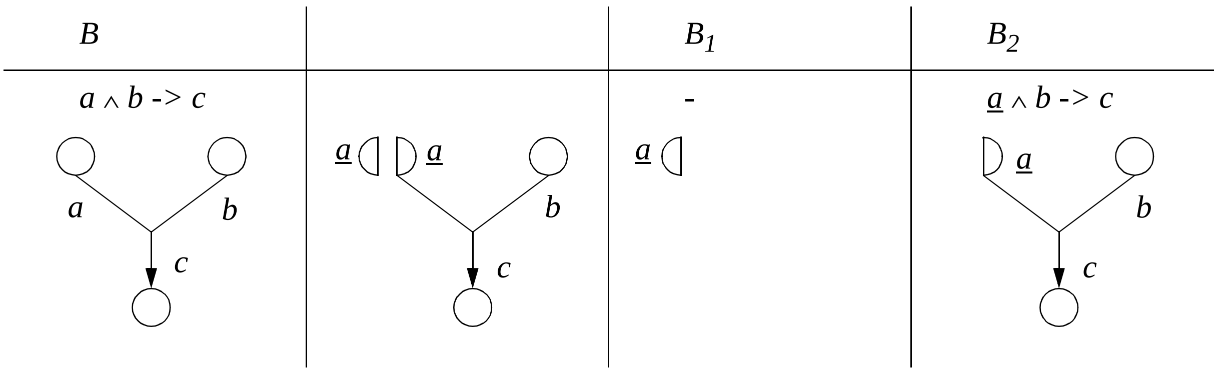


Figure 5.6: Distribution of one condition action

In Figure 5.6 the assignment of *c* to B_2 makes it impossible to have conditions and constraints on *c* defined in B_1.

Group II: Distribution of Result

This group covers the distribution of the result action *c*, but not the condition actions *a* and *b*. In this group we consider two possibilities, namely *a* and *b* are in the same behaviour, or *a* and *b* are in different behaviours. In case *a* and *b* are in the same behaviour, the condition $a \wedge b$ can only be defined in this behaviour. In case *a* and *b* are in different behaviours, the condition is distributed, such that their composition corresponds to the original condition.

Figure 5.6 depicts these two possible decompositions. A mirror image in which B_1 and B_2 are exchanged would cover all possibilities.

In decomposition (1) of Figure 5.6, * -> $\underline{c}$ indicates that $\underline{c}$ must be enabled in B_1 at the moment that *a* and *b* occur. Since *a* and *b* do not belong to B_1, it is not possible in B_1 to know when *a* and *b* have occurred. This means that $\underline{c}$ must be enabled in B_1 from the beginning, either by a start or an entry.

Group III: Distribution of Both Conditions

This group covers the distribution of both condition actions *a* and *b*, but not the result action *c*. Similarly to group I, due to the fact that action *c* is not distributed, the condition $a \wedge b$ cannot be decomposed, i.e. all conditions for *c* must be defined in the same behaviour where *c* is defined.

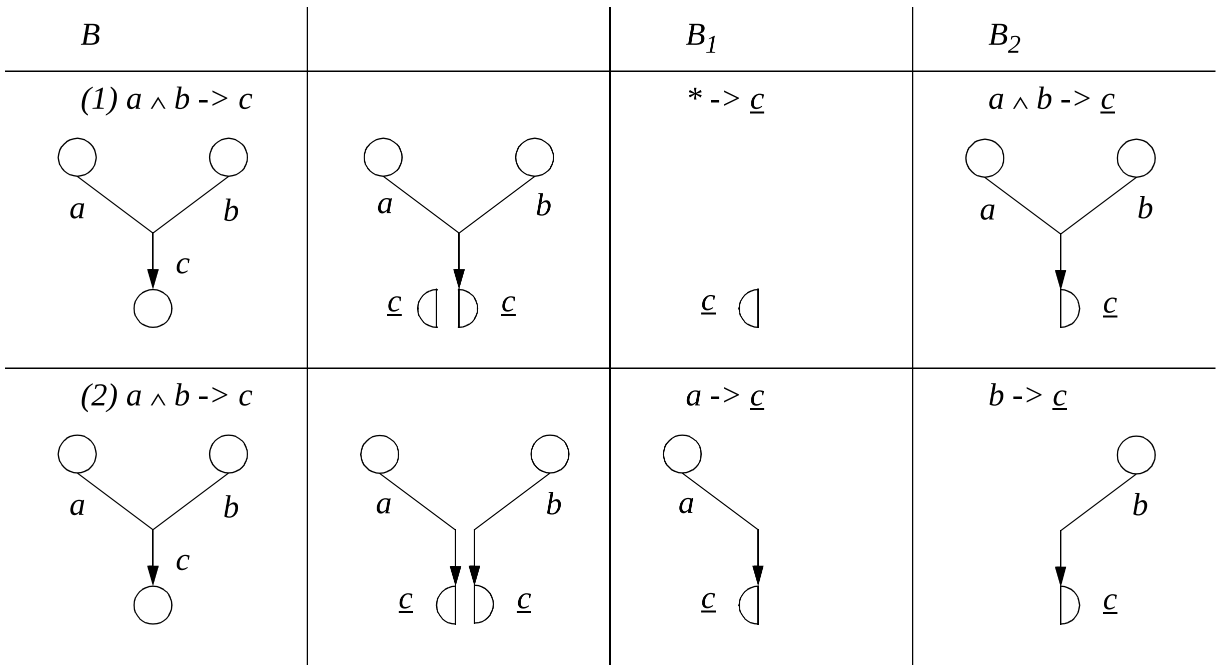


Figure 5.7: Distribution of one result action

Figure 5.8 depicts a possible decomposition. Mirror images in which B_1 and B_2 are exchanged would cover all the possibilities for this group.

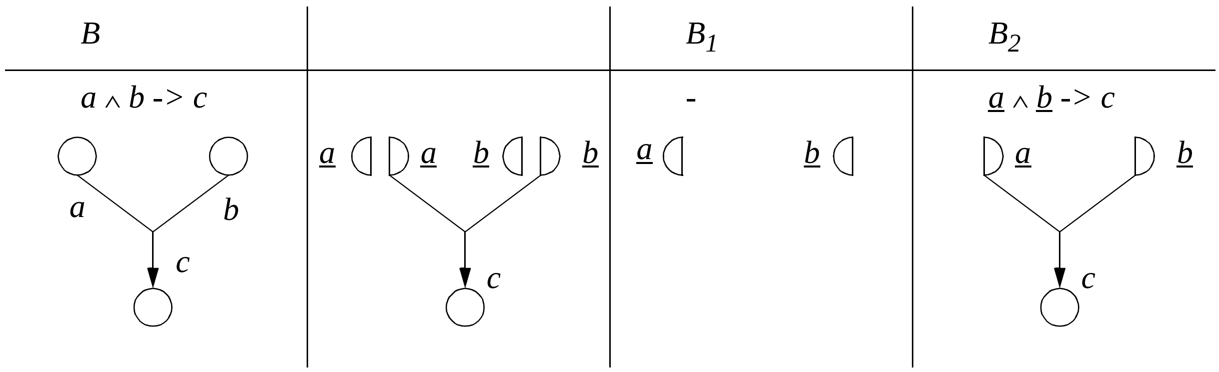


Figure 5.8: Distribution of both condition actions

Group IV: Distribution of One Condition and Result

This group covers the distribution of one condition action, *a* or *b*, and the result action *c*. We consider that *a* has been distributed, without loss of generality. Since *c* and *a* are distributed, we have some freedom in the assignment of the causality relations between *a* and *c* to B_1 and B_2: we can place the $\underline{a} \wedge b$ condition in one of the behaviours, we can assign the causality between *a* and *c* to one behaviour and the causality between *b* and *c* to the other, or we can place the $\underline{a} \wedge b$ condition in one of the behaviours and duplicate the causality between *a* and *c* in the other behaviour.

Figure 5.9 depicts these three possible decompositions. Mirror images in which B_1 and B_2 are exchanged, and the distribution of *b* in place of *a* would cover all the possibilities.

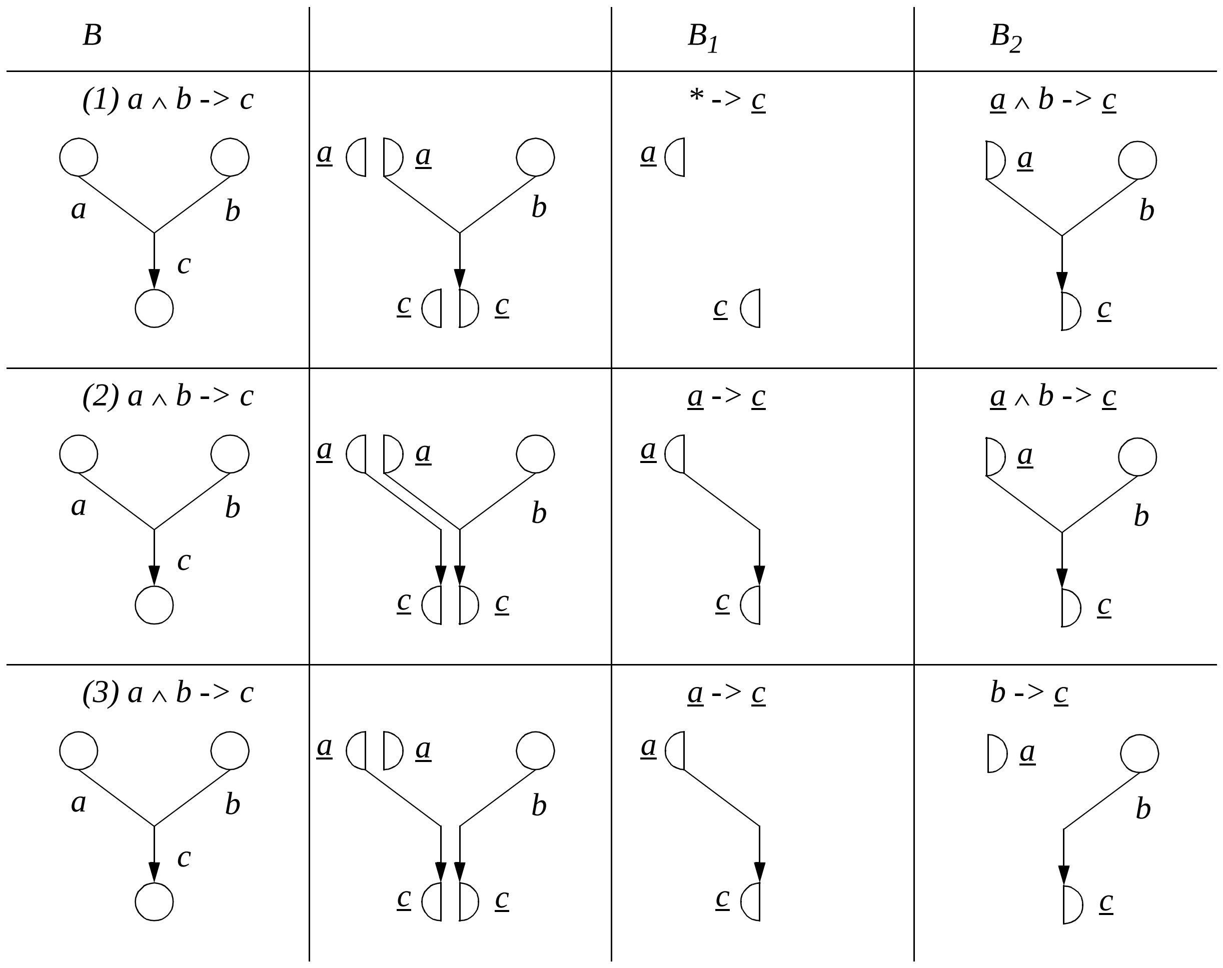


Figure 5.9: Distribution of one condition and the result

In decomposition (1) of Figure 5.9, * -> <u>c</u> indicates that *c* must be enabled in B_1 at the moment that *a* and *b* occur. Since *a* belongs to B_1, *c* may be enabled in B_1 from the beginning, either by a start or an entry, or by any condition which is also a necessary condition of *a*.

Although the duplication of constraints in B_1 and B_2 in decomposition (2) of Figure 5.9 may seem technically undesirable, in the more general case, where references to attributes are possible, this kind of decomposition may be used to express *separation of concerns*. Suppose for example that *a* has values v_1 and v_2, which are used for the determination of the value attributes of *c*. In this case we can use B_1 to constrain the reference to v_1 and B_2 to constrain the reference to v_2. This remark applies to all cases of duplication of constraints that appear in the sequel.

Group V: Distribution of Both Conditions and Result

This group covers the distribution of both condition actions *a* and *b*, and the result action *c*. There are many possible decompositions in this group, which can be systematically generated by considering: (i) minimal distribution, where the condition $a \wedge b$ (actually $\underline{a} \wedge \underline{b}$) is placed in one of the behaviours, (ii) duplications of the condition $a \wedge b$ and (iii) maximal distribution, where each behaviour contains either a causality relation between *a* and *c*, or between *b* and *c*.

Figure 5.10 depicts these four possible decompositions. Mirror images in which B_1 and B_2 are exchanged would cover all the possibilities for this group.

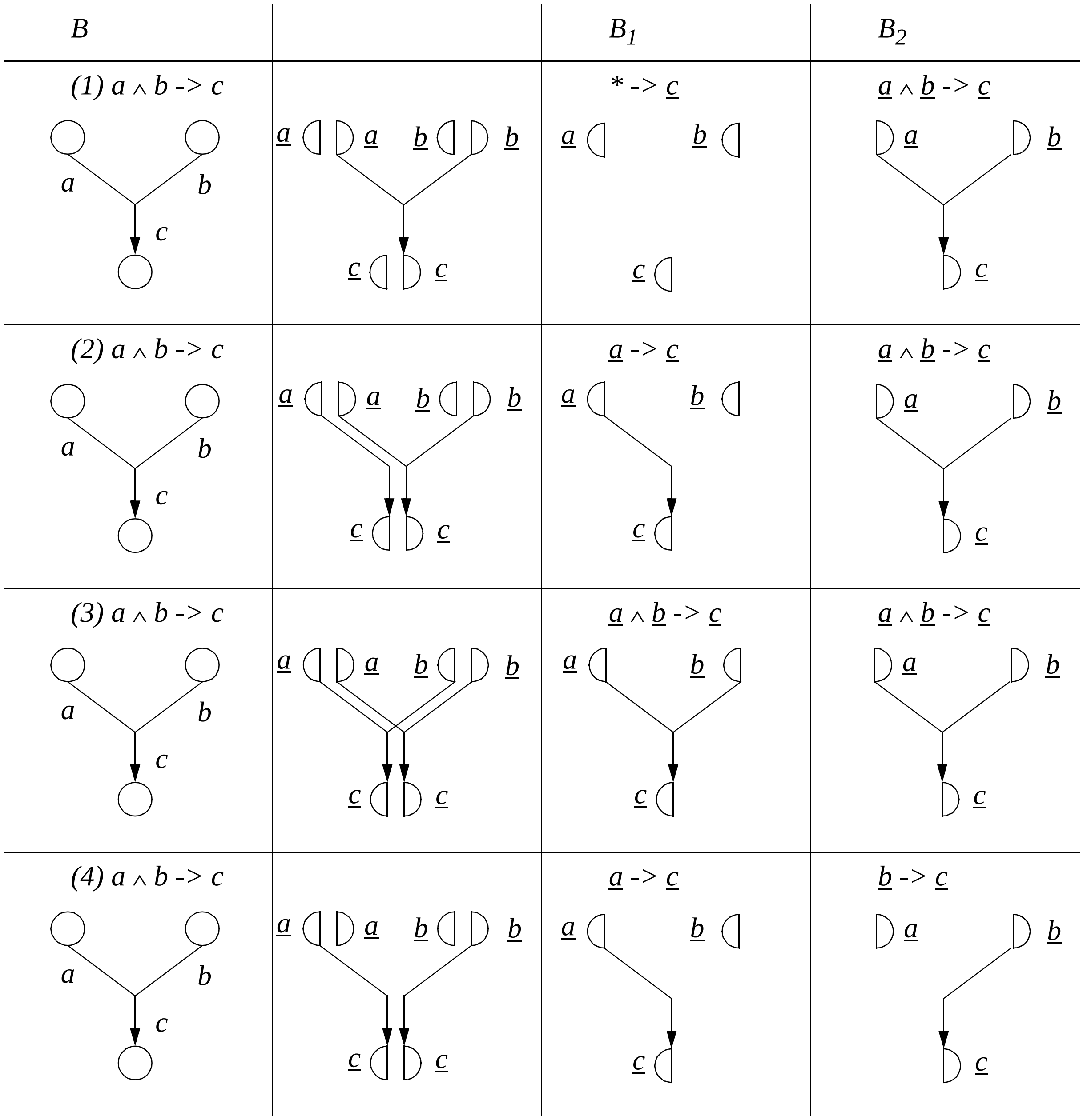


Figure 5.10: Distribution of all actions

In decomposition (1) of Figure 5.10, * -> c indicates that *c* must be enabled in B_1 at the moment that *a* and *b* occur. Since both *a* and *b* belong to B_1, *c* may be enabled in B_1 from the beginning, either by a start or an entry, by any condition which is also a necessary condition of *a* or *b*, or by a conjunction of necessary conditions of *a* and *b*.

Transitivity of Necessary Conditions

In all five groups, references to actions that are necessary conditions for *a* or *b* or both could also be made in the conditions of *c*, in conjunction ($\wedge$ combination). However this can only be done if *a* or *b* or both belong to the behaviour in which the condition is defined.

Figure 5.11 depicts an example in which $a \wedge b \text{ -> } c$ is decomposed and *a* and *c* are distributed, which corresponds to group IV. In case for some action *d*, there is causality relation say $d \text{ -> } a$ in the monolithic behaviour, then $d \text{ -> } \underline{c}$ in B_1 and $\underline{a} \wedge b \text{ -> } \underline{c}$ in B_2 is also a correct decomposition of the causality relation of *c*. Since *d* is a condition for *a*, and *a* is a condition for *c*, *d* is indirectly a condition for *c*, which follows from the transitivity of causality. Including this condition in the causality relation of *c* results in a correct decomposition.

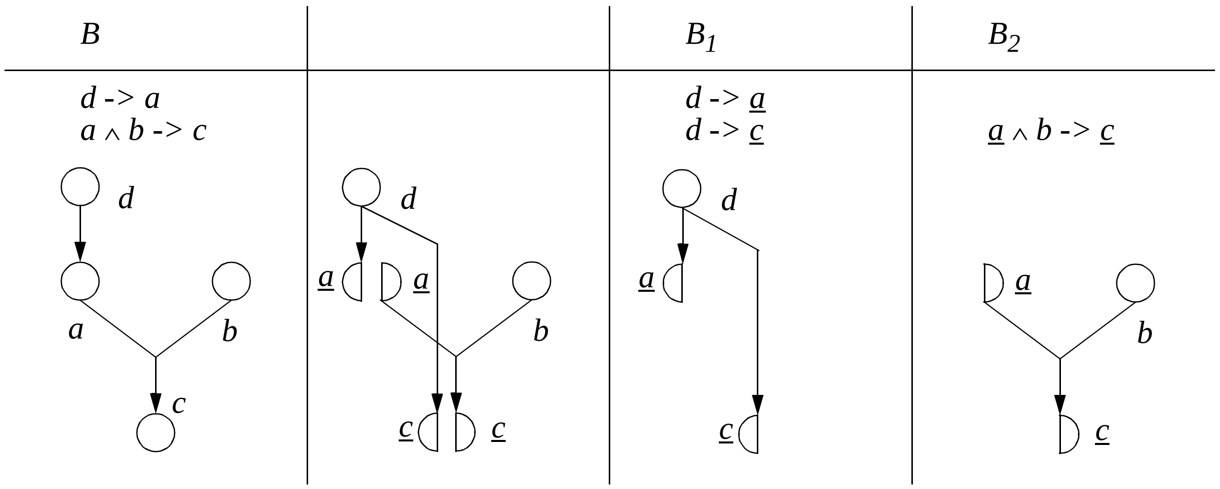


Figure 5.11: Transitivity of necessary conditions

However, the use of references to actions that were not originally intended to be referred to may cause a lot of serious complications. From the point of view of the design goals, if *a* is a necessary condition for *c*, and also contains the values established in *d* in its retained values, a direct reference to *d* in the conditions of *c* is redundant in a constraint-oriented decomposition of this behaviour. Another argument is that in case of complex behaviours, the total of the possibilities for reference may increase to an uncontrollable amount. Correctness conditions may get far too complex, when compared to a more constrained use of references. Therefore we assume that, for the reasons above, explicit reference to earlier causes should be avoided.

5.2.2 Decomposition of Disjunction (*or* Conditions)

We consider that a certain behaviour *B* contains a causality relation $a \vee b \text{ -> } c$. We also suppose that *a, b* and *c* may be or may be not distributed over behaviours B_1 and B_2. This analysis concentrates exclusively on the causality relation of *c*.

Possible decompositions are presented in groups, where specific distributions of actions are considered. These groups are systematically presented in the sequel.

Group I: Distribution of One Condition

This group covers the distribution of one condition action, *a* or *b*, and not the result action *c*. Due to the fact that action *c* is not distributed, the condition $a \vee b$ cannot be decomposed, i.e. the condition must be defined in the same behaviour where *c* is defined.

Figure 5.12 depicts a possible decomposition. The distribution of *b* in place of *a* and the mirror images in which B_1 and B_2 are exchanged would cover all the possibilities for this group.

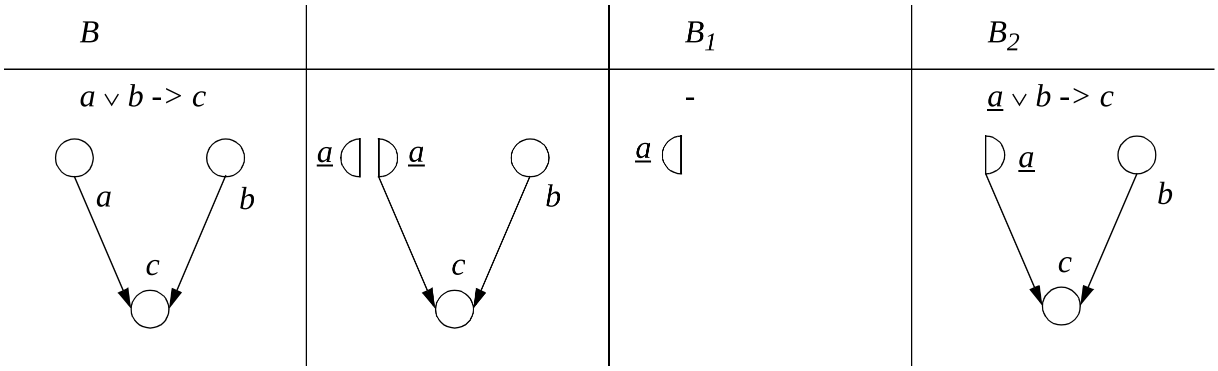


Figure 5.12: Distribution of one condition action

Group II: Distribution of Result

This group covers the distribution of the result action *c*, but not the condition actions *a* and *b*. Again there are two possibilities, namely *a* and *b* are in the same behaviour or *a* and *b* are in different behaviour. In the case of disjunction, however, only if *a* and *b* are in the same behaviour a decomposition that corresponds to the original monolithic causality relation is possible.

Figure 5.13 depicts this possible decomposition. A mirror image in which B_1 and B_2 are exchanged would cover all possibilities for this group.

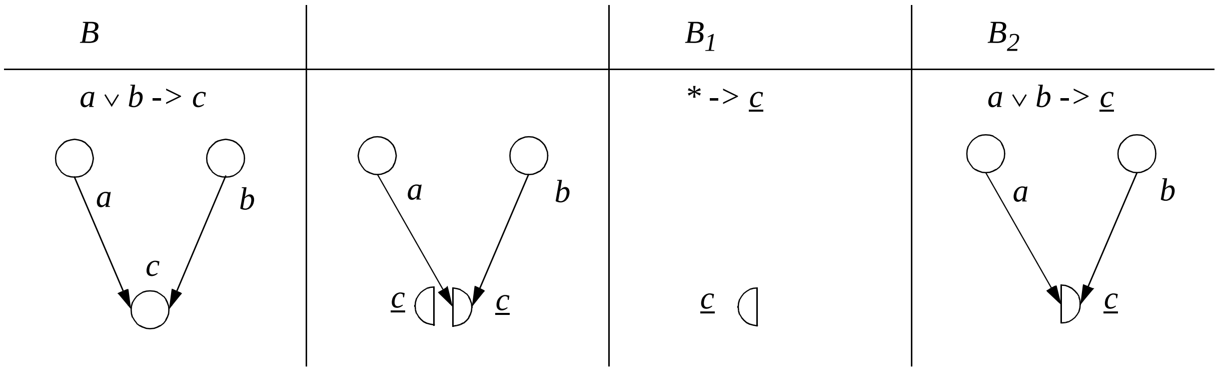


Figure 5.13: Distribution of the result action

In Figure 5.13, * -> *c* indicates that *c* must be enabled in B_1 at the moment that *a* or *b* occur. Since *a* and *b* do not belong to B_1, it is not possible in B_1 to know when *a* and *b* have occurred. This means that *c* must be enabled in B_1 from the beginning, either by a start or an entry.

Group III: Distribution of Both Conditions

This group covers the distribution of both condition actions *a* and *b*, but not the result action *c*. Similarly to group I, due to the fact that action *c* is not distributed, the condition $a \vee b$ cannot be decomposed, such that the condition must be defined in the same behaviour where *c* is defined.

Figure 5.14 depicts this possible decomposition. A mirror image in which B_1 and B_2 are exchanged would cover all the possibilities for this group.

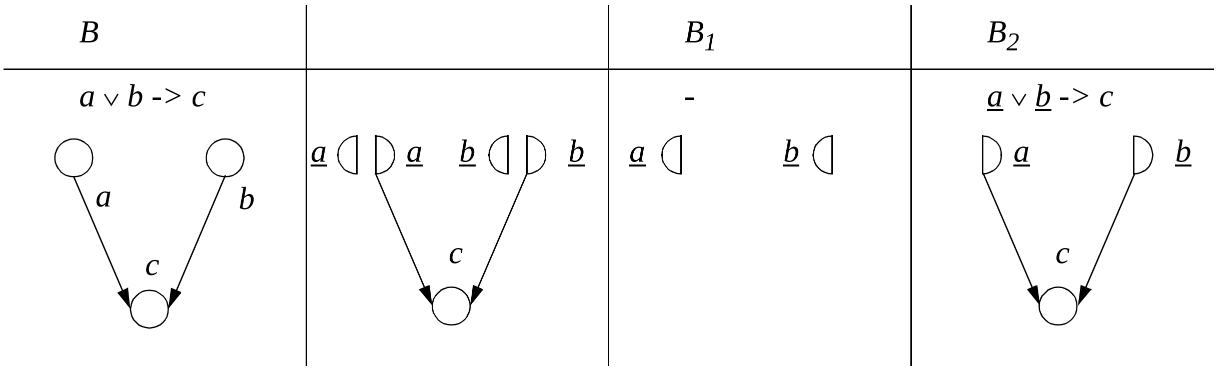


Figure 5.14: Distribution of both condition actions

Group IV: Distribution of One Condition and Result

This group covers the distribution of one condition action, *a* or *b*, and the result action *c*. Since the condition $a \vee b$ cannot be correctly generated if *a* and *b* are in different behaviours, only one possible correct decomposition exists for a certain assignment of actions to behaviours.

Figure 5.15 depicts the only possible decomposition for this group. Mirror images in which B_1 and B_2 are exchanged, and the distribution of *b* in place of *a* would cover all possibilities for this group.

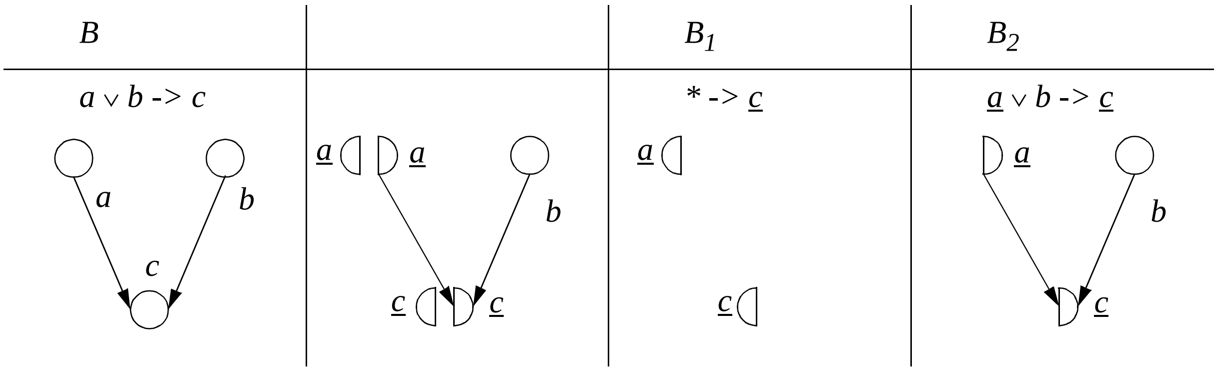


Figure 5.15: Distribution of one condition action and the result action

In Figure 5.15, * -> $\underline{c}$ indicates that *c* must be enabled in B_1 at the moment that *a* or *b* occur. Action *a* belongs to B_1, but in this case enabling *c* by a condition which is also a necessary condition of *a* yields an incorrect decomposition, since *a* may not be even enabled when *b* happens, and the occurrence of *b* is sufficient for *c* to be enabled. This means that *c* must be enabled in B_1 from the beginning, either by a start or an entry.

Group V: Distribution of Both Conditions and Result

This group covers the distribution of both condition actions *a* and *b*, and the result action *c*. Again, since the condition $a \vee b$ cannot be correctly generated if *a* and *b* are in different behaviours, there are only two possible decompositions for a given assignment of actions to behaviours in this group: condition $a \vee b$ (actually $\underline{a} \vee \underline{b}$) is placed in one of the sub-behaviours, or condition $\underline{a} \vee \underline{b}$ is placed in both sub-behaviours.

Figure 5.16 depicts the alternatives that belong to this group for a given assignment of actions to behaviours. Mirror images in which B_1 and B_2 are exchanged would cover all the possibilities for this group.

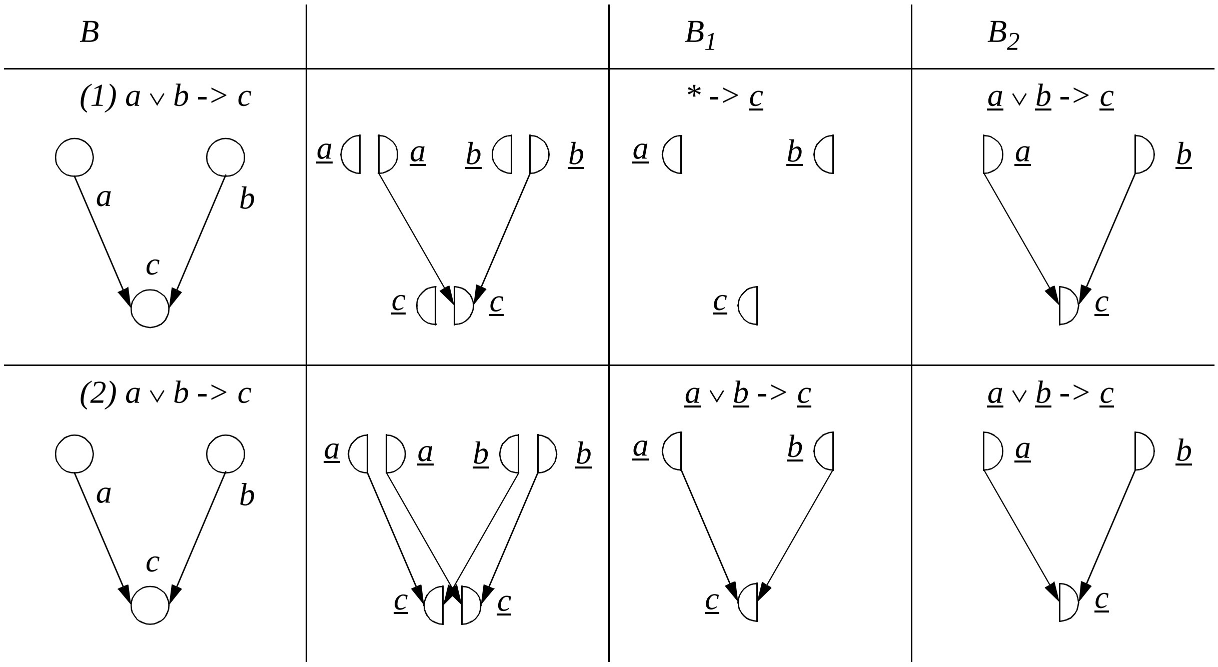


Figure 5.16: Distribution of all actions

In decomposition (1) of Figure 5.16, * -> $\underline{c}$ indicates that *c* must be enabled in B_1 at the moment that *a* or *b* occur. Since both *a* or *b* belong to B_1, *c* may be enabled in B_1 from the beginning, either by a start or an entry, or a disjunction of necessary conditions of *a* and *b*.

Decomposition (2) of Figure 5.16 should be carefully considered. According to the definition of disjunction the original action *c* should be either caused by *a* or by *b*, but not by both of them. This implies that this decomposition is only correct if both sub-behaviours agree on the same cause for *c*, either *a* or *b*. Such condition is automatically satisfied if *a* and *b* are defined such that there is no possible execution of this behaviour in which both *a* and *b* occur, since in this case the occurrence of *c* can be uniquely related to either *a* or *b* in both sub-behaviours. However this condition may be too restrictive. More generally this decomposition is correct if the sub-behaviours B_1 and B_2 are always able to agree on the same cause for *c*, otherwise this decomposition is incorrect.

5.2.3 Decomposition of Exclusion

We consider that a certain monolithic behaviour *B* contains a causality relation $a \wedge \neg b$ -> c. We also suppose that *a, b* and *c* may be or may be not distributed over behaviours B_1 and B_2. This analysis concentrates exclusively on the causality relation of *c*.

Table 5.1 shows all the possible decompositions for this causality relation. Figures and explanations are omitted, since these decompositions are similar to the ones obtained in Section 5.2.1.

Table 5.1: Possible decompositions of $a \wedge \neg b$ -> c

Decomposition	Distributed Actions	B_1	B_2
(1)	*a*	-	$\underline{a} \wedge \neg b$ -> c
(2)	*b*		$a \wedge \neg \underline{b}$ -> c
(3)	*c*	* -> $\underline{c}$	$a \wedge \neg b$ -> $\underline{c}$
(4)	*c*	a -> $\underline{c}$	$\neg b$ -> $\underline{c}$
(5)	*a, b*	-	$\underline{a} \wedge \neg \underline{b}$ -> c
(6)	*a, c*	* -> $\underline{c}$	$\underline{a} \wedge \neg b$ -> $\underline{c}$
(7)	*a, c*	$\underline{a}$ -> $\underline{c}$	$\underline{a} \wedge \neg b$ -> $\underline{c}$
(8)	*a, c*	$\underline{a}$ -> $\underline{c}$	$\neg b$ -> $\underline{c}$
(9)	*b, c*	$a \wedge \neg \underline{b}$ -> $\underline{c}$	$\neg \underline{b}$ -> $\underline{c}$
(10)	*b, c*	a -> $\underline{c}$	$\neg \underline{b}$ -> $\underline{c}$
(11)	*b, c*	$a \wedge \neg \underline{b}$ -> $\underline{c}$	* -> $\underline{c}$
(12)	*a, b, c*	* -> $\underline{c}$	$\underline{a} \wedge \neg \underline{b}$ -> $\underline{c}$
(13)	*a, b, c*	$\underline{a}$ -> $\underline{c}$	$\underline{a} \wedge \neg \underline{b}$ -> $\underline{c}$
(14)	*a, b, c*	$\underline{a} \wedge \neg \underline{b}$ -> $\underline{c}$	$\underline{a} \wedge \neg \underline{b}$ -> $\underline{c}$
(15)	*a, b, c*	$\underline{a}$ -> $\underline{c}$	$\neg \underline{b}$ -> $\underline{c}$

In decomposition (3) of Table 5.1, * -> $\underline{c}$ indicates that *c* must be enabled in B_1 at the moment that *a* occurs given that *b* has not occurred. Since *a* and *b* do not belong to B_1, it is not possible in B_1 to know when *a* and *b* have occurred. This means that *c* must be enabled in B_1 from the beginning, either by a start or an entry. In decomposition (6) of Table 5.1, * -> $\underline{c}$ indicates that *c* must be enabled in B_1 at the moment that *a* and *b* occur. Since *a* belongs to B_1, *c* may be enabled in B_1 from the beginning, either by a start or an entry, or by any condition which is also a necessary condition of *a*. In decomposition (12) of Table 5.1, * -> $\underline{c}$ indicates that *c* must be enabled in B_1 at the moment that *a* occurs given that *b* has not occurred. Since both *a* and *b* belong to B_1, *c* may be enabled in B_1 from the beginning, either by a start or an entry, or by any condition which is also a necessary condition of *a*.

The decomposition rules for the causality relation $a \vee \neg b$ -> c can be derived directly from the rules of section 5.2.2 and will not be further discussed in this text.

5.2.4 General Decomposition Rules

Observing the decomposition options presented before we can conclude the following:

- there are many possibilities for the decomposition of necessary conditions (conjunction). Necessary conditions can be distributed or not amongst the sub-behaviours, but all conditions have to be defined in at least one of the sub-behaviours;
- there are not so many possibilities for the decomposition of sufficient conditions (disjunction) as in the case of necessary condition (conjunction). A sufficient condition itself cannot be decomposed, and it must be placed in one of the sub-behaviours or duplicated (placed in both);
- the rules for decomposition of exclusion are similar to the rules for the decomposition of causality.

The constraint-oriented structuring technique consists of distributing actions over sub-behaviours, and combining these sub-behaviours. For each distributed action the conditions and constraints of all sub-behaviours apply, in a sort of conjunction. Being similar to the behaviour composition rules, the decomposition of conjunctions (*and* conditions) is favoured and therefore offers more possibilities than the decomposition of disjunction (*or* conditions).

The rules given above can be generalized, such that the following holds:

1. any condition can be defined as $\vee_j (\wedge_i \{c_{ij}\})$ or $\wedge_j (\vee_i \{c_{ij}\})$ where c_{ij} are elementary conditions of the form a_k (causality) or $\neg\, a_k$ (exclusion)
2. given a causality relation $C_1 \vee C_2$ -> c, where C_1 and C_2 are arbitrary conditions of the form $\wedge_i \{c_{ij}\}$, and c_{ij} are conditions of the form a_k or $\neg\, a_k$, one can decompose this causality relation according to the possibilities presented in Table 5.2;

Table 5.2: Decompositions of $C_1 \vee C_2$ -> c

B_1	B_2
$C_1 \vee C_2$ -> c	-
$C_1 \vee C_2$ -> $\underline{c}$	* -> $\underline{c}$
$C_1 \vee C_2$ -> $\underline{c}$	$C_1 \vee C_2$ -> $\underline{c}$

3. given a causality relation $C_1 \wedge C_2$ -> c, where C_1 and C_2 are arbitrary conditions of the form $\vee_i \{c_{ij}\}$, where c_{ij} are conditions of the form a_k or $\neg\, a_k$, one can decompose this causality relations according to the possibilities presented in Table 5.3.

In Table 5.2 and Table 5.3 mirror images in which B_1 and B_2 are exchanged have been ignored. The actions of C_1 and C_2 can be also decomposed or not, according to the possibilities imposed by their causality relations and specific assignments of actions to behaviours.

Table 5.3: Decompositions of $C_1 \wedge C_2$ -> c

B_1	B_2
$C_1 \wedge C_2 \text{ -> } c$	-
$C_1 \text{ -> } \underline{c}$	$C_2 \text{ -> } \underline{c}$
$C_1 \wedge C_2 \text{ -> } \underline{c}$	$* \text{ -> } \underline{c}$
$C_1 \wedge C_2 \text{ -> } \underline{c}$	$C_1 \wedge C_2 \text{ -> } \underline{c}$
$C_1 \text{ -> } \underline{c}$	$C_1 \wedge C_2 \text{ -> } \underline{c}$
$C_2 \text{ -> } \underline{c}$	$C_1 \wedge C_2 \text{ -> } \underline{c}$

5.2.5 Decomposition of Initial Actions

Initial actions enabled by entries in an intended monolithic behaviour may also be distributed when constraint-oriented behaviour composition is applied.

Figure 5.17 illustrates the distribution of initial actions enabled by an entry.

B		B_1	B_2
entry -> a *entry* -> b $a \wedge b$ -> c		$\underline{entry}$ -> $\underline{a}$ $\underline{entry}$ -> $\underline{b}$ $\underline{a}$ -> $\underline{c}$	$\underline{entry}$ -> $\underline{a}$ $\underline{entry}$ -> $\underline{b}$ $\underline{b}$ -> $\underline{c}$

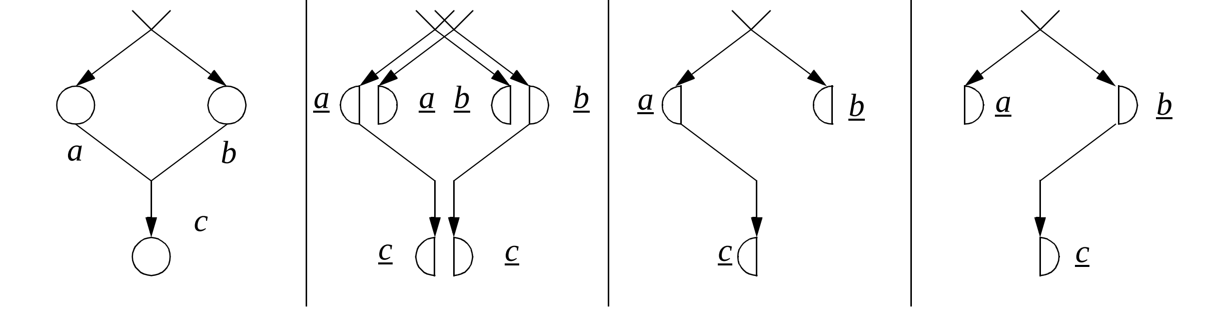


Figure 5.17: Initial action decomposition with entry duplication

In this example, actions a and b are enabled by the same entry in the monolithic behaviour. This entry is distributed over two sub-behaviours B_1 and B_2 together of the actions a and b in the constraint-oriented composition, such that these actions can be enabled. Actions a and b should be enabled by the conditions assigned to the entry of behaviour B. In the corresponding constraint-oriented composition, a and b are enabled in both sub-behaviours B_1 and B_2 by the same entry.

Figure 5.18 depicts another decomposition alternative for the distribution of an entry.

In this example the entry of the monolithic behaviour is distributed, but actions a and b are not. However, action c should be enabled from the beginning of the behaviour in B_1, since it is not possible in B_1 to know when both a and b take place. An alternative is to use the distributed entry to enable c in B_1, which is depicted in Figure 5.18.

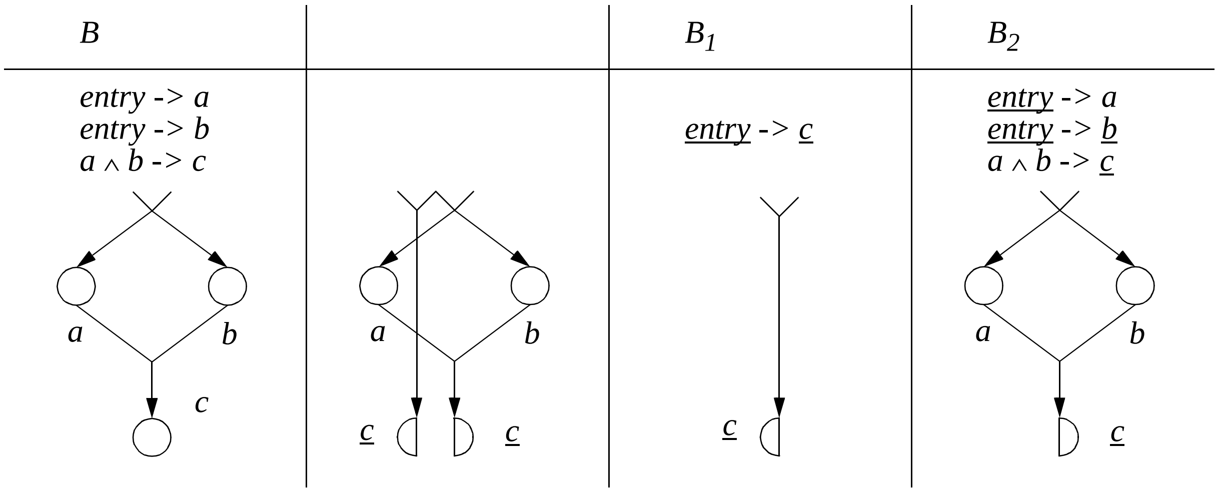


Figure 5.18: Initial action decomposition with entry assigned to result action

We conclude that there is some design freedom in the distribution of initial actions and in the assignment of entry conditions to actions.

5.3 Distribution of Attribute Conditions and Constraints

So far we have only discussed the distribution of action conditions in terms of causality and exclusion, without considering action attributes. However, realistic instances of design can only be addressed if action attributes are also considered.

Attribute conditions and constraints can also be distributed over sub-behaviours, which in combination correspond to a certain intended monolithic behaviour. Distribution choices for conditions and constraints on action attributes over sub-behaviours and the implications of these choices are discussed in the sequel.

5.3.1 Values of Information

Conditions and constraints on values of information of an action can be distributed over sub-behaviours, such that this action only occurs if all conditions of all sub-behaviours are satisfied and with values of information that comply to all constraints of all sub-behaviours.

Conditions on Values of Information

We illustrate the distribution of conditions on action values with an example. Consider the following monolithic causality relation:

$$a_1\ (v_1\text{: } Nat)\ [1 \le v_1 < 5] \text{ -> } a_2$$

This condition could be distributed over two sub-behaviours in the following way:

$$\underline{a}_1\ (v_1\text{: } Nat)\ [v_1 \ge 1] \text{ -> } \underline{a}_2\ (\text{* } 1\text{*})$$

$\underline{a}_1$ *(*v_1*: Nat) [*$v_1 < 5$*] ->* $\underline{a}_2$ *(* 2 *)*

In this distribution of conditions, causality relation *(* 1 *)* is responsible for condition $v_1 \geq 1$, while causality relation *(* 2 *)* is responsible for condition $v_1 < 5$, such that only if both conditions are satisfied a_2 is allowed to occur. This distribution of conditions is a correct implementation of the monolithic causality relation above.

Figure 5.19 shows this distribution example.

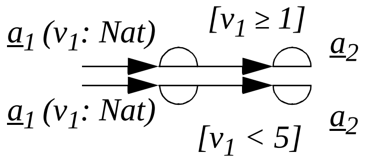

$\underline{a}_1$ (v_1: Nat)
[$v_1 \geq 1$]
$\underline{a}_2$
$\underline{a}_1$ (v_1: Nat)
[$v_1 < 5$]
$\underline{a}_2$

Figure 5.19: Distribution of conditions on action values

In this example we do not consider the conditions and constraints of a_1, which should be considered in its causality relation and in its distribution.

Constraints on Action Value

Similarly to [9], we consider in this work that there are three basic forms of value establishment: value checking, value passing and value negotiation. These forms of value establishment influence the choices for the distribution of constraints on action values over sub-behaviours. Since these forms of value establishment have been extensively discussed, amongst others in [9], [8] and [1], we only briefly illustrate them with simple examples below. In these examples we consider only two sub-behaviours, for the sake of simplicity. These forms of value establishment determine the degrees of freedom one has to decompose constraints on action values.

1. *value checking*: both sub-behaviours offer the same value of information, such that they synchronize on this value;
2. *value passing*: one sub-behaviour offers a value of information, while the other imposes no constraints on the values of information to be established. In this way a value of information is passed from one sub-behaviour to the other.
3. *value generation*: both sub-behaviours have constraints on the values of information, and a value that complies to both constraints is established in a non-deterministic fashion. In this way we can abstract from the specific negotiation mechanisms that make the establishment of this value possible.

Consider the following monolithic causality relation:

a_1 *->* a_2 *(*v_2*: Nat) [*$v_2 = 5$*]*

A decomposition of the constraint *[*$v_2 = 5$*]* in which value checking is applied is the following:

a_1 *->* $\underline{a}_2$ *(*v_2*: Nat) [*$v_2 = 5$*] (* 1 *)* *start* -> $\underline{a}_2$ *(v_2: Nat) [v_2 = 5] (* 2 *)*

In causality relation *(* 1 *)*, after a_1 happens, a_2 is allowed to happen with value $v_2 = 5$. In causality relation *(* 2 *)* a_2 may happen at any time, but only with value $v_2 = 5$. The combination of these causality relations is a correct implementation of the monolithic causality relation above.

A decomposition of the constraint *[v_2 = 5]* in which value passing is applied is the following:

a_1 -> $\underline{a}_2$ *(v_2: Nat) [v_2 = 5] (* 1 *)*

start -> $\underline{a}_2$ *(v_2: Nat) (* 2 *)*

In causality relation *(* 1 *)*, after a_1 happens, a_2 is allowed to happen with value $v_2 = 5$. In causality relation *(* 2 *)* a_2 may happen at any time with any value of sort *Nat*. The combination of these causality relations is another correct implementation of the monolithic causality relation above.

Consider the following monolithic causality relation:

a_1 -> a_2 *(v_2: Nat) [$1 \leq v_2 < 5$]*

A possible decomposition of the constraint *[$1 \leq v_2 < 5$]* in which value generation is applied is the following:

a_1 -> $\underline{a}_2$ *(v_2: Nat) [$v_2 \geq 1$] (* 1 *)*

start -> $\underline{a}_2$ *(v_2: Nat) [$v_2 < 5$] (* 2 *)*

In causality relation *(* 1 *)*, after a_1 happens, a_2 is allowed to happen with value *[$v_2 \geq 1$]*. In causality relation *(* 2 *)*, a_2 may happen at any time with value *[$v_2 < 5$]*. The combination of these causality relations is a correct implementation of the monolithic causality relation above, since the value v_2 to be established at a_2 must comply to both constraints.

Figure 5.20 depicts the examples of value checking, value passing and value generation presented above.

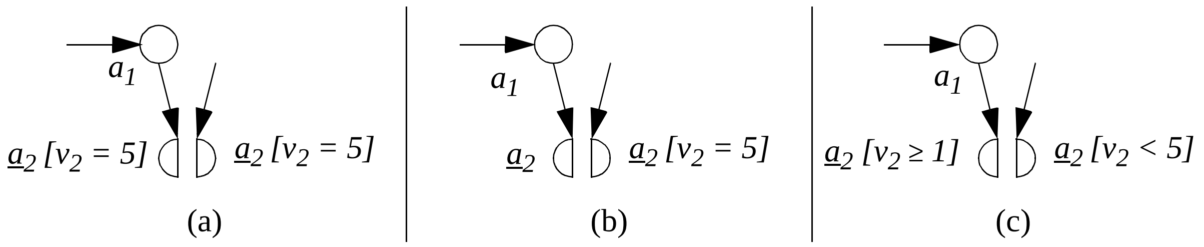

a_1
$\underline{a}_2$ [v_2 = 5]
$\underline{a}_2$ [v_2 = 5]
(a)
a_1
$\underline{a}_2$
$\underline{a}_2$ [v_2 = 5]
(b)
a_1
$\underline{a}_2$ [$v_2 \geq 1$]
$\underline{a}_2$ [$v_2 < 5$]
(c)

Figure 5.20: Three basic forms of value establishment (a) value checking, (b) value passing and (c) value negotiation

Refinement of Distributed Constraints

Given a certain distributed action, further refinements on its value attribute constraints can only be correctly performed if the intended monolithic causality relation is considered. For example, consider the following causality relations for the distributed action a_2:

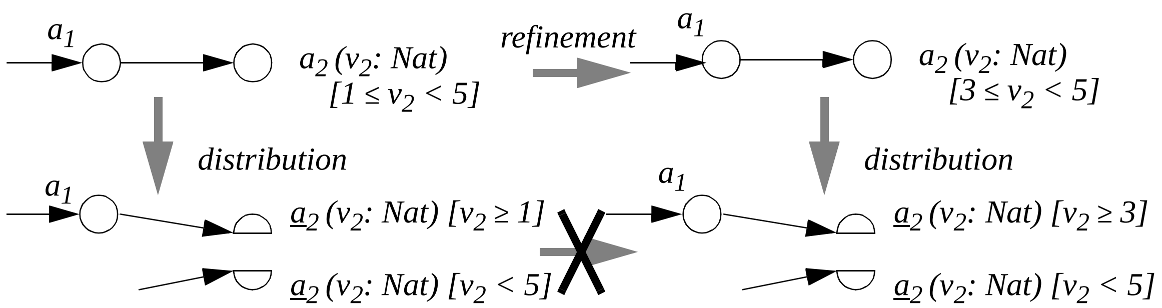
a_1 -> $\underline{a}_2$ *(v_2: Nat) [$v_2 \geq 1$] (* 1 *)*

start -> $\underline{a}_2$ *(v_2: Nat) [$v_2 < 5$] (* 2 *)*

Intuitively we could have decided to restrict the choices of v_2 in *(* 1 *)*, for example. A possible restriction would be to replace $v_2 \geq 1$ by $v_2 \geq 3$, since $v_2 \geq 3$ implies $v_2 \geq 1$. This refinement is only correct because the condition $v_2 < 5$ can still be valid for some v_2. Similarly we could have replaced $v_2 \geq 1$ by $v_2 \geq 7$, but this refinement would have resulted in a deadlock behaviour.

We conclude that restrictions of distributed constraints cannot be considered in isolation. In case a refinement in terms of a restriction of the allowed values in an action has to be defined, we have to initially consider the monolithic causality relation, and then make the selection and the distribution of constraints. In the example above we must consider the monolithic causality relation a_1 -> a_2 *(v_2: Nat) [$1 \leq v_2 < 5$]*, and replace it by a restriction, for example a_1 -> a_2 *(v_2: Nat) [$3 \leq v_2 < 5$]*. This refined monolithic causality relation can then be distributed over sub-behaviours.

Figure 5.21 depicts this refinement procedure.



a_1
a_2 (v_2: Nat) [$1 \leq v_2 < 5$]
refinement
a_1
a_2 (v_2: Nat) [$3 \leq v_2 < 5$]
distribution
distribution
a_1
$\underline{a}_2$ (v_2: Nat) [$v_2 \geq 1$]
$\underline{a}_2$ (v_2: Nat) [$v_2 < 5$]
a_1
$\underline{a}_2$ (v_2: Nat) [$v_2 \geq 3$]
$\underline{a}_2$ (v_2: Nat) [$v_2 < 5$]

Figure 5.21: Refinement procedure for distributed causality relations

5.3.2 Timing Constraints

Similarly to constraints on values of information, constraints on the time attributes of actions can also be distributed over sub-behaviours of a constraint-oriented behaviour composition.

We assume that timing constraints define, for each sub-behaviour, the time moments when an action is allowed to occur. An action actually occurs at a time moment that complies to all the constraints of all sub-behaviours, or it does happen at all, in case the intersection of these constraints is empty. This interpretation of timing constraints allows the definition of behaviour combinations at a high-level of abstraction with respect to the mechanisms that perform actions and their specific timing characteristics, by defining only the requirements which should be fulfilled by these mechanisms.

Consider the following example of monolithic causality relation:

$$a_1\ (t_1\text{: Time}) \text{ -> } a_2\ (t_2\text{: Time})\ [t_1 + 5 < t_2 < t_1 + 10]$$

This causality relation defines that a_2 is only allowed to happen in the interval $]t_1 + 5, t_1 + 10[$, but it actually does not matter when it happens inside this interval. We can decompose this causality relation in the following two causality relations:

$$a_1\ (t_1\text{: Time}) \text{ -> } \underline{a}_2\ (t_2\text{: Time})\ [t_2 > t_1 + 5]\ (*\ 1*)$$

$$a_1\ (t_1\text{: Time}) \text{ -> } \underline{a}_2\ (t_2\text{: Time})\ [t_2 < t_1 + 10]\ (*\ 2\ *)$$

Causality relation *(* 1 *)* imposes that a_2 happens in the interval $]t_1 + 5, \infty[$, while causality relation *(* 2 *)* imposes that a_2 happens in the interval $]t_1, t_1 + 10[$. The combination of both constraints imposes that a_2 happens in the interval $]t_1 + 5, t_1 + 10[$, such that this distribution of constraints yields a correct implementation of the monolithic causality relation above.

Figure 5.22 depicts the timing diagram of the example above.

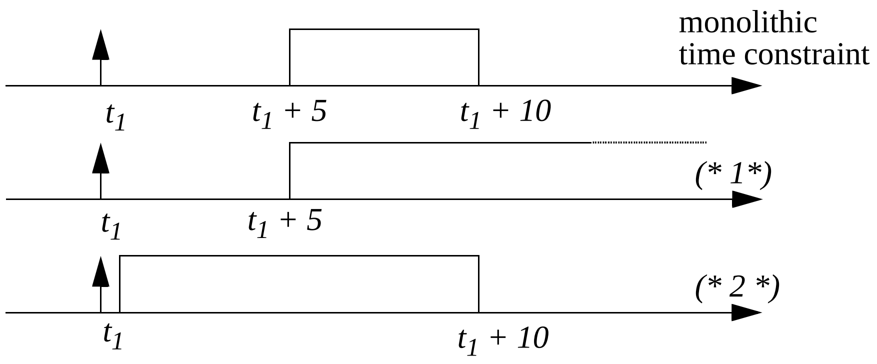


Figure 5.22: Distribution of timing constraints

In this example we abstract from the specific mechanisms that implement the causality relations and their compositions. For example, both sub-behaviours may be wishing to perform the action from the beginning of the periods determined by their constraints, or a third sub-behaviour, to be introduced in later design steps, may collect all constraints and schedule the actions accordingly. The composition of causality relations simply prescribes that the mechanism that implements these constraints is responsible for establishing a_2 at a time moment t_2 that complies to the constraints of both causality relations.

Alternative Interpretation

Another interpretation of timing constraints, which seems to be quite straightforward, is to consider the timing constraints on an action of a certain sub-behaviour as the periods of time in which the sub-behaviour is actively committed to participate in the action. This interpretation has been taken for example in Chapter 4 of [8], which discusses architectural aspects of timing requirements. Furthermore, in [8] it is assumed that in case all sub-behaviours that participate in an action are actively committed to its execution, the action occurs immediately.

Applying this interpretation on the example of Figure 5.22 we conclude that a_2 will always occur at the next measurable time moment after $t_1 + 5$, which is too restrictive for the abstraction level of timing requirements we wish to consider. Executing the action at the first moment of the intersection of the timing constraints is just a particular characteristic of a possible implementation mechanism for these causality relations and should not be considered as a rule at this abstraction level.

In the interpretation of timing constraints assumed in this work we should not be able to identify from the specification when each sub-behaviour is actively committed to perform an action. Even worse, trying to identify these moments only obscures the interpretation of timing constraints.

It appears that this alternative interpretation of timing constraints has been originated from some work on the introduction of timing in process algebras, such as [2], [3], [4], [5], [6], [7] and [10], to name just a few. In most of these models, timing is defined in terms of the time span between actions, periods when behaviours are wishing to participate in actions, and time passage itself, which means that the time concept becomes actually implemented in the process algebra formalism.

In our interpretation of timing constraints only timing requirements for the occurrence of actions are represented. In case we would have an operational model in which actual time passage would be modelled, we could use it to model the mechanisms that implement these causality relations and their composition at a lower abstraction level. These mechanism should satisfy the timing constraints as defined in the causality relations.

5.4 Behaviour Representation

This section discusses the representation of behaviours in terms of a constraint-oriented composition of sub-behaviours, such as the assignment of actions to sub-behaviours and the use of constraint-oriented behaviour composition in combination with causality-oriented behaviour composition.

5.4.1 Action Assignment

Functional entities can only interact if they share interaction points. An approach towards assignment of interactions to the behaviours of the functional entities could be, therefore, to make the functional entities participate in all interactions that occur at the interaction points to which they are attached. Figure 5.23 depicts a generic structure of functional entities that is used to illustrate this approach.

In case the functional entities participate in all interactions at its interaction points this implies that in Figure 5.23 F_1 participates in all interactions at interaction points ip_1 and ip_3, F_2 participates in all interactions at interaction points ip_1 and ip_2, and F_3 participates in all interactions at

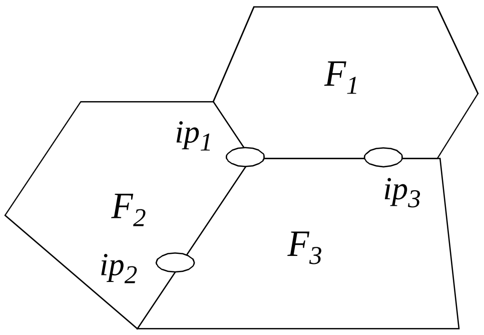


Figure 5.23: Structure of functional entities

interaction points ip_1, ip_2 and ip_3. However, this assignment strategy is too conservative. In order to be able to support more general forms of separation of concerns between the behaviours of the functional entities, we have to allow a functional entity to refrain from participating in some interactions that occurs at one of its interaction points, in case these interactions are not of concern to the functional entity.

Recalling from [9], the interaction between system parts is only possible if these parts share a means of interaction, but it is actually determined by the interaction system formed by these parts, where the semantics and syntax of inter-operability are encapsulated. Applying this reasoning on the interconnection of functional entities, we conclude that interacting functional entities must share interaction points, but one should define beforehand the interactions in which each functional entity participates, such that these interactions are assigned to the behaviour of this functional entity. The combination of the interactions assigned to the behaviours of the functional entities determines the interaction system of these functional entities, since they determine which actions are common to which functional entities.

Figure 5.24 depicts the interaction system between the three functional entities of Figure 5.23, which consists of a proper assignment of actions to the behaviours of these functional entities.

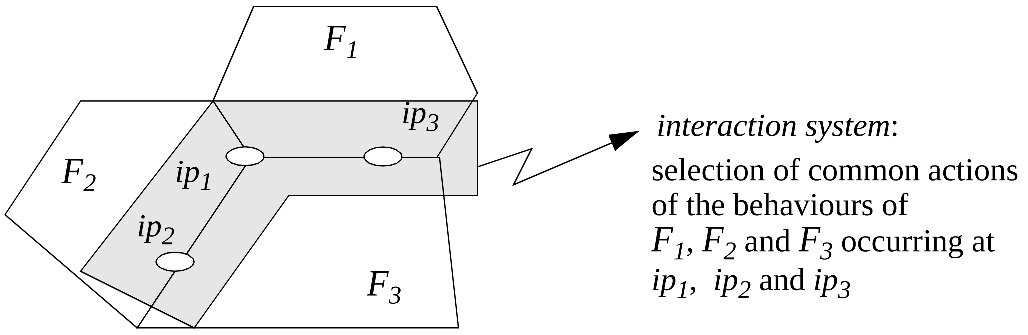

F_1
ip_3
F_2
ip_1
ip_2
F_3
interaction system:
selection of common actions of the behaviours of F_1, F_2 and F_3 occurring at ip_1, ip_2 and ip_3

Figure 5.24: Structure of functional entities and their interaction system

Behaviour structuring should be compatible with functional entity structuring in order to be able to represent it. In this way a behaviour participates in all and only in actions which are known to it. This assumption is rather straightforward, but making it explicit allows us to characterize the common actions of behaviours. This also implies that the characteristics of common actions in which a behaviour participate are the same characteristics that allow behaviours to uniquely identify actions, such that there is no need to define extra mechanisms for action identification at this point.

A strategy for defining constraint-oriented behaviour compositions in a quite general way is to define a set of unique actions of an intended monolithic behaviour, and assign them to the sub-behaviours of the constraint-oriented behaviour composition.

Consider the example of an intended monolithic behaviour *B*, which has actions *a*, *b*, *c*, *d* and *e*. Suppose now we want to represent this behaviour as a composition of behaviours B_1, B_2 and B_3. It is sufficient, at the level of structuring of the monolithic behaviour *B*, to indicate which actions are assigned to which sub-behaviours. Figure 5.25 depicts this behaviour and a specific distribution of actions over behaviours.

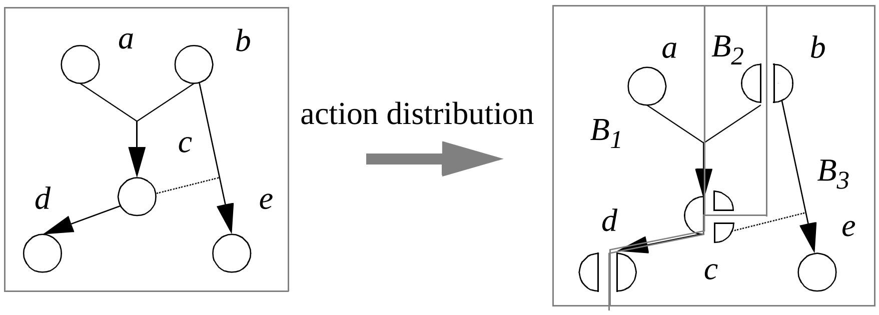


Figure 5.25: Monolithic behaviour and action distribution

In Figure 5.25 we assign *a*, *c* and *d* to B_1, *b* and *c* to B_2 and *b*, *c*, *d* and *e* to B_3. This implies that actions *b*, *c*, *d* are distributed, and that actions *a* and *e* remain monolithic, since they are assigned only to B_1 and B_3 respectively. Particularly *c* is shared by three behaviours, which corresponds to a multi-way synchronization.

Figure 5.26 depicts possible behaviours for B_1, B_2 and B_3 that conform to the possibilities for distribution of conditions discussed in section 5.2.

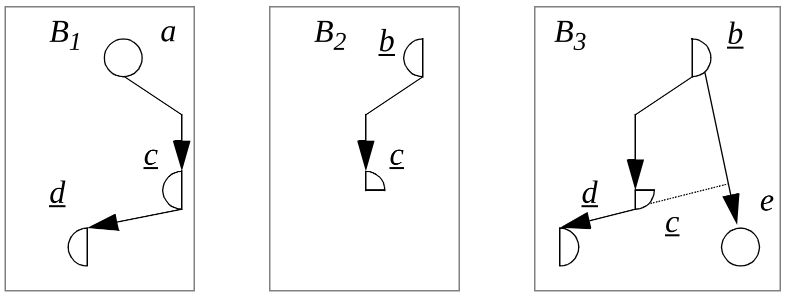


Figure 5.26: Distribution of conditions over sub-behaviours

It may be desirable to define which actions are shared by which sub-behaviours explicitly, although this is not absolutely necessary. Since the same action identifiers are used by all sub-behaviour that define a behaviour, one could determine by inspection which actions are shared by which behaviours. However, the explicit definition of the action assignment to sub-behaviours provides an overview of the global behaviour composition, and can be used to check the consistency of the sub-behaviour definitions.

The behaviour of the example above can be defined in the following way:

$$
\begin{aligned}
&B := \{\ (\ b: B_2, B_3,\ c: B_1, B_2, B_3,\ d: B_1, B_3), \\
&\quad \textit{where} \\
&\quad B_1 := \{\ \textit{start} \rightarrow a,\ a \rightarrow \underline{c},\ \underline{c} \rightarrow \underline{d}\ \} \\
&\quad B_2 := \{\ \textit{start} \rightarrow \underline{b},\ \underline{b} \rightarrow \underline{c}\ \} \\
&\quad B_3 := \{\ \textit{start} \rightarrow \underline{b},\ \underline{b} \rightarrow \underline{c},\ \underline{c} \rightarrow \underline{d},\ \neg\underline{c} \rightarrow e\}\ \}
\end{aligned}
$$

The assignment of actions to sub-behaviours is explicitly defined in the beginning of the behaviour definition of *B*.

Figure 5.27 illustrates the effect of constraint-oriented behaviour composition with four arbitrary behaviours B_1, B_2, B_3 and B_4.

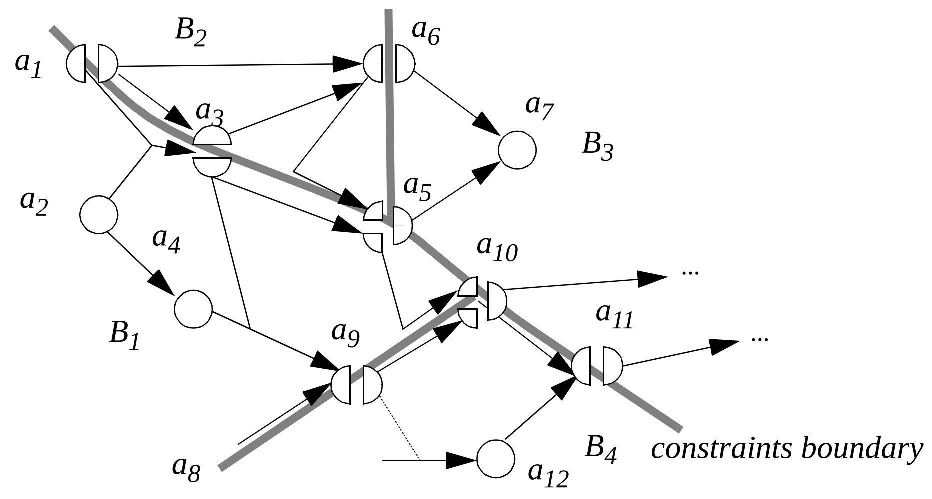


Figure 5.27: Arbitrary constraint-oriented behaviour composition

Composition of Behaviours

An alternative approach to the definition of a behaviour is through composition, where a designer defines sub-behaviours and composes these sub-behaviour to represent some intended monolithic behaviour. This approach may be necessary when a complex intended monolithic behaviour that cannot be represented in a monolithic form has to be expressed. In this case the intended behaviour is defined as a composition of sub-behaviour according to the constraint-oriented behaviour composition, and the monolithic behaviour is the interpretation of the composition.

The rules for interpreting the monolithic behaviour of a composition of sub-behaviours can be inferred from the reversed application of the decompositions addressed in section 5.2. The patterns of composition should be identified in the sub-behaviours of section 5.2, and the corresponding monolithic behaviour can be established.

In some cases the reversed application of the decompositions of section 5.2 may yield monolithic behaviours with impossible actions. Figure 5.28 shows an example.

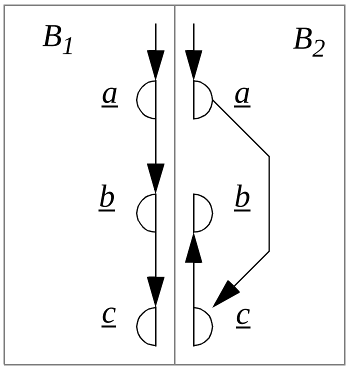


Figure 5.28: Example of composition with impossible actions

In the composition depicted in Figure 5.28 action *a* can happen at any time, but action *b* can only happen if *a* and *c* happen, according to B_1 and B_2, respectively, and action *c* can only happen if *b* and *a* happen according to B_1 and B_2, respectively. Consequently this composition can only execute action *a*, which implies that actions *b* and *c* in principle could be removed in the implementation of this composition. Some research seems to be necessary in order to identify rules for the removal of actions in this case, and to study the complications that may arise when action attributes are considered.

5.4.2 Relationships with Causality-oriented Behaviour Composition

In the causality-oriented behaviour composition, causality relations are decomposed such that conditions and result actions are assigned to separate behaviours. The graphical representation of causality-oriented composition depicts conditions being disconnected from result actions. In the constraint-oriented behaviour composition, conditions and constraints are distributed onto superposed sub-behaviours. The graphical representation of constraint-oriented composition depicts actions being divided into pieces.

Figure 5.29 compares the graphical representations of causality-oriented and constraint-oriented behaviour composition.

Considering a metaphorical definition of these structuring techniques in terms of how one cuts a monolithic behaviour with a knife in order to generate sub-behaviours, we conclude that in the causality-oriented behaviour composition one cuts the causality relations, disconnecting the conditions from the result, while in the constraint-oriented behaviour composition one cuts the actions into pieces through the lines of the causality relations.

Constraint-oriented behaviour composition can be used in combination with causality-oriented behaviour composition in two ways: (i) some constraint-oriented behaviours have entries and exits that are attached to each other or, (ii) some causality-oriented behaviours have actions which they share, forming in this way a constraint-oriented composition. Figure 5.30 shows examples of

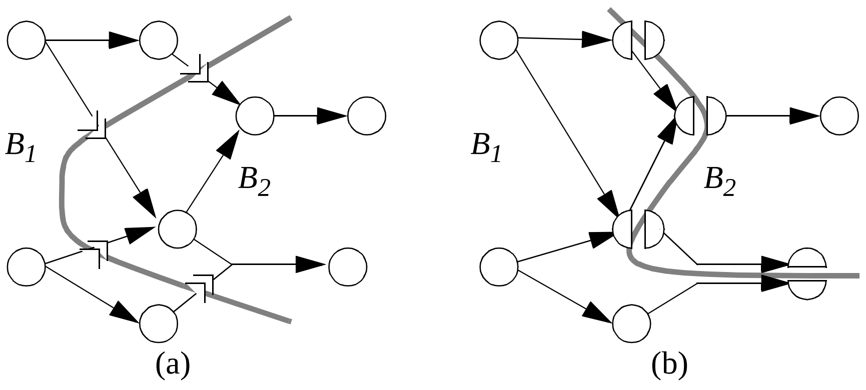


Figure 5.29: Graphical comparison between (a) causality-oriented behaviour composition and (b) constraint-oriented behaviour composition

these two ways causality and constraint-oriented behaviour compositions can be combined, which have the same equivalent monolithic behaviour.

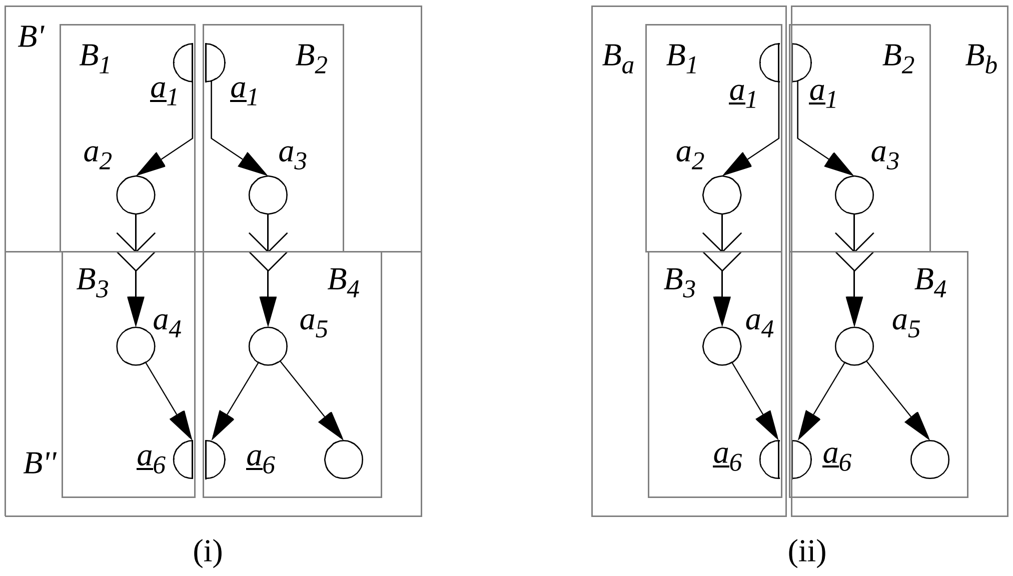


Figure 5.30: Two ways for combining causality and constraint-oriented behaviour composition

In the behaviour of Figure 5.30 (i), behaviour B' is a constraint-oriented composition of behaviours B_1 and B_2 and B'' is a constraint-oriented composition of behaviours B_3 and B_4. These two behaviours are combined using causality-oriented composition, by connecting the exits of B' to the entries of B''. In the behaviour of Figure 5.30 (ii), behaviour B_a is a causality-oriented composition of behaviours B_1 and B_3 and B_b is a causality-oriented composition of behaviours B_2 and B_4. These two behaviour are combined using constraint-oriented behaviour composition, since they share actions a_1 and a_6.

Although these alternatives for combining causality and constraint oriented behaviour composition may result in equivalent monolithic behaviours, they may offer different opportunities for the mapping onto functional entities. Alternative (i) allows functional entities to be composed such that one of the functional entities starts the other, in a sequential manner, while alternative (ii) allows interacting functional entities to be defined. In Figure 5.30 (i) if behaviours *B'* and *B''* are assigned to two functional entities F_1 and F_2, these functional entities do not interact, but the behaviour of F_1 double-enables the behaviour of F_2. In Figure 5.30 (i) if behaviours B_a and B_b are assigned to two functional entities F_a and F_b, these functional entities interact directly, since their behaviour share actions a_1 and a_6.

Repetitive Behaviours

In case a repetitive monolithic behaviour is structured using constraint-oriented behaviour composition, it is possible that the sub-behaviours have to share multiple, possibly infinite, instances of actions. Figure 5.31 illustrates this with a constraint-oriented composition of two infinite behaviours B_1 and B_2..

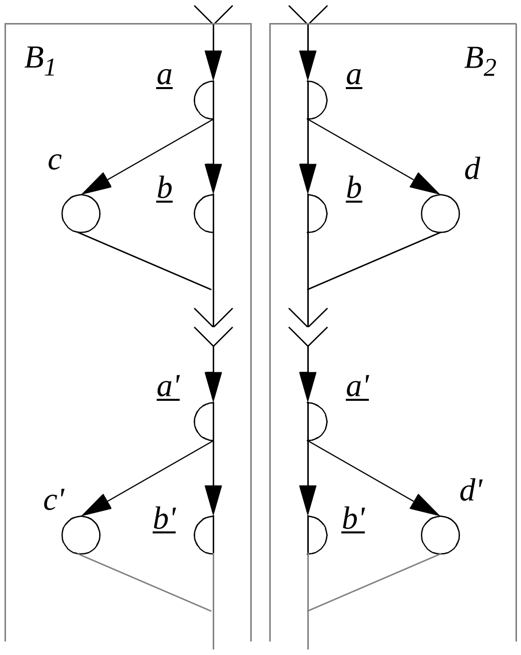


Figure 5.31: Constraint-oriented composition of infinite behaviours

The notation for representing constraint-oriented behaviour composition should be extended in order to support the definition of behaviours that share a possibly infinite set of actions. In Figure 5.31, these actions are the various instances of *a* and *b* that are shared by B_1 and B_2. We indicate this in the following way:

B:= { (a*, b* : B_1, B_2),
        start -> B_1 (entry), start -> B_2 (entry),
where
B_1 := { entry -> $\underline{a}$,
        $\underline{a}$ -> $\underline{b}$, $\underline{a}$ -> c,
        $\underline{a} \wedge c$ -> B_1 (entry) }
B_2 := { entry -> $\underline{a}$,
        $\underline{a}$ -> $\underline{b}$, $\underline{a}$ -> d,
        $\underline{a} \wedge d$ -> B_2 (entry) }

In the behaviour definition above, *a*, b* : B_1, B_2* means that instances of actions *a* and *b* are shared by B_1 and B_2.

Example

We illustrate constraint-oriented behaviour composition of behaviours that share an infinite set of actions with the example of one direction of communication of a connectionless service behaviour, which delivers data without necessarily preserving the order in which data is sent.

This behaviour can be defined, in a simplified form without attribute values, in the following way:

```
CLservice := { (ind*: LocalRcv, Remote)
        start -> Remote (entry),
        start -> LocalRcv (entry),
        where
        Remote := { entry -> req,
                req -> ind,
                req -> Remote (entry) }
        LocalRcv : = { entry -> ind,
                ind -> LocalRcv (entry) } }
```

Figure 5.32 shows this behaviour using the graphical representation.

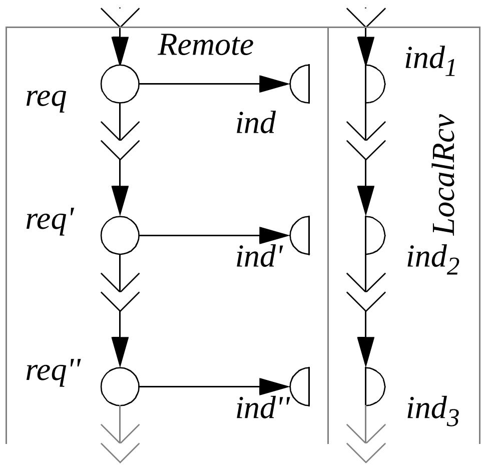


Figure 5.32: One direction of communication of a connectionless service behaviour

In Figure 5.32 in case the *ind* instances of *Remote* coincided one to one with the *ind* instances of *LocalRcv*, we would obtain a behaviour that is similar to the behaviour of the FIFO queue of Figure 4.17. However, behaviour *LocalRcv* participates in *ind* instances, but it is not possible for this behaviour to determine which instance of *ind* it participates in. Therefore we have denoted the *ind* instances in the *Remote* behaviour in a different way than the *ind* instances in the *LocalRcv* in Figure 5.32, such that, for example, ind_1 may correspond to one of *ind*, *ind'*, *ind''*, etc. and the same applies to the other ind_i of *LocalRcv*. Using this interpretation of behaviour *CLService* we actually define that all instances of *ind* are interleaved with each other in behaviour *LocalRcv*. This can considered as a shorthand notation, since the explicit definition of interleaving between all possible instances of *ind* using the behaviour patterns of section 3.4.2 would result in a too complex behaviour definition.

5.5 Expressive Power

This section discusses the expressive power gained by causality relations with the introduction of causality-oriented behaviour composition.

5.5.1 Synchronization

Synchronization is a behaviour pattern in which behaviours share actions, such that an action only occurs if all behaviours that share this action allow the action to occur. Causality-oriented behaviour composition enhances the expressive power of causality relations, by allowing the representation of synchronization between behaviours.

Consider the following behaviour definition:

$$\begin{aligned}&B := \{\ (a_3\!: B_1, B_2),\\ &\qquad start \rightarrow B_1\ (entry),\ start \rightarrow B_2\ \ (entry),\\ &\qquad where\\ &\qquad\ B_1 := \{\ entry \rightarrow a_1,\ a_1 \rightarrow \underline{a}_3,\ \underline{a}_3 \rightarrow a_4\}\\ &\qquad\ B_2 := \{\ entry \rightarrow a_2,\ a_2 \rightarrow \underline{a}_3\ \}\end{aligned}$$

In this behaviour, action a_3 is shared by behaviours B_1 and B_2. Action a_3 only occurs if it is allowed by B_1 and B_2, i.e. after a_1 and a_2 have happened. In case a_3 happens, it happens for both B_1 and B_2, such that B_1 and B_2 synchronize on $\underline{a}_3$.

Independence between behaviours can be considered as a special case of synchronization in which no action is shared between behaviours.

5.5.2 Comparison with LOTOS

In LOTOS, gates represent the means of interaction between processes, and processes participate in the interactions at its gates, according to the rules of the parallel operator. In case we associate the concept of gate with the location attribute of an action, this would imply that LOTOS forces the designer to define behaviours that participate in all actions at certain locations or processes that are interleaved at these locations. The consequence of this modelling decision is that in some complex specifications a set of dummy LOTOS processes have to be defined only to provide synchronization, in order to avoid global deadlocks. We have chosen instead to combine behaviours at an action level, making existence of common locations merely a requirement for consistent assignments of behaviours to functional entities.

In LOTOS one can define that processes choose synchronization in a non-deterministic way. Consider for example the following LOTOS behaviour expression:

```
    P: = P1 [a] || (P2 [a] ||| P3 [a])
```

In this example processes `P2` and `P3` may be both wishing to synchronize in `a` with `P1`, but the actual synchronization occurs in a non-deterministic fashion. The corresponding model in our design model should consider the synchronization between `P1` and `P2`, and between `P1` and `P3` as distinct conflicting actions, in the following way:

$P := \{ (a\colon P_1, P_2, a'\colon P_1, P_3),$
where
$\quad P_1 := \{ \ldots \wedge \neg a' \text{ -> } a, \ldots \wedge \neg a \text{ -> } a', \ldots\}$
$\quad P_2 := \{ \ldots \text{ -> } a, \ldots\}$
$\quad P_2 := \{ \ldots \text{ -> } a', \ldots\} \ldots \}$

LOTOS is incapable of coping with dynamic generation of gates, which seems to be a problem for some specifications. A common artifice to circumvent this problem is to define location information as action parameters. Generalized forms of causality relations are able to cope with dynamic generation of locations, since location attributes in causality relations are just values of a data sort representing the physical and logical locations.

5.6 References

[1] K. Bogaards. *A Methodology for the Architectural Design of Open Distributed Systems*. PhD thesis, University of Twente, Enschede, Netherlands, 1990.

[2] T. Bolognesi and F. Lucidi. LOTOS-like process algebras with urgent or timed interactions. In K. Parker and G. Rose, editors, *FORTE'91. Fourth International Conference on Formal Description Techniques*, 1991.

[3] G. Leduc. An upward compatible timed extension to LOTOS. In K. Parker and G. Rose, editors, *Fourth International Conference on Formal Description Techniques*, 1991.

[4] F. Moller and C. Tofts. A temporal calculus of communicating systems. In J. Baeten and J. Klop, editors, *Proceedings of CONCUR'90. Theories of Concurrency: Unification and Extension*, volume 458 of *Lecture Notes in Computer Science*, pages 401–415, Germany, 1990. Springer-Verlag.

[5] X. Nicollin and J. Sifakis. The algebra of timed processes ATP: Theory and application. Technical Report RT-C26, LGI-IMAG, Grenoble, France, Dec. 1990.

[6] J. Quemada, A. Azcorra, and D. Frutos. A timed calculus for LOTOS. In *FORTE '89. Participant's Proceedings*, 1989.

[7] G. Reed and A. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58:249–261, 1988.

[8] J. Schot. *The role of Architectural Semantics in the formal approach towards Distributed Systems Design*. PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[9] C. A. Vissers, L. Ferreira Pires, and D. A. Quartel. *The Design of Telematic Systems*. University of Twente, Enschede, the Netherlands, Nov. 1993. Lecture Notes.

[10] W. Yi. Real-time behaviour of asynchronous agents. In J. Baeten and J. Klop, editors, *Proceedings of CONCUR'90. Theories of Concurrency: Unification and Extension*, volume 458 of *Lecture Notes in Computer Science*, Germany, 1990. Springer-Verlag.

Chapter 6

Behaviour Refinement

This chapter discusses the *behaviour refinement* design operation, which consists of replacing an abstract behaviour by a more concrete behaviour. Since an abstract behaviour can be replaced by many different alternative more concrete behaviours, and the choice of a specific concrete behaviour is determined by specific design objectives, this design operation can not be automated in its totality. However one can determine the correctness of this design operation by checking whether the concrete behaviour conforms to the abstract behaviour.

This chapter is structured as follows: section 6.1 introduces and defines behaviour refinement and section 6.2 presents a method for determining the abstraction of a concrete behaviour. This method can be applied to assess whether a concrete behaviour conforms to an abstract behaviour. Section 6.3 discusses the role of behaviour refinement in the design process, and section 6.4 illustrates behaviour refinement with some examples.

6.1 Definition

During the design process we may have to replace abstract designs by more concrete designs, in which internal design structure is explicitly defined. Behaviour refinement is defined as a design operation in the behaviour domain in which an abstract behaviour is replaced by a concrete behaviour that conforms to this abstract behaviour. Behaviour refinement allows designers to add internal behaviour structure to an abstract behaviour.

Actions of an abstract behaviour are called *abstract reference actions*. We assume that all abstract reference actions are represented in the concrete behaviour, such that each abstract reference action has a corresponding *concrete reference action* in the concrete behaviour.

By assuming that all abstract reference actions have corresponding concrete reference actions we make it possible to compare the abstract behaviour with the concrete behaviour, in order to assess whether the concrete behaviour conforms to the abstract behaviour. This comparison takes place through the reference actions, which are the reference points in the abstract and concrete behaviours for assessing conformance.

Figure 6.1 depicts an example of behaviour refinement.

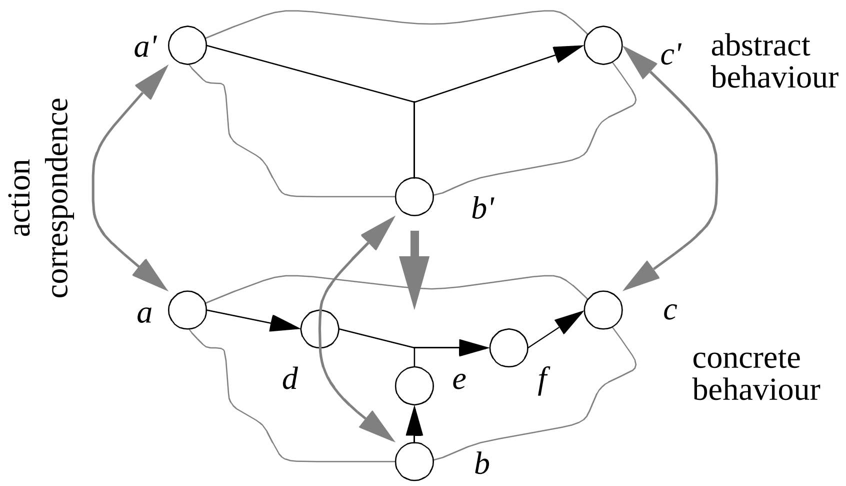


Figure 6.1: Example of behaviour refinement

In Figure 6.1, actions *a'*, *b'* and *c'* are abstract reference actions, and actions *a*, *b* and *c* are their corresponding concrete reference actions, respectively.

All actions of the abstract behaviour are abstract reference actions, and each abstract reference action has a corresponding concrete reference action. Actions of the concrete behaviour that do not have corresponding abstract reference actions have been inserted in the behaviour refinement, and characterize the behaviour modifications performed in this design operation. In Figure 6.1, for example, the inserted actions *d*, *e* and *f* of the concrete behaviour do not have corresponding actions in the abstract behaviour, and have been inserted in this design operation.

We assume that the following activities have to be performed in an instance of behaviour refinement:

1. delimitation of abstract behaviour;
2. elaboration of concrete behaviour;
3. determination of the abstraction of the concrete behaviour;
4. comparison between the abstraction of the concrete behaviour and the original abstract behaviour.

Some of these activities are briefly discussed in the sequel. Section 6.2 defines the rules that allow one to determine the abstraction of a concrete behaviour.

6.1.1 Behaviour Delimitation

The abstract behaviours that we consider for refinement must be delimited by their abstract reference actions. We do not consider the refinement of behaviours which have actions that are not abstract reference actions but can influence the occurrence of the abstract reference actions.

In general, it may be more difficult to refine a complex monolithic behaviour than to refine subbehaviours of this monolithic behaviour and compose their refinements. Therefore we should be able to select sub-behaviours of a monolithic behaviour, refine them separately, and compose these refinements in order to obtain the refinement of the monolithic behaviour. This can be useful, for example, in the case of a repetitive and infinite behaviour, since an infinite number of abstract reference actions and their relationships can only be refined if we can refine finite sub-behaviours of this infinite behaviour separately and compose their refinements.

Figure 6.2 depicts an example in which two sub-behaviours of a monolithic behaviour are selected, such that these two sub-behaviours can be refined separately.

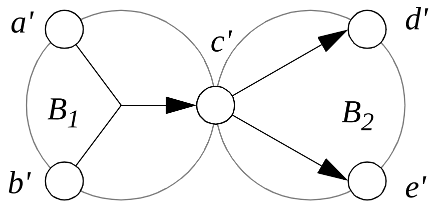


Figure 6.2: Delimitation of abstract behaviours

In Figure 6.2 we show a monolithic behaviour consisting of actions *a'*, *b'*, *c'*, *d'* and *e'*. We assume that two sub-behaviours of this single behaviour should be selected, such that they can be refined separately: sub-behaviour B_1, delimited by actions *a'*, *b'* and *c'* and sub-behaviour B_2, delimited by actions *c'*, *d'* and *e'*. In this way the relationships between *a'*, *b'* and *c'* are refined in the refinement of B_1, and the relationships between *c'*, *d'* and *e'* are refined in the refinement of B_2.

The selection of behaviours for refinement must comply to the requirement that all actions of an abstract behaviour are abstract reference actions. This imposes certain restrictions on the behaviours that can be selected. Figure 6.3 shows an example that illustrates these restrictions.

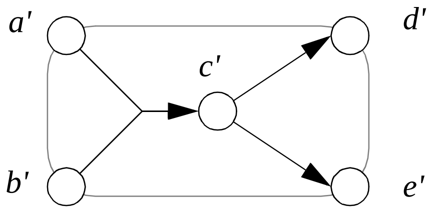


Figure 6.3: Invalid abstract behaviour for behaviour refinement

In Figure 6.3, action *c'* is a condition for actions *d'* and *e'*, and it is caused by *a'* and *b'*. In case we decide to select *a'*, *b'*, *d'* and *e'* for refinement, we would have to do something about *c'*. Action *c'* cannot be simply abandoned, since it is responsible for the relationships between *a'* and *b'*, and *d'* and *e'*. Action *c'* cannot be considered as an action that is not an abstract reference action either, otherwise we cannot assess the conformance of concrete behaviours to this abstract behaviour. In order to avoid these problems we impose that such forms of behaviour are invalid abstract behaviours for behaviour refinement.

Concrete behaviours are also delimited by their concrete reference actions. Actions in a concrete behaviour should be either concrete reference actions or inserted actions, such that we would either consider or abstract from them, respectively, when assessing the conformance of a refinement.

Figure 6.4 depicts an example of a concrete behaviour that is not delimited by its concrete reference actions. In Figure 6.4 there are conditions of *d* and *e* that refer to actions outside the concrete behaviour that refines the relations between *a*, *b* and *c*, such that this behaviour is not delimited by *a*, *b* and *c*, and therefore can not be considered as a proper refinement of an abstract behaviour.

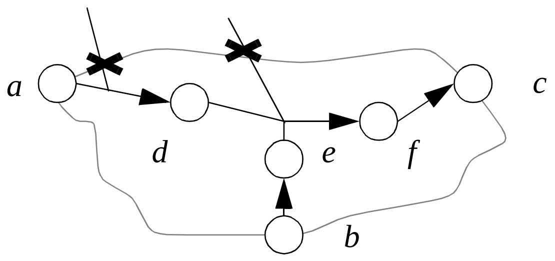


Figure 6.4: Invalid concrete behaviour

6.1.2 Conformance Requirements

An instance of behaviour refinement is considered to be correctly performed if the concrete behaviour *conforms to* the abstract behaviour. Intuitively one can characterize conformance between a concrete and an abstract behaviour by two requirements:

1. *preservation of causality and exclusion*: causality and exclusion relationships between abstract reference actions defined in the abstract behaviour are preserved in the concrete behaviour by their corresponding concrete reference actions;
2. *preservation of attribute values*: attribute values of the concrete reference actions are the same as the attribute values of their corresponding abstract reference actions.

The two correctness requirements above imply, for example, that in Figure 6.1 the inserted actions and the causality relations inside the closed line linking concrete reference actions should conform to the causality relations inside the closed line linking abstract reference actions. These conformance requirements define a class of possible concrete behaviours for an abstract behaviour, such that designers can select one of these concrete behaviours based on specific circumstances of the design process, such as characteristics and availability of components, improvement of reliability, etc., getting a step closer to an implementation.

In essence behaviour refinement allows one to introduce (internal) structure in an abstract behaviour. This structure is not defined in the abstract behaviour, since the definition of the abstract behaviour should still allow many alternative behaviour structures to be defined in concrete behaviours that conform to this abstract behaviour.

Preservation of Causality and Exclusion

Causality and exclusion relationships between the abstract reference actions that are defined in an abstract behaviour should be preserved by their corresponding concrete reference actions. However, since the causality relations of the abstract reference actions are modified in the concrete behaviour, these causality and exclusion relationships should be maintained by the inserted actions and the causality relations of the concrete behaviour.

In Figure 6.1, for example, action *c'* in the abstract behaviour can only happen if both actions *a'* and *b'* have happened. We do not consider conditions and constraints on attribute values for the time being. In the concrete behaviour, action *c* should only be allowed to happen if both actions *a* and *b* happen. Indeed, action *c* can only happen if action *f* happens, and *f* can only happen if both *e* and *d* happen, and since *d* can only happen if *a* happens, and *e* can only happen if *b* happens, we conclude that *c* can only happen if *a* and *b* happen. Therefore the causality relationships between actions *a'*, *b'* and *c'* are preserved by *a*, *b* and *c*, respectively, in the concrete behaviour.

The causality relation of an abstract reference action and the causality relation of its corresponding concrete reference actions may be substantially different. In Figure 6.1, for example, the causality relation of *c'* in the abstract behaviour is $a' \wedge b' \text{ -> } c'$, but in the concrete behaviour the causality relation of *c* is $f \text{ -> } c$.

Preservation of Attribute Values

Another condition for a concrete reference action to conform to an abstract reference action is that it preserves the attribute values of the abstract reference action. A concrete reference action should not establish attribute values that are not established by its corresponding abstract reference action. In this way attribute values that are possible for a concrete reference action should also be possible for its corresponding abstract reference action.

Yet one could consider two alternatives for the preservation of attribute values:

1. *strong preservation*: all attribute values that are possible for an abstract reference action are also possible for its corresponding concrete reference action;
2. *weak preservation*: there may be attribute values that are possible for an abstract reference action but are not possible for its corresponding concrete reference action.

The difference between strong and weak preservation of attribute values is rather subtle, and is illustrated with simple examples depicted in Figure 6.5.

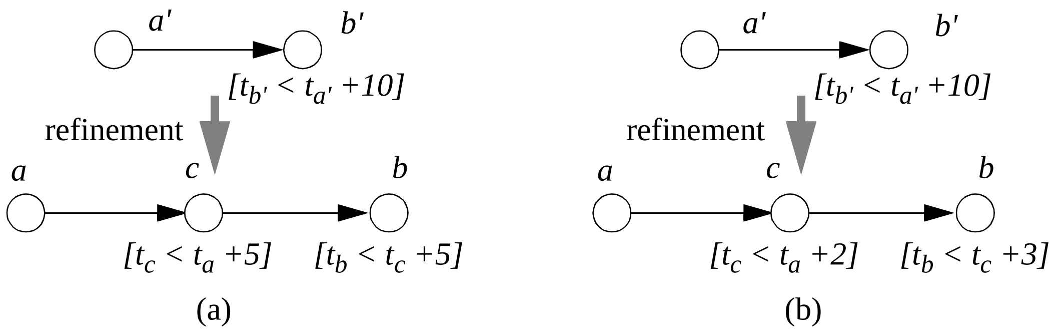


Figure 6.5: Examples of preservation of attribute values
(a) strong preservation; (b) weak preservation

In Figure 6.5 (a) all time attribute values defined for *b'* in the abstract behaviour are still possible for *b* in the concrete behaviour. Action *c* happens before $t_a + 5$ and action *b* happens before $t_c + 5$, implying that *b* happens before $t_a + 10$, such that the possibilities of the original behaviour have not been reduced. For example, it is always possible that *b* happens at $t_a + 7$ in this refinement.

In Figure 6.5 (b) not all time attribute values defined for *b'* in the abstract behaviour are possible for *b* in the concrete behaviour. If we substitute the time constraints of *c* in *b* we conclude that *b* does not happen after $t_a + 5$. However, in case *b* happens before $t_a + 5$ implicitly *b* happens before $t_a +10$, which means that this refinement could be considered correct. In this refinement, time attribute values that are possible in the original behaviour have been reduced. For example, it is not possible that *b* happens at $t_a +7$ in this refinement.

We handle weak preservation of attribute values in this work by considering that before applying behaviour refinement the options of behaviour for the original behaviour may be reduced, such that during behaviour refinement only strong preservation applies. In this way we would first modify the condition $t_{b'} < t_{a'} +10$ to $t_{b'} < t_{a'} +5$ in the example of Figure 6.5 (b), and then impose strong preservation of attribute values as a conformance requirement.

Strong preservation and weak preservation could be compared to the concepts of equivalence of systems and observational compatibility presented in [3], respectively. Furthermore strong preservation actually implies weak preservation, such that if strong preservation holds for two behaviours then weak preservation also holds.

In the examples above, direct references to attribute values of the reference actions in the abstract behaviour become indirect references through inserted actions in the concrete behaviour. For example in Figure 6.5 (a), action *b'* refers to the time attribute $t_{a'}$ of action *a'* directly, but in the concrete behaviour action *b* refers directly to t_c and indirectly to t_a, namely through a specific relation between t_a and t_c. Changes in attribute references is an important characteristic of behaviour refinement.

6.1.3 Role of Abstraction

Conformance between behaviours can be assessed if we know under which circumstances a certain behaviour can be considered as an abstraction of another more concrete behaviour. Once we have identified which abstract behaviour corresponds to which concrete behaviours we can apply this correspondence inversely in the design process, by selecting a concrete behaviour that conforms to an abstract behaviour.

The approach towards determining which behaviour is an abstraction of a concrete behaviour discussed in this work starts by considering that concrete reference actions are coupled by their causality relations, defining specific causality, exclusion and attribute value relationships between these actions. Abstracting from actions that are inserted during behaviour refinement we can determine which behaviour is an abstraction of the remaining actions and their relationships.

Figure 6.6 illustrates the role of abstraction in behaviour refinement.

In Figure 6.6, actions *a*, *b, c*, *d*, *e* and *f* are coupled by their causality relations in the concrete behaviour. Abstracting from actions *d*, *e* and *f* we can determine which behaviour is an abstraction of the actions *a*, *b* and *c* and their causality relationships. In this abstract behaviour, actions *a'*, *b'* and *c'* represent actions *a, b* and *c,* respectively. Actions *a*, *b* and *c* are coupled through actions *d*, *e* and *f* in the concrete behaviour, while actions *a'*, *b'* and *c'* are expected to be directly related in the abstract behaviour. Once we know which abstract behaviour corresponds to the concrete

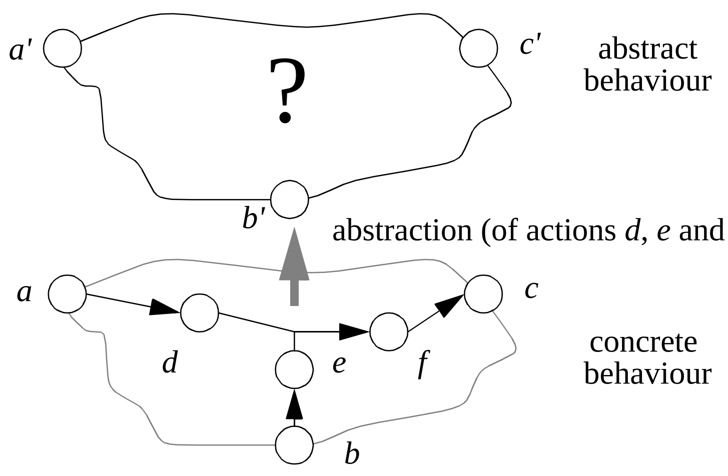


Figure 6.6: Arbitrary behaviour and its abstraction

behaviour in Figure 6.6 we can use this knowledge to assess the correctness of an instance of behaviour refinement that starts with this abstract behaviour and results in this concrete behaviour.

6.2 Abstraction Rules

In behaviour refinement, causality relations between abstract reference actions are replaced by causality relations involving their corresponding concrete reference actions and some inserted actions. References to action attributes of abstract reference actions are replaced by references to the attributes of concrete reference actions and inserted actions. There are in principle many different alternative concrete behaviours that conform to a certain abstract behaviour, generated by the different alternatives for the insertion of actions, causality relations and references to attributes. This implies that a concrete behaviour cannot be automatically generated nor deduced from the abstract behaviour, since when formulating a refinement a designer introduces new design information, which follows from specific design choices.

However, given a concrete behaviour and its concrete reference actions, one should be able to deduce the corresponding abstract behaviour, by abstracting from the inserted actions and their influence on the concrete behaviour.

The following steps define a method to deduce the abstract behaviour of a certain given concrete behaviour:

1. abstract from references to inserted actions and their attribute values that appear in the conditions of other actions of the concrete behaviour;
2. (possibly) simplify the causality relations obtained, e.g. by replacing terms such as $a_i \wedge a_i$ and $a_i \vee a_i$ by a_i;
3. go to step 1 again, unless a behaviour without inserted actions has already been obtained.

When we abstract from inserted actions in step 1 we obtain a more abstract behaviour with respect to the initial behaviour of this step. The application of this method on a concrete behaviour results in a behaviour involving only abstract reference actions. Rules for abstracting from references to inserted actions and their attribute values are discussed in this section. These rules are called *abstraction rules*.

6.2.1 Behaviours with Causality

Consider a concrete behaviour defined in terms of causality relations involving only causality (occurrence of actions) and without conditions and constraints on action attributes. In this case, since we do not allow circular definitions in causality relations, an action will never be found in its conditions nor in the conditions of its condition actions. Furthermore causality conditions are transitive, which makes it possible to devise an abstraction rule in which inserted actions are replaced by their conditions.

We discuss two situations below: (i) an inserted action we abstract from is a necessary condition of an action and (ii) an inserted actions we abstract from is a sufficient condition of an action.

Inserted Action as Necessary Condition

In case an inserted action is a necessary condition of an action, the conditions of the inserted action are also necessary conditions for the action, if only causality is used. For example, suppose action *d* is an inserted action in the behaviour *B:= { start -> a, start -> b, a -> d, d* $\wedge$ *b -> c }*. Action *d* is a necessary condition for *c*, but since *a* is a necessary condition of *d*, action *a* is also a necessary condition for *c*. This implies that *c* cannot happen if *a* does not happen.

We conclude that in general one can abstract from an inserted action that is a necessary condition in a behaviour, by replacing it by its conditions. Figure 6.7 shows that an abstract behaviour can be obtained from behaviour *B* above, by abstracting from action *d*.

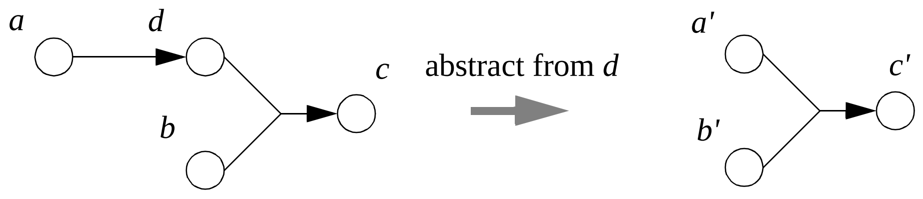


Figure 6.7: Abstraction from inserted action as necessary condition

When we abstract from action *d* in Figure 6.7 we obtain a more abstract behaviour, in terms of the more abstract actions *a'*, *b'* and *c'*.

Inserted Actions as Sufficient Conditions

In case an inserted action is a sufficient condition for an action, the conditions of the inserted action are also necessary conditions for the sufficient condition of the action, if only causality is used. For example, suppose *d* is an inserted action in the behaviour *B:= {start -> a, start -> b, a -> d, d* $\vee$ *b -> c}*. Action *d* is a sufficient condition for *c*, but since *a* is a necessary condition of *d*, action *a* is also a necessary condition of a sufficient condition of *c*. This implies that *c* cannot happen caused by *d* if *a* does not happen.

We conclude that in general one can also abstract from an inserted action that is a sufficient condition in a behaviour, by replacing it by its conditions. Figure 6.8 shows that an abstract behaviour can be obtained from behaviour *B* above, by abstracting from action *d*.

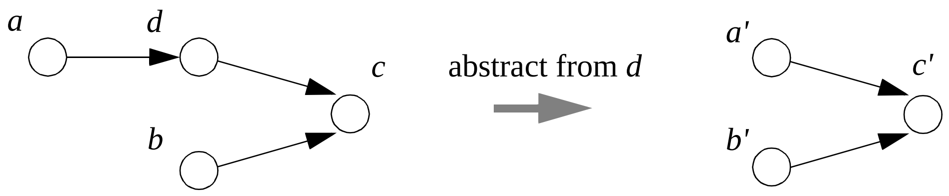


Figure 6.8: Abstraction from inserted action as sufficient condition

General Abstraction Rule

We conclude, from the observations above, that the following general rule can be defined for abstracting from inserted actions in the deduction of the abstract behaviour (step 1), in the case of a concrete behaviour defined only in terms of causality:

> *Abstraction Rule 1:*
> an inserted action that is a causality condition for an action of the concrete behaviour can be replaced by the condition of the inserted action as defined in its causality relation.

Examples

Consider behaviour *B:= { start -> a, a -> c, c -> b }*, where actions *a* and *b* are concrete reference actions. Using the method for determining the abstract behaviour and abstraction rule 1, we abstract from *c* by replacing it by *a* in the causality relation of *b* and obtain the abstract behaviour *B':= { start -> a', a' -> b' }*, where *a'* and *b'* are abstractions of *a* and *b* respectively.

Consider behaviour *B:= { start -> a, start -> b, a -> d, b -> e, d* ∧ *e -> c }*, where *a, b* and *c* are concrete reference actions. We can abstract from *d* and *e* in the causality relation of *c*, and we conclude that *B':= { start -> a', start -> b', a'* ∧ *b' -> c' }* is the abstract behaviour of *B*.

Consider behaviour *B:= { start -> a, a -> c, a -> d, c* ∧ *d -> b }*, where *a* and *b* are concrete reference actions. In this case we substitute *c* ∧ *d* in the condition of *b* by *a* ∧ *a*, which reduces to *a* following step 2 of the method given above, and we conclude that *B':= { start -> a', a' -> b'}* is the abstract behaviour of *B*.

Figure 6.9 depicts the examples discussed above.

Some examples of abstraction from sufficient conditions are presented below.

Consider behaviour *B := { start -> a, start -> b, a -> d, b -> e, d* ∨ *e -> c }*, where actions *a, b* and *c* are concrete reference actions. Applying abstraction rule 1 to abstract from *d* and *e* we conclude that *B':= { start -> a', start -> b', a'* ∨ *b' -> c' }* is the abstraction of *B*.

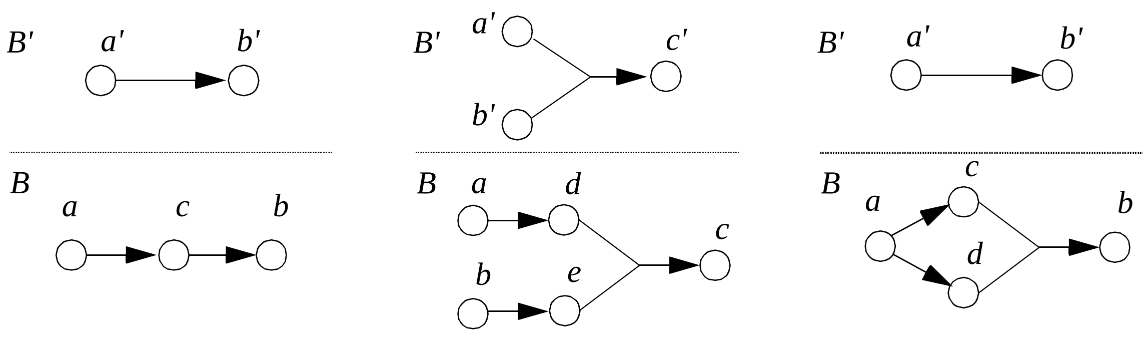


Figure 6.9: Some behaviours with causality and their abstractions

Consider behaviour *B:= { start -> a, a -> c, a -> d, c ∨ d -> b}*, where *a* and *b* are concrete reference actions. In this case we can replace *c ∨ d* in the condition of *b* by *a ∨ a*, which reduces to *a* following step 2 of the method, and we conclude that *B':= { start -> a', a' -> b'}* is the abstraction of *B*.

Figure 6.10 depicts these behaviours and their corresponding abstractions.

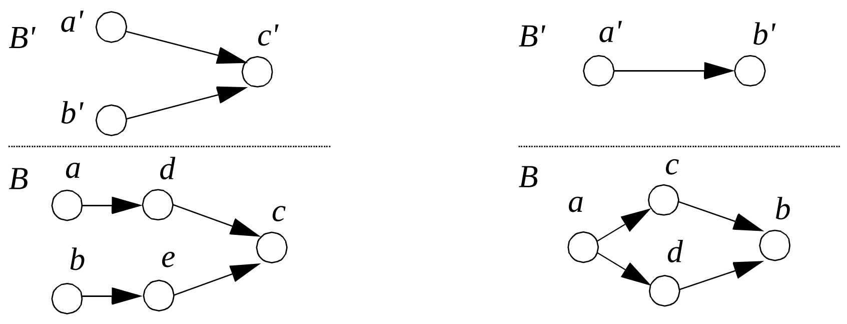


Figure 6.10: Abstraction from actions as sufficient conditions

6.2.2 Behaviours with Exclusion

Unlike causality conditions, in general replacing an action in an exclusion condition by its conditions does not yield correct abstractions. An example is behaviour *B:= { start -> a, start ∧ ¬ c -> b, a -> c}*, where *a* and *b* are concrete reference actions. The application of abstraction rule 1 on this behaviour would result in the behaviour *B' := { start -> a', start ∧ ¬ a' -> b'}*, but this behaviour is surely no abstraction of *B*. For example, in behaviour *B* action *b* may happen after *a*, while *b'* can not happen after *a'* in *B'*.

Considering all possible action sequences and dependencies that result from behaviour *B* above we conclude that the correct abstraction of this behaviour is the one in which *a'* and *b'* are independent of each other, but there is a probability that *b'* does not happen, which is the consequence of *c* happening before *b* has happened in the concrete behaviour *B*.

Since abstraction rule 1 can not be generally applied to behaviours with exclusion conditions, in the sequel we investigate the situations in which abstraction rule 1 can be applied, and develop new rules for those situations in which abstraction rule 1 does not apply.

No Exclusion Involving Inserted Actions

The application of abstraction rule 1 on exclusion conditions of inserted actions may yield incorrect abstractions. However, in case the replacement of inserted actions by their conditions according to the abstraction rule 1 does not involve exclusion conditions we still can obtain correct abstractions, even if the behaviour contains exclusion conditions. This more general condition for the application of abstraction rule 1 implies that the non-occurrence of inserted actions is not used in any causality relation, and that no exclusion condition appears in the causality relation of the inserted actions.

Figure 6.11 depicts two examples in which abstraction rule 1 can be safely applied.

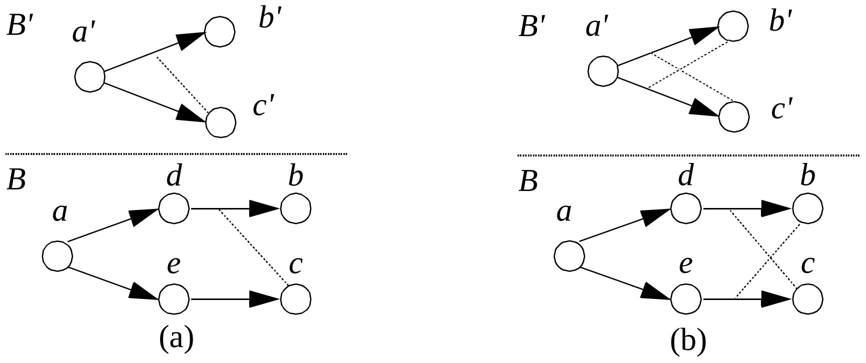


Figure 6.11: Behaviours with exclusion and their abstractions

In both examples of Figure 6.11 exclusion is used only in the causality relations of concrete reference actions, and appears only in terms of the non-occurrence of concrete reference actions, such that only occurrences of inserted actions are replaced according to abstraction rule 1. In both cases the abstract behaviours can be obtained by applying abstraction rule 1.

The extra condition on abstraction rule 1 to make it applicable to these cases is that the inserted action being replaced is in causality form (occurrence) and that the conditions of this inserted action only contain causality and no exclusion.

Making Uncertainty Explicit

In some cases the concrete behaviour explicitly defines the conditions in which an abstract reference action may or may not happen. Figure 6.12 depicts two examples in which the uncertainty on the occurrence of an abstract action is made explicit in the concrete behaviour.

The abstraction of the concrete behaviour of Figure 6.12 (a) should represent that if *c* in the concrete behaviour happens before *d*, *d* is not allowed to happen, and in this case *b* does not happen. Similarly, the abstraction of the concrete behaviour of Figure 6.12 (b) should represent that if *e* in the concrete behaviour happens before *d*, *d* is not allowed to happen, and in this case *b* does not

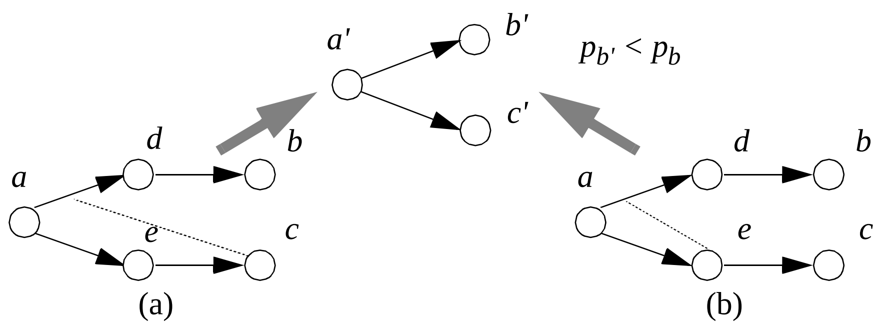


Figure 6.12: Two behaviours with exclusion and their common abstraction

happen. In the abstract behaviour we abstract from *d* and *e*, such that in the abstract behaviour we can only state that *b'* may not happen after *a'*. The specific situations in which *b'* happens or does not happen can not be explicitly represented in the abstract behaviour.

The probability attribute defines a conditional probability, since it corresponds to the probability that an action occurs given that its conditions are satisfied. In general the probability attribute value of $p_{b'}$ is smaller than the probability attribute value of p_b, since the probability that *b* happens after that *d* happens is bigger than the probability that *b* happens after that *a* happens, due to the possibility that *d* does not happen after *a*. Therefore the possibility that *b'* does not happen after *a'* due to the inserted actions should be implicitly represented in the probability attribute of *b'*. The concrete behaviours actually make the uncertainty on the occurrence of *b'* explicit, by defining the internal behaviour that determines this uncertainty.

The observations above can be used to define the following rule to replace an inserted action in the deduction of the abstract behaviour (step 1), in case this inserted action has a necessary exclusion condition:

> *Abstraction Rule 2:*
> considering an inserted action that is a necessary condition for an action, say *a*, a necessary exclusion condition of this inserted action can be discarded when the inserted action is replaced by its conditions. The probability attribute value of the abstraction of action *a* in the resulting abstract behaviour decreases with respect to the probability attribute value of *a*. Exceptions are the situations involving action conflict discussed further on.

Considering the concrete behaviour of Figure 6.12 (a), we apply abstraction rule 2 by replacing *d* by $a \wedge \neg c$ in the causality relation of *b* and discarding the condition $\neg c$, which results in the causality relation *a'* -> *b'*, where *b'* has a smaller probability attribute value than *b*. A similar reasoning applies to the concrete behaviour of Figure 6.12 (b). Figure 6.12 also shows that abstraction rule 2 applies to an exclusion by a reference action, such as in Figure 6.12 (a), or by an inserted action, such as in Figure 6.12 (b).

Inversion of Causality

The effect of the exclusion of an action by an inserted action may imply that the causality relation of another action has to be used in an inverted form in the deduction of the abstract behaviour. Figure 6.13 depicts an example meant to illustrate this inversion of causality.

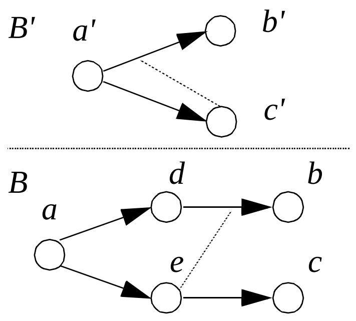


Figure 6.13: Example of inversion of causality

In Figure 6.13, the application of abstraction rule 1 to abstract from action *e* yields the causality relation *a'* -> *c'* for *c'*. The causality relation of *b* is originally *d* ∧ ¬*e* -> *b*, such that *d* can be replaced by its condition *a*, but we have to be careful about the replacement of ¬*e*. We cannot simply apply abstraction rule 2 and remove the non-occurrence of *e* from the causality relation of *b*, because this would result in two independent actions *b'* and *c'*, which is an incorrect abstraction of the concrete behaviour. Furthermore *e* is an exclusion condition of the reference action *b*, which does not comply to the conditions of abstraction rule 2. Therefore we have to define a specific rule for the abstraction of the non-occurrence of an inserted action as a necessary condition of an action in the deduction of the abstract behaviour.

Inverting the causality relation of *c* we can conclude that, in case *e* does not happen, *c* does not happen either, which means that the non-occurrence of *e* implies the non-occurrence of *c*. The non-occurrence of *e* can be replaced by the non-occurrence of *c* in this case, resulting in the abstract behaviour of Figure 6.13. Notice that a necessary condition for this abstraction to be correct is that we can abstract from possible references in *c* to attribute values of *e*.

We recall from the definition of causality and exclusion that in executions of the concrete behaviour *B* in Figure 6.13 action *c* can only possibly happen after *e*, and that, if both *b* and *e* happen, *e* happens after *b*. This implies that *b* will never happen after *c* if both *b* and *c* happen, while *c* may happen at any time after *b* happens. Abstracting from action *e* we can deduce that the abstract behaviour obtained by inverting the original causality relation of *c* is indeed an abstraction of the concrete behaviour.

The following rule can be defined to replace an exclusion condition of an inserted action in the causality relation of an action in the deduction of the abstract behaviour (step 1):

> *Abstraction Rule 3:*
> exclusion by an inserted action in the causality relation of an action, say *a*, can be replaced by exclusion by another action, say *b*, where *b* is caused by the inserted action. This abstraction is only acceptable if we can abstract from possible references in *b* to attribute values of the inserted action.

Actions in Conflict

When an inserted action to be removed forms a choice with another action we get a situation in which the application of abstraction rules 2 and 3 results in an incorrect abstraction. Figure 6.14 illustrates this situation with two examples of concrete behaviour.

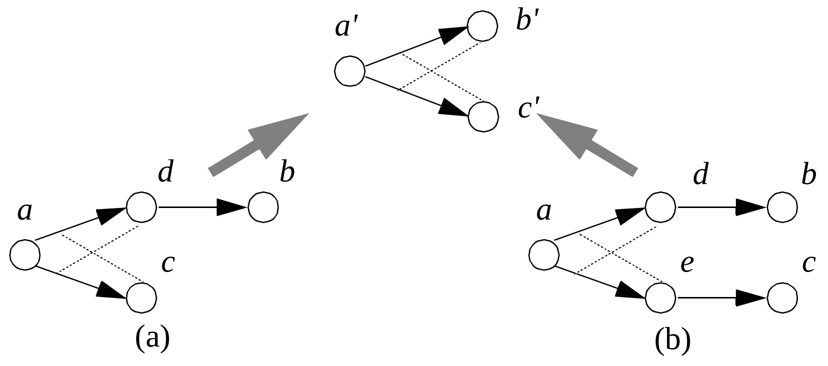


Figure 6.14: Two behaviours with actions in conflict and their common abstraction

In the concrete behaviour of Figure 6.14 (a) we can apply the abstraction rules given before to substitute *d* by its condition and to discard the condition ¬*c*, resulting in the causality relation *a'* -> *b'*, where *b'* has a smaller probability attribute than *b*. We can also apply abstraction rule 3, and substitute the condition ¬*d* in *c* by ¬*b*, resulting in the causality relation *a'* ∧ ¬ *b'* -> *c'*. However, the abstract behaviour obtained in this way does not represent the concrete behaviour properly, since it is not possible that *b* and *c* both happen in an execution of the concrete behaviour, while *b'* and *c'* can both happen in an execution of the abstract behaviour. Similar reasoning applies to concrete behaviour of Figure 6.14 (b).

Therefore we cannot abstract from actions in conflict using the rules given so far. The following abstraction rule can be defined to abstract from inserted actions in case of action conflicts:

> *Abstraction Rule 4*:
> considering that an inserted action to be removed is a necessary condition of an action, say *a*, the conflict between another action, say *b*, and the inserted action is inherited by actions *a* and *b*.

Limitations

The abstraction rules defined in this section do not handle exclusion as a sufficient condition. Since this use of exclusion does not appear very often in usual behaviour patterns, we leave the definition of abstraction rules to handle these cases for further study.

6.2.3 Attribute Values

In the rules defined so far we have not considered abstraction from references to attribute values of inserted actions. Since references to attribute values are only possible when causality conditions are used, it is possible to define rules that are applied in combination with abstraction rule 1 above, such that the influence of the inserted actions on the attribute values of the abstract behaviour can be taken into account. Such rules are defined for each attribute type in the sequel.

Action Values

The following rule should be applied to abstract from references to action values of inserted actions in combination with abstraction rule 1:

> *Abstraction Rule 1a:*
> references to action values of the inserted actions being removed are substituted by their values or constraints.

We illustrate the application of this rule on the following concrete behaviour:

$$\begin{aligned} B := \{\ & \mathit{start} \rightarrow a, \\ & a\ (v_a\!: \mathit{Nat}) \rightarrow c\ (v_c\!: \mathit{Nat})\ [v_c = v_a^2], \\ & c\ (v_c\!:\mathit{Nat}) \rightarrow b\ (v_b\!: \mathit{Nat})\ [v_b = v_c + 10]\ \} \end{aligned}$$

Suppose *a* and *b* are reference actions, we replace the condition of *b,* which is $c\ (v_c\!:\mathit{Nat})$, by the conditions of *c*. Applying rule 1a we replace the reference to v_c in the constraint of v_b by the constraint on v_c, which is v_a^2. This results in the following abstract behaviour:

$$B' := \{\ \mathit{start} \rightarrow a',\ a'\ (v_{a'}\!:\mathit{Nat}) \rightarrow b'\ (v_{b'}\!: \mathit{Nat})\ [v_{b'} = v_{a'}^2 + 10]\ \}$$

From the refinement point of view, i.e. considering that we want to find a conforming concrete behaviour for an abstract one, we conclude that under the condition that the function that determines the values of the reference actions can be decomposed, which is the case in this example, we can refine this behaviour by assigning parts of the computation of the function to the different causality relations. In the example the calculation of v_a^2 is assigned to the relation between *a* and *c*, and the addition of *10* is assigned to the relation between *c* and *b*.

Timing Requirements

Similarly to action values, we may also have to abstract from references to time attribute values of inserted actions when we deduce the abstraction of a concrete behaviour. The following rule should be applied to abstract from references to time attribute values of inserted actions, in combination with abstraction rule 1:

> *Abstraction Rule 1b:*
> references to time attribute values of inserted actions being removed are substituted by their values or constraints.

In order to determine the specific ways in which timing constraints can be replaced, we systematically discuss the various options for this replacement by considering a behaviour *B:=* *{ start -> a, a -> c [T_c (t_a, t_c)], c -> b [T_b (t_c, t_b)] }*, and its abstract behaviour *B':= { start -> a', a' -> b' [$T_{b'}$ ($t_{a'}$, $t_{b'}$)] }.*

Figure 6.15 shows the generic example used to systematically discuss the replacement of timing constraints.

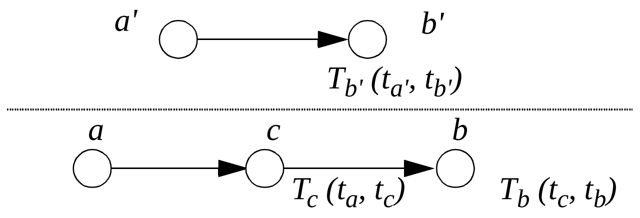


Figure 6.15: Timing requirements and implicit time constraints

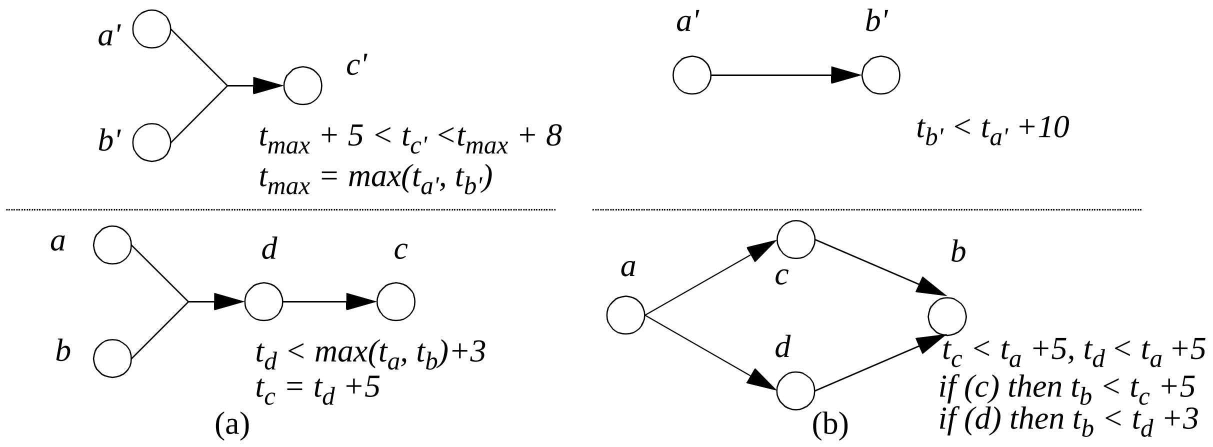
Some specific forms of T_c and T_b and their corresponding $T_{b'}$ are indicated in Table 6.1.

Table 6.1: Some timing constraints and their abstractions

T_c	T_b	$T_{b'}$
$t_c < t_a + 2$	$t_b < t_c + 4$	$t_{b'} < t_{a'} + 6$
$t_c < t_a + 2$ $t_c > t_a$	$t_b = t_c + 4$	$t_{a'} + 4 < t_{b'} < t_{a'} + 6$
$t_c < t_a + 2$ $t_c > t_a$	$t_b > t_c + 4$	$t_{b'} > t_{a'} + 4$
$t_c > t_a + 2$	$t_b > t_c + 4$	$t_{b'} > t_{a'} + 6$
$t_c > t_a + 2$	$t_b = t_c + 4$	$t_{b'} > t_{a'} + 6$
$t_c > t_a + 2$	$t_b < t_c + 4$	$t_{b'} > t_{a'} + 2$
$t_c = t_a + 2$	$t_b = t_c + 4$	$t_{b'} = t_{a'} + 6$
$t_c = t_a + 2$	$t_b < t_c + 4$ $t_b > t_c$	$t_{a'} + 2 < t_{b'} < t_{a'} +6$
$t_c = t_a + 2$	$t_b > t_c + 4$	$t_{b'} > t_{a'} + 6$

Table 6.1 shows that in some cases one has to consider the implicit time constraint of causality in order to deduce the abstract timing constraints. For example in case T_c is $t_c < t_a$ +2 and T_b is $t_b = t_c + 4$, we could simply remove t_c in T_b by combining these conditions, and we get $t_{b'} < t_{a'} + 6$. However if the occurrence of *c* has a maximum delay of *2* with respect to *a*, and the occurrence of *b* has a fixed delay of *4* with respect to *c*, it implies that *b'* has a minimum delay of *4* with respect to *a'*. This condition can only be correctly deduced if we consider the implicit time constraint $t_c > t_a$ and combine it with T_b, which results in the constraint $t_{a'} + 4 < t_{b'} < t_{a'} +6$.

Figure 6.16 depicts two examples of concrete behaviours with timing requirements, and their corresponding abstractions.



a'
b'
c'
$t_{max} + 5 < t_{c'} < t_{max} + 8$
$t_{max} = max(t_{a'}, t_{b'})$
a'
b'
$t_{b'} < t_{a'} + 10$
a
b
d
c
$t_d < max(t_a, t_b)+3$
$t_c = t_d + 5$
(a)
a
c
d
b
$t_c < t_a +5, t_d < t_a +5$
if (c) then $t_b < t_c +5$
if (d) then $t_b < t_d +3$
(b)

Figure 6.16: Concrete behaviours with timing requirements

In Figure 6.16 (a), when applying abstraction rule 1 to abstract from *d* in the causality relation of *c* we also apply abstraction rule 1b, such that the constraint of t_d is substituted in the explicit and implicit timing constraint of t_c. This results in the constraint $max\ (t_{a'}, t_{b'}) + 5 < t_{c'} < max\ (t_{a'}, t_{b'}) + 8$. In Figure 6.16 (b), when applying abstraction rule 1 to abstract from *c* and *d* we also apply abstraction rule 1b, resulting, after simplification, in the causality relation *a' ($t_{a'}$: Time) ->* *b' ($t_{a'}$: Time) [$t_b < t_a +10 \vee t_b < t_a +8$]*. Since $t_b < t_a +8$ implies $t_b < t_a +10$, this time constraint can be simplified to $t_b < t_a +10$.

Probability Requirements

Abstraction rules for probability requirements are discussed considering two specific situations: (i) the inserted action being replaced is a necessary condition for an action, and (ii) the inserted action being replaced is a sufficient condition for an action.

For inserted actions that are a necessary condition for an action one can simply apply the following rule:

> *Abstraction Rule 1c:*
> for an inserted actions being replaced that is a necessary condition of an action, say *a*, we calculate the probability that both the inserted action and action *a* occur, and replace the result in the probability attribute of the abstraction of action *a*.

Figure 6.17 depicts a simple example in which the inserted action to be replaced is the only necessary condition of the reference action.

In Figure 6.17 the probability attribute of *b'* has to be deduced from its conditions, namely the occurrence of *a'*. This is done by considering that the probability that *b* happens given that *a* happens is the probability that both *b* and *c* happen, which is then the probability attribute of *b'*. Applying rule 1c above we conclude that $p_{b'} = p_c . p_b = 60\%$.

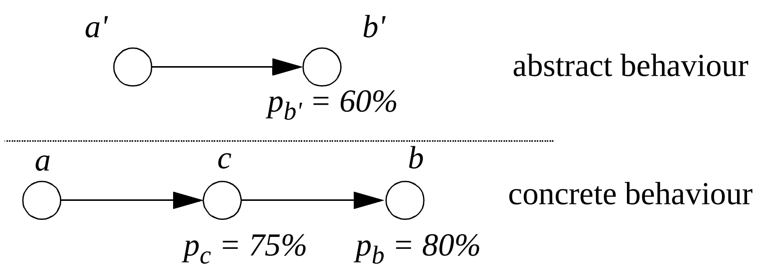


Figure 6.17: Simple abstraction of probability requirements

From the point of view of refinement, i.e. considering that we want to obtain a concrete behaviour from an abstract behaviour, we could have interpreted the probability requirements of the abstract reference actions in two distinct ways: (i) as the desired probability or (ii) as the worst case probability.

In Figure 6.17, the probability that b' happens given that a' has happened is the same as the probability that b happens given that a has happened. However, in case the probability p_c could be increased to say *80%*, the resulting probability that action b happens given that action a has happened would have been increased to *64%*. In this case we could either reject the refinement, following interpretation (i), or accept it, following interpretation (ii). Applying interpretation (ii) we actually consider the probability attribute $p_{b'}$ to be a variable that satisfies the constraint $p_{b'} \geq 60\%$, referring in this way not to a single behaviour, but to a class of behaviours, where in each of these behaviours $p_{b'}$ has a value that is bigger or equal to *60%*. We consider only alternative (i) in this text.

Figure 6.18 depicts another example in which we abstract from actions that are necessary conditions for other actions, and we calculate the probability attributes of the remaining actions.

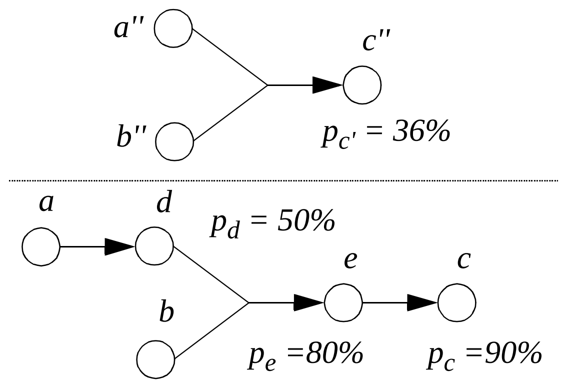


Figure 6.18: Abstraction of probability requirements in necessary conditions

In Figure 6.18 we apply abstraction rules 1 and 1c twice to consider the influence of the inserted actions e and d. When action e is replaced by its condition we calculate the temporary probability $p_{c|d \wedge b}$= 72%, which is probability that c happens given that d and b happen. When action d is replaced by its condition we finally determine the probability attribute of c'', which gets the value $p_d \cdot p_{c|d \wedge b} = 36\%$.

It appears that the abstraction rule for the probability attribute of an action that is a sufficient condition of another action cannot be given in a single rule as we have done for the case of a necessary condition. Figure 6.19 illustrates this problem.

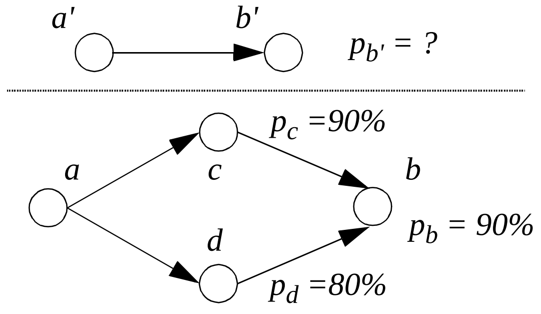


Figure 6.19: Abstraction of probability requirements in necessary conditions

In case we would try to abstract from action *c* in Figure 6.19 before abstracting from action *d*, we would have to calculate the portion of the probability that action *b* happens caused by *c* separately from the probability that action *b* happens caused by *d*. We suppose instead that the probability attribute of *b* is a single value $p_b = 90\%$, which represents the probability that *b* happens in case its conditions are satisfied. In this case we are forced to abstract from *c* and *d* at the same time.

A naive approach to calculate the probability attribute of abstract action *b'* in Figure 6.19 is to calculate the probability that the conditions for *b* are true, which is the probability that only *c*, only *d* or both happen, and multiply these probabilities by the probability attribute of *b*. However the probability that the conditions for *b* are true calculated in this way also include the situations in which *c* or *d* occur after *b*, in which case they cannot influence the occurrence of *b*.

We conclude that the probability attribute $p_{b'}$ can only be properly calculated if we know the *probability density function*[1] of the time attribute of *c*, *d* and *b*. In this case we could calculate the probabilities that (i) *b* happens caused by *c* given that *d* does not happen before *b*, (ii) *b* happens caused by *d* given that *c* does not happen before *b* and (iii) *b* happens when both *c* and *d* happen before *b*, by integrating the density functions over the intervals in which *b* is allowed to happen according to its timing constraints. The sum of these three probabilities should result in the probability attribute value $p_{b'}$. A more specific treatment of such examples is for further study.

Some research is necessary to define more general rules and techniques for abstracting from probability attribute values. Some proposals for representing probability in formal semantics, such as in [1], for example, apply probabilities to the alternative behaviours of the choice operator in order to assign specific weights to behaviour choices. An interesting research question is the relationship between probabilities in choice and our definition of probability attribute.

1. The probability density function in this case defines the relationship between the time attribute values and the probability attribute values. Suppose for action *c* the time constraint is $[t_c \leq t_a + 10]$, we could consider t_c as a random variable with probability function e.g. $f(t) = 0$ for $t > t_a + 10$ and $f(t) = p_c/10$ (uniform distribution). In this case $f(t)\,\mathrm{d}t$ can be seen as the probability that t_c gets a value in the interval $[t, t + \mathrm{d}t]$ in case *a* has happened, and the total probability that *c* happens given that *a* has happened could be calculated by integrating $f(t)\,\mathrm{d}t$ from t_a to $t_a + 10$.

Generalized Application of the Rules

Abstraction rules 1a, 1b and 1c can be applied simultaneously and in combination with abstraction rule 1, in case references to action values, time attribute values of inserted actions have to be replaced, and the influence of probability requirements of inserted actions in the abstract behaviour has to be considered.

Figure 6.20 depicts an example in which some of these rules are simultaneously applied.

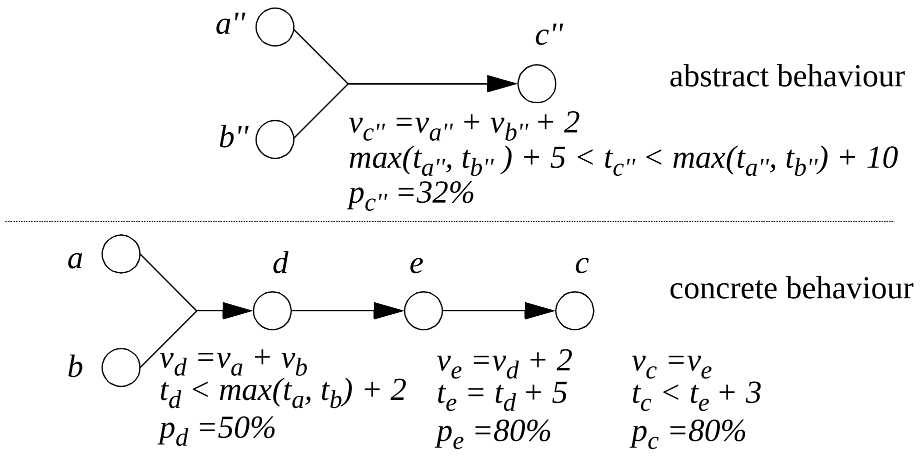


Figure 6.20: Generalized behaviour refinement example

We summarize the steps for deducing the abstract behaviour below:

Step 1:

- *Rule 1*: abstract from *e*, replacing it by *d* (its conditions);
- *Rule 1a*: replace v_e by $v_d + 2$ in the constraint of v_c;
- *Rule 1b*: replace $t_c < t_e + 3$ by $t_d + 5 < t_c < t_d + 8$ in the constraint of t_c;
- *Rule 1c*: replace probability attribute value by $p_{c|d} = 64\%$;

Step 2 (void, no simplification): behaviour in terms of *a'*, *b'*, *c'* and *d'*;

Step 3 (new instance of *Step 1*):

- *Rule 1*: remove *d'*, replacing it by $a' \wedge b'$;
- *Rule 1a*: replace $v_{d'}$ by $v_{a'} + v_{b'}$ in the constraint of $v_{c'}$;
- *Rule 1b*: replace $t_{c'} > t_{d'} + 5$ by $t_{c'} > max(t_{a'}, t_{b'}) + 5$, and $t_{c'} < t_{d'} + 8$ by $t_{c'} < max(t_{a'}, t_{b'}) + 10$;
- *Rule 1c*: replace probability attribute value by $p_{c'|a' \wedge b'} = 32\%$.

Step 2 (void, no simplification): behaviour in terms of *a''*, *b''* and *c''*.

After applying the abstraction rules we have abstracted from all inserted actions and taken their influence in the abstract behaviour into consideration. Figure 6.20 also depicts the abstract behaviour obtained using the method and the abstraction rules. Notice that the method imposes no specific order for abstracting from inserted actions. In case we had started by abstracting from action *d* and then from action *e* we would have obtained the same abstract behaviour.

6.3 Role in the Design Process

Considering the design milestones presented in Chapter 2 we observe that the integrated system perspective has to be refined to a distributed system perspective in order to be implemented. This process may be performed iteratively, such that a final distributed system perspective can be obtained after repetitive application of this refinement.

In order to obtain a distributed system perspective from an integrated system perspective one has to search for cooperating functional entities that can replace the functional entity represented in the integrated system perspective. This design step is often called *functionality decomposition* in the literature ([5], [6], [4]). In this thesis we call this design step (*functional*) *entity decomposition*, in order to be consistent with the terminology used so far.

Figure 6.21 illustrates entity decomposition, considering that an arbitrary functional entity F is replaced by the cooperating functional entities F_1, F_2, F_3 and F_4.

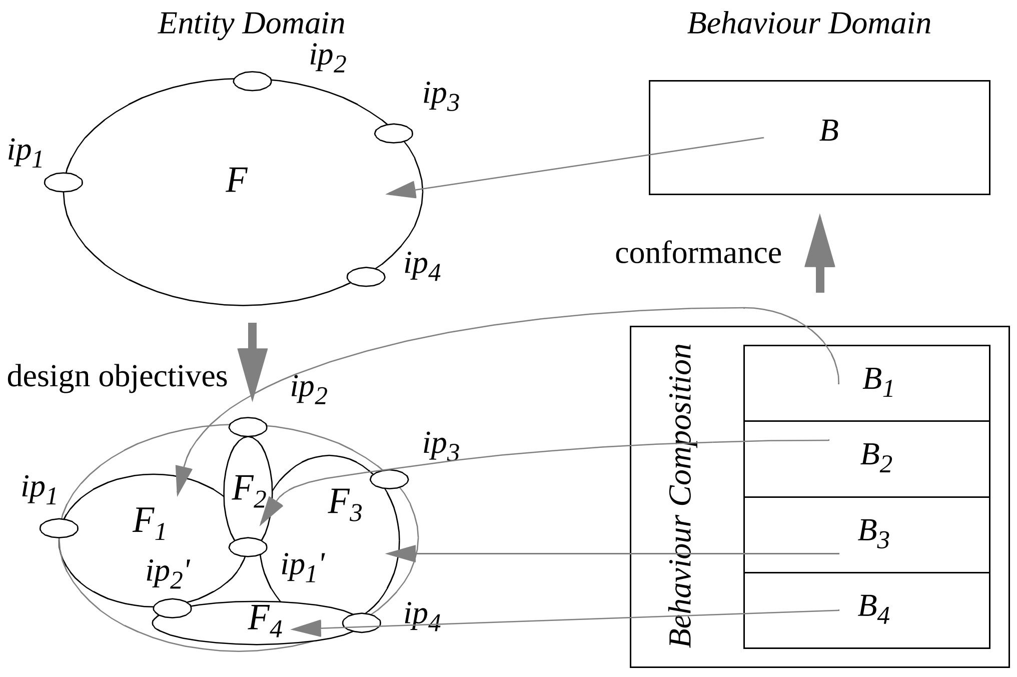


Figure 6.21: Functional entity decomposition

6.3.1 Entity Domain

The design objective of entity decomposition is defined in terms of elements of the entity domain: entity decomposition consists of replacing a functional entity by a corresponding composition of functional entities.

Functional entities are delimited by interaction points. This implies that in order to define a composition of functional entities from a single functional entity we have to insert interaction points in this single functional entity, allowing the functional entities of the composition to be delimited. Insertion of interaction points is a necessary manipulation to achieve entity decomposition.

In Figure 6.21, for example, interaction points ip_1' and ip_2' are inserted in the original functional entity F, such that functional entities F_1, F_2, F_3 and F_4 are delimited by $\{ip_1, ip_1', ip_2'\}$, $\{ip_2, ip_1'\}$, $\{ip_3, ip_4, ip_1'\}$ and $\{ip_4, ip_2'\}$, respectively.

Although we presented entity decomposition by referring to the example of an integrated system and a distributed system, entity decomposition can actually be generalized by considering that the functional entity to be decomposed may have action points or interaction points or both, and that the resulting composition of functional entities may also have action points or interaction points or both.

We postulate that the following conditions hold in the entity domain:

- interaction points of the original functional entity must be preserved in the decomposition;
- action points of the original functional entity may be either preserved or transformed in interaction points in the decomposition;
- more action or interaction points may be inserted in the decomposition.

6.3.2 Behaviour Domain

The behaviour of a functional entity is defined in terms of actions and interactions occurring at the action points and interaction points of the functional entity, respectively, and the relationships between these actions and interactions, defined by the causality relations of these actions and interactions. We denote actions and interactions indistinctly by the term action in the sequel.

Two actions are directly related with each other if one is mentioned in the causality relation of the other. According to the consistency conditions between the entity and behaviour domains discussed in section 2.1.1, actions can only be directly related in a behaviour if the action points where these actions occur belong to a single functional entity.

When entity decomposition is performed, the behaviour of the original functional entity has to be decomposed in sub-behaviours, such that these sub-behaviours are assigned to the resulting functional entities. Action points that belong to the original functional entity may be assigned to different functional entities in the resulting design. In this case actions that are directly related in the behaviour of the original functional entity cannot be directly related in the behaviours of the resulting functional entities, but have to be indirectly related by interactions occurring at interaction points shared by these functional entities.

In Figure 6.21 we suppose, for example, that there are interactions at ip_1 and ip_2 which are directly related in the behaviour of F. In the resulting design these interactions have to be indirectly related through interactions occurring at ip_1', i.e. through the behaviours of F_1 and F_2.

We recall that an interaction can be considered as an action if we abstract from the specific responsibilities of the functional entities involved in this interaction. Therefore one can consider the interactions at interaction points inserted during the entity decomposition design step as if they were actions inserted in the behaviour of the original functional entity, before responsibilities on the execution of these interactions are assigned to the resulting functional entities.

We conclude that in some instances of entity decomposition, namely when actions that were directly related in the behaviour of the original functional entity are assigned to different functional entities in the resulting design, the designer has to perform behaviour refinement before defining constraints and assigning behaviours to the resulting functional entities.

Summarizing, the following design operations play a role in entity decomposition:

1. (possible) behaviour refinement of the original functional entity, according to the objectives of the design step. The final configuration of functional entities must be taken into consideration;
2. definition of separate constraints on the execution of actions of the concrete behaviour, defining in this way sub-behaviours;
3. assignment of these sub-behaviours to the resulting functional entities. The total behaviour obtained is the composition of the behaviours of the functional entities.

This chapter discusses design operation (1). Design operation (2) can be performed by applying constraint-oriented behaviour composition, which is discussed in Chapter 5 and design operation (3) is rather trivial. Behaviour refinement plays an important role in the design process, although some instances of entity decomposition can be performed without behaviour refinement. The actions to be inserted in an instance of behaviour refinement and their precise role in the resulting design, in case behaviour refinement supports entity decomposition, should be defined from specific objectives of the design step, such as for example the structure of functional entities to be obtained. The resulting behaviour structure should be able to express the resulting composition of functional entities.

6.3.3 Example

Figure 6.22 illustrates the design operations necessary for entity decomposition with a simple design example.

In Figure 6.22 functional entity F has two action points ap_a and ap_b, where actions a' and b' happen, respectively. Behaviour B', consisting of a' and b', is assigned to F. The design step aims at decomposing F into F_1 and F_2, such that ap_a belongs to F_1 and ap_b belongs to F_2.

Since a' and b' are originally directed related in the behaviour of F, it is necessary to insert an interaction point ip_c to be shared by F_1 and F_2. In the behaviour domain we replace actions a' and b' by actions a and b, respectively, inserting action c between them, which makes it possible for action b to indirectly refer to action a through c. Behaviour B formed by a, b and c should conform to the original behaviour B'. Action c represents the interaction that happens at ip_c, abstracting from the participation of F_1 and F_2. Since our final objective is to define the behaviours of F_1 and F_2 and their composition, constraints on action c have to be identified. In this way behaviour B is structured in sub-behaviours B_1 and B_2, which are then assigned to functional entities F_1 and F_2, respectively.

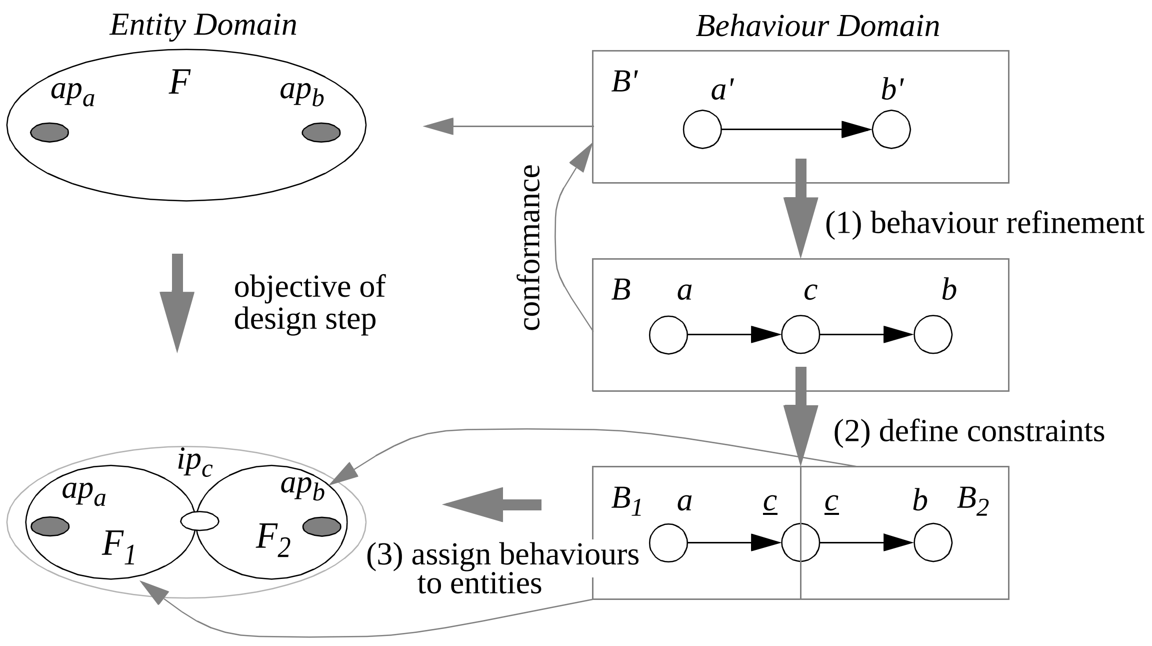


Figure 6.22: Design operations for entity decomposition

6.4 Application Examples

This section illustrates behaviour refinement and its application in the design process with a couple of examples. In some of the examples the method for deducing the abstract behaviour for a concrete behaviour defined before can be directly applied to check the correctness of the refinement, while in other examples the application of the method is not possible.

6.4.1 Action Choice

Consider that we have an abstract behaviour which consists of a choice between two actions *a'* and *b'*. Applying behaviour refinement on this behaviour corresponds to selecting a concrete behaviour, defined in terms of actions *a* and *b* that correspond to *a'* and *b'*, respectively, and some inserted actions. This concrete behaviour should conform to the abstract behaviour. We also suppose that actions *a* and *b* should not be directly related in the concrete behaviour.

Inverting abstraction rule 4, we conclude that the concrete behaviour can be defined in terms of (i) a necessary condition of *b* in conflict with *a*, or vice-versa, or (ii) conflicting necessary conditions of *a* and *b*. We also suppose that *a* and *b* will turn into interactions in a subsequent design operation in which part of this behaviour is assigned to a system and another part to the system’s environment. According to alternative (ii), when we turn *a* and *b* into interactions the system will always determine the choice alone, while the environment is forced to accept whatever the system determines. We choose to concentrate on alternative (i), because according to this alternative when *a* and *b* turn into interactions the environment would still be able to influence the choice.

Figure 6.23 depicts the initial refinement of action choice according to alternative (i).

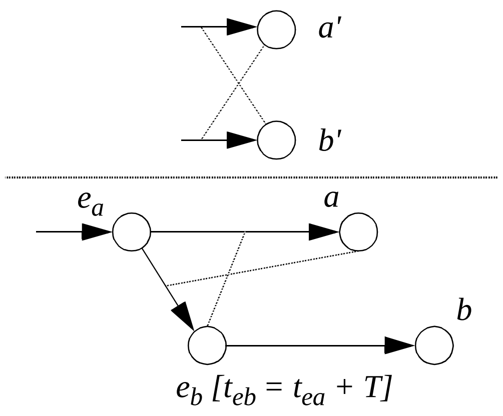
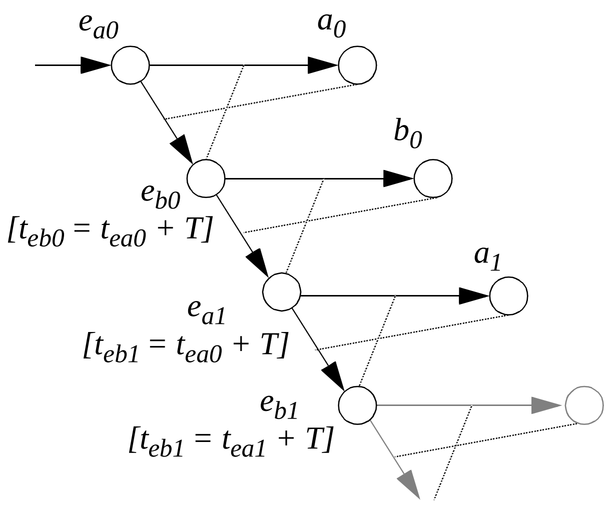


Figure 6.23: Initial refinement of action choice

In Figure 6.23 we suppose that some action e_a enables the occurrence of action *a*, such that after e_a happens action *a* may happen. In case action *a* does not happen *T* units of time after e_a, action e_b happens, enabling the occurrence of *b*.

Although the concrete behaviour of Figure 6.23 can be considered as an implementation of the abstract behaviour, it has the drawback that the chances for *a* and *b* to happen are unfair with respect to the time periods in which *a* and *b* are allowed to happen, since although *a* is allowed to happen first, it should happen within a period of *T* time units, while *b* is allowed to happen for an infinite time period in case *a* does not happen. In order to make a fair distribution of time periods, action *b* should be disabled at some time, such that action *a* is enabled again. This behaviour pattern should repeat itself, until either *a* or *b* happen. Figure 6.23 shows this concrete behaviour.



e_{a0}
a_0
b_0
e_{b0}
$[t_{eb0} = t_{ea0} + T]$
a_1
e_{a1}
$[t_{eb1} = t_{ea0} + T]$
e_{b1}
$[t_{eb1} = t_{ea1} + T]$

Figure 6.24: Fair concrete behaviour for an action choice

For the sake of clarity we have represented actions *a* and *b* in Figure 6.23 by the set of actions a_0, a_1,... and b_0, b_1,..., respectively, where each of these actions has a condition which is a sufficient condition for the occurrence of *a* or *b*.

The correctness of this refinement can be intuitively assessed if we consider that the behaviour in Figure 6.23 has been built in two steps: (i) choosing a correct concrete behaviour for the abstract action choice behaviour, depicted in Figure 6.23, and (ii) by repeating this correct concrete behaviour in a mirrored form with respect to *a* and *b*, which makes the whole behaviour symmetric and gives fair chances for the execution of *a* and *b*.

6.4.2 Timing Requirements

Some examples of timing requirements, alternative refinements and assignments of constraints to functional entities are discussed below.

Figure 6.25 illustrates a possible refinement of synchronous timing requirements.

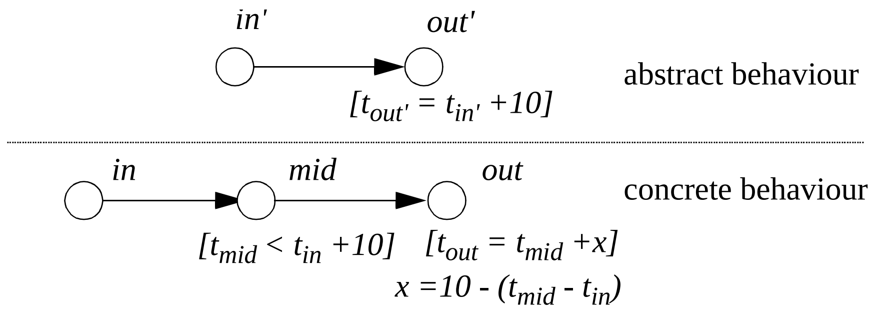

in'
out'
abstract behaviour
[$t_{out'} = t_{in'} + 10$]
in
mid
out
concrete behaviour
[$t_{mid} < t_{in} + 10$]
[$t_{out} = t_{mid} + x$]
$x = 10 - (t_{mid} - t_{in})$

Figure 6.25: Refinement of synchronous timing requirements

In the concrete behaviour of Figure 6.25 action *mid* has been inserted between actions *in'* and *out'*. Since *out*' should happen at moment $t_{in'} + 10$, information on how much time is left for $t_{in} + 10$ has to be provided as a value of information to *out* through action *mid*.

The concrete behaviour of Figure 6.25 still leaves some flexibility in the time of occurrence of *mid*. Another more strict refinement could impose, for example, that *mid* happens at moment $t_{in} + 5$, and that out happens at moment $t_{mid} + 5$, such that the original time condition is built in the time conditions of the refinement. An advantage of this refinement is that the provision of information from *mid* to *out* about timing constraints is not necessary any more.

It is interesting to consider the possible assignments of constraints to functional entities in the example above, in a similar way as in [5]. Suppose there are two functional entities F_1 and F_2, that handle possible value references between *in* and *mid*, and *mid* and *out*, respectively, two functional entities $Timer_1$ and $Timer_2$ that handle timing constraints and an environment.

Figure 6.26 depicts a possible assignment of constraints to these functional entities.

Since constraints on action values are in principle independent of timing constraints, such a structure of functional entities should produce the desired integrated behaviour.

6.4.3 Data Splitting

Consider an instance of data transfer, which we call a word transfer service instance ([2]). The behaviour of this service is such that an action req_a in which a 16-bit word of data is established is

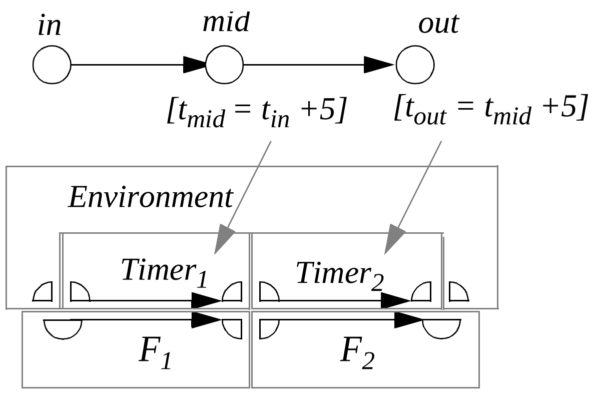


Figure 6.26: Assignment of constraints after refinement

followed by a corresponding action *ind_a* in which the data is delivered. This behaviour is defined as follows:

```
WTS (* Word Transfer Service Instance*) :=
{       start -> $req_a$ ($w_1$: word),
        $req_a$ ($w_1$:word) -> $ind_a$ ($w_2$:word) [$w_2$=$w_1$] }
```

Suppose we want to refine this service, aiming at defining three functional entities: a transmitting and a receiving protocol entity and an (underlying) octet transfer service provider. Two alternatives for the behaviour of the octet transfer service are considered: (i) two independent channels for octet transfer, and (ii) a single channel for octet transfer which preserves the order of the octets. We do not consider timing nor probability aspects in this example.

Underlying Independent Channels

Using two independent channels as underlying octet transfer service we have that action *req* causes two actions, *req_1* and *req_2*, independent of each other, each one of them establishing an octet. Actions *req_1 and req_2* cause actions *ind_1* and *ind_2*, respectively, in which the octets are transferred. Both *ind_1* and *ind_2* must occur in order to cause *ind*. Actions *req* and *ind* are supposed to be the concrete counterparts of actions *req_a* and *ind_a*, respectively. We assume that we have assigned the transfer of the first octet to one of the channels and the transfer of the second octet to the other one, allowing the word to be assembled in *ind*. This behaviour is defined below:

```
$WTS_1$ :=
{       start -> req ($w_1$:word),
        req ($w_1$:word) -> $req_1$ ($o_1$:octet) [$o_1$=first($w_1$)],
        req ($w_1$:word) ->
                $req_2$ ($o_1$:octet) [$o_2$=second($w_1$)],
        $req_1$ ($o_1$:octet) -> $ind_1$ ($o_3$:octet)[$o_3$=$o_1$],
        $req_2$ ($o_2$:octet) -> $ind_2$ ($o_4$:octet)[$o_4$=$o_2$],
        $ind_1$ ($o_3$:octet) $\wedge$ $ind_2$ ($o_4$:octet) ->
                ind ($w_2$:word) [$w_2$ =conc($o_3$,$o_4$)] }
```

Applying the method given before for determining the abstraction of this behaviour we can abstract from ind_1 and ind_2, by replacing them by their conditions req_1 and req_2, respectively, in the causality relation of *ind*, according to abstraction rule 1. The constraint $w_2 = conc(o_3, o_4)$ can be also replaced by $w_2 = conc(o_1, o_2)$ according to abstraction rule 1a. Applying abstraction rule 1 again to abstract from req_1' and req_2' we replace them by their condition *req'*, which results in the condition $req' (w_1{:}word) \wedge req' (w_1{:}word)$, which can be simplified to $req' (w_1{:}word)$ for *ind'*. In case $conc\ (first(w), second(w)) = w$ we can also replace $w_2 = conc(o_1, o_2)$ by $w_2 = w_1$ in the constraint of *ind'*. The resulting abstract behaviour is the following:

```
WTS₁'' :=
{       start -> req'' (w₁:word),
        req'' (w₁:word) -> ind'' (w₂:word) [w₂ =w₁] }
```

By inspection we can conclude that behaviour WTS_1'' conforms to the original abstract behaviour *WTS*, since *req''* and *ind''* have the same causality relationships and value dependencies as req_a and ind_a.

Underlying FIFO Channel

Using a FIFO channel for the underlying octet transfer service we get a behaviour in which initially action *req* causes an action req_1, where the first octet of the word is established. The value of *word* is kept in the retained values of action req_1, which causes req_2, where the second octet is established. Actions req_1 and req_2 cause actions ind_1 and ind_2, respectively, in which the octets are transferred. Action ind_1 is one of the necessary conditions of ind_2, and the first octet is kept in the retained values of ind_2. Action *ind* is caused by ind_2. Again actions *req* and *ind* are supposed to be the concrete counterparts of actions req_a and ind_a, respectively. This behaviour can be defined as follows:

```
WTS₂ :=
{       start -> req (w₁:word),
        req (w₁:word) ->
                req₁ (o₁:octet,w₁:word) [o₁=first(w₁)],
        req₁ (o₁:octet,w₁:word) ->
                req₂ (o₂:octet) [o₂=second(w₁)],
        req₁ (o₁:octet, w₁:word) -> ind₁ (o₃:octet)[o₃=o₁],
        req₂ (o₂:octet) ∧ ind₁(o₃:octet) ->
                ind₂ (o₄:octet, o₃:octet) [o₄=o₂]
        ind₂ (o₄:octet, o₃:octet) ->
                ind (w₂:word) [w₂=conc (o₃,o₄)] }
```

Applying the method given before for determining the abstraction of this behaviour we can start by abstracting from ind_2 by replacing it by its conditions $req_2 \wedge ind_1$ in the causality relation of *ind* according to abstraction rule 1. The constraint $w_2 = conc\ (o_3, o_4)$ can be also replaced by $w_2 = conc\ (o_3, o_2)$ according to abstraction rule 1a. Applying abstraction rule 1 again to abstract from req_2' and ind_1' we replace them by their condition req_1' in the causality relation of *ind'*, which results in the condition $req_1' \wedge req_1'$, which can be simplified to req_1'. We can also replace $w_2 = conc(o_3, o_2)$ by $w_2 = conc(o_1, second(w_1))$ according to abstraction rule 1a. Finally we can replace req_1'' by its condition req'' and the constraint $w_2 = conc(o_1, second(w_1))$ by $w_2 = conc(first(w_1), second(w_1)$, which can be simplified to $w_2 = w_1$ if $conc(first(w_1), second(w_1) = w_1$, in the causality relation of *ind''*. The resulting abstract behaviour is the following:

```
WTS₂''' :=
{       start -> req''' (w₁:word),
        req''' (w₁:word) -> ind''' (w₂:word) [w₂ =w₁] }
```

By inspection we can conclude that behaviour WTS_2''' conforms to the abstract behaviour *WTS*.

Figure 6.27 depicts these alternative refinements.

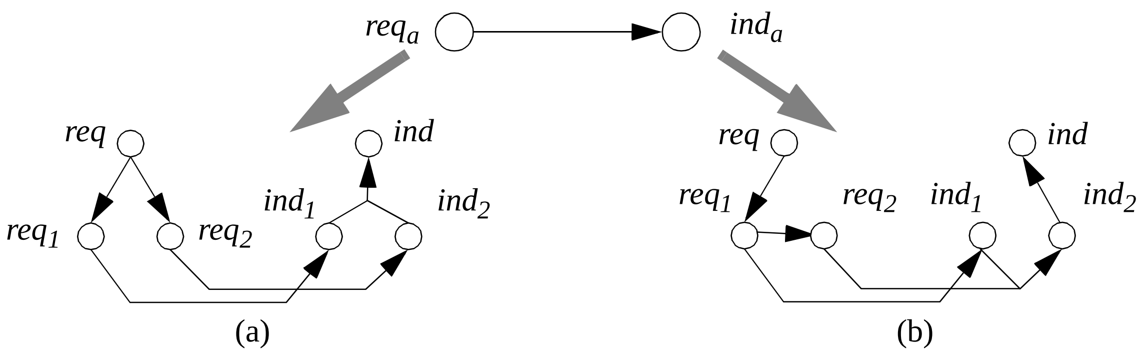


Figure 6.27: Refinements of a Word Transfer Service with underlying (a) independent channels and (b) FIFO channel

Functional Entities

Once we have determined the concrete behaviour in terms of actions, the constraints on these actions can be distributed over functional entities, making actions turn into interactions. In the concrete behaviour of Figure 6.27 (a) we can assign the causality relations between req_1 (req_2) and ind_1 (ind_2) to the octet transfer service provider, while the other relationships are assigned to the word transfer protocol entities. Figure 6.28 depicts these assignments.

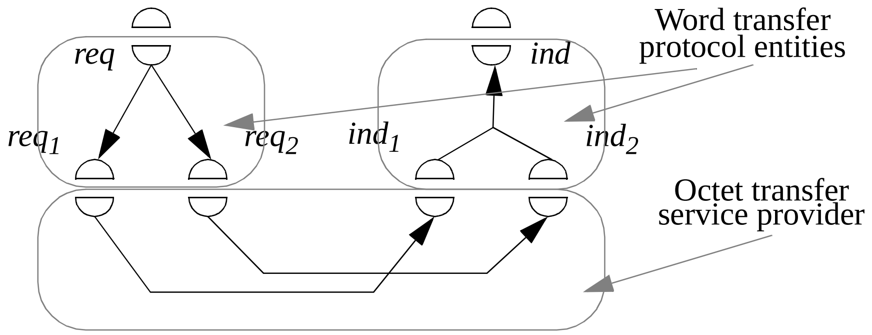

req
ind
req_1
req_2
ind_1
ind_2
Word transfer protocol entities
Octet transfer service provider

Figure 6.28: Assignment of behaviours to functional entities for refinement with independent channels

In the concrete behaviour of Figure 6.27 (b), we can assign the causality relations between req_1 (req_2) and ind_1 (ind_2) to the octet transfer service provider. The ordering relation between req_1

and req_2 is distributed over the service provider and the transmitting protocol entity, since this protocol entity imposes the establishment of the first (second) octet in req_1 (req_2). The retained value of req_1 (w_1), can be only be handled by the transmitting protocol entity. The octet transfer service provider can handle one octet at a time, imposing that req_1 and req_2 should happen in sequence. This order is preserved by the service provider during transfer, which is represented by the causality relation between ind_1 and ind_2 in the service provider. In the receiving protocol entity the retained value of ind_2 is used to pass the first octet established in ind_1 to *ind*. Figure 6.29 depicts these assignments.

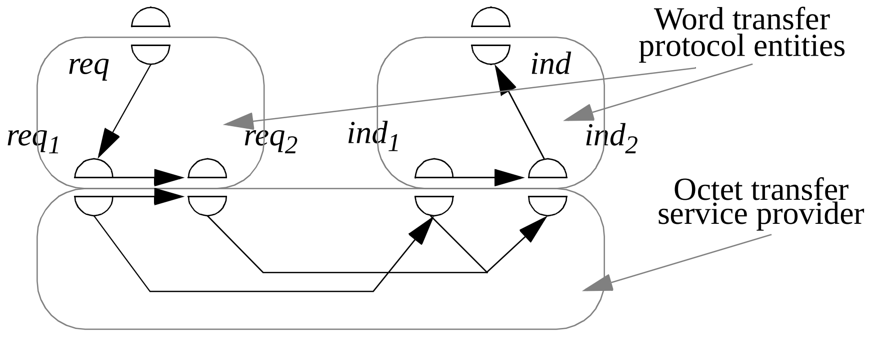

req
ind
Word transfer protocol entities
req1
req2
ind1
ind2
Octet transfer service provider

Figure 6.29: Assignment of behaviours to functional entities for refinement with FIFO queue

Although the values of information established in an action have to be the same for each partition of the action when it turns into an interaction, this is not the case for the retained values. In Figure 6.29, for example, the retained value of req_1 can only be handled by the transmitting protocol entities, since it implies participation in action *req*, where this value is established. Therefore the retained values are not shared by the functional entities participating in an interaction in the same way as action values are.

In both cases of behaviour assignment to functional entities discussed above the protocol entities and their corresponding word transfer service users share the execution of *req* and *ind*. We suppose that the transmitting protocol entity is always ready to perform *req* while the receiving service user is always ready to perform *ind*.

Assignments of constraints to functional entities do not disturb the correctness of the behaviour refinement, because the decomposition of constraints follows the rules presented before in Chapter 5. In our examples, since the concrete behaviours of Figure 6.27 are correct refinements of the abstract behaviour, we can also assume that the behaviours of Figure 6.28 and Figure 6.29 are correct refinements of the abstract behaviour.

6.4.4 Data Retransmission

Consider the abstract behaviour of Figure 6.27 and suppose we want to refine this behaviour by introducing an underlying service which keeps the ordering of the transmitted data but may loose data. Furthermore, the probability that data arrives (p_{ind}) of the original behaviour is supposed to be bigger than the probability that data arrives of the underlying service, which means that a retransmission mechanism should be defined. In order to make our discussion more concrete we suppose that the desired data arrival probability can be obtained if data is retransmitted twice.

Figure 6.30 depicts the behaviour of an underlying service which is unreliable but delivers data in order, for three instances of data transmission.

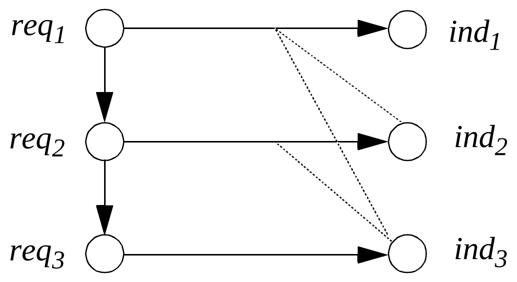


Figure 6.30: Unreliable FIFO medium

In the behaviour depicted in Figure 6.30, in case a certain ind_i occurs it excludes the other ind_j for $j < i$. This implies that either data arrive in the correct order, or they do not arrive, such that data do not overtake earlier sent data.

We also assume that there is a medium with a similar behaviour as depicted in Figure 6.30 in the opposite direction of the data transfer, which can be used to by the receiving side to acknowledge the reception of data.

The transmitting side of the concrete behaviour should send data after a *req*, and wait for an acknowledgment or a timeout *T*. Data is retransmitted if no acknowledgement arrives before the timeout. We have supposed that data can be retransmitted at most twice. One of the design objectives is to minimize the use of the underlying service, otherwise we would simply send data three times without the need for acknowledgement.

Figure 6.31 depicts the behaviour of the transmitting side.

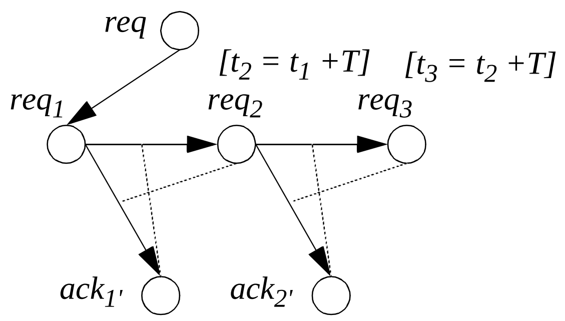


Figure 6.31: Transmitting side of concrete behaviour

The receiving side of the concrete behaviour receives data from the underlying service, generates an *ind* with data, and acknowledges the data. Each new data item that arrives is simply acknowledged, since at this point this instance of data transfer has been already completed. Data is received at most three times, namely in case two acknowledgements are lost.

Figure 6.32 depicts the behaviour the receiving side.

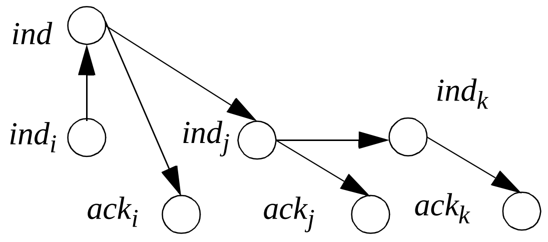


Figure 6.32: Receiving side of concrete behaviour

Figure 6.33 depicts the composite behaviour of the transmitting side, receiving sides and underlying service, refraining from the explicit representation of some conditions and constraints. For example *ind* should be caused by the earliest of ind_1, ind_2 and ind_3 and it should cause the corresponding acknowledgement.

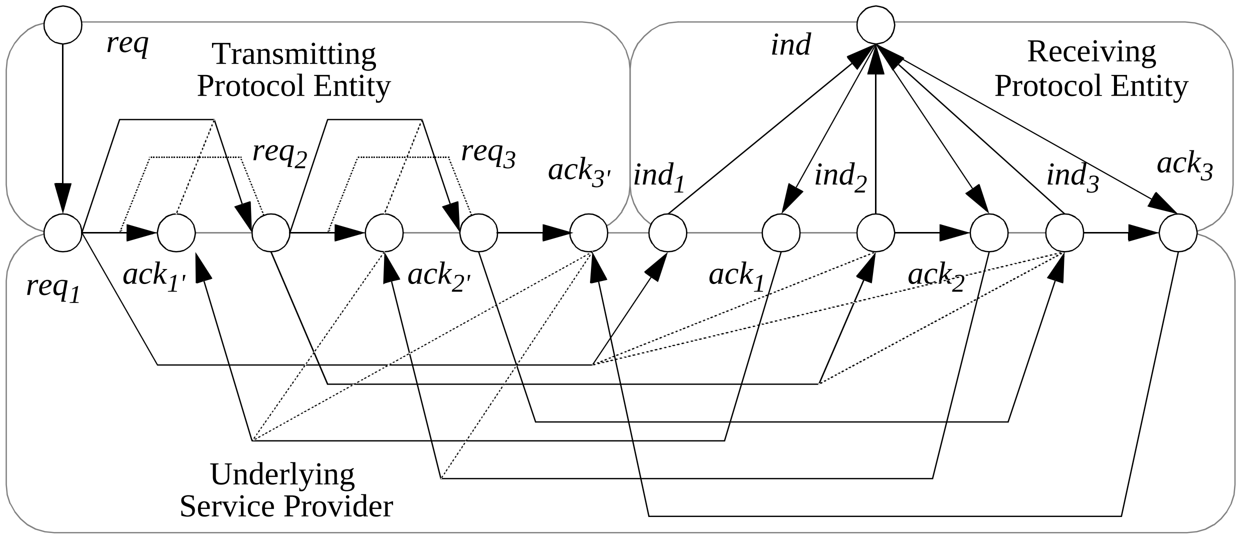

req
Transmitting Protocol Entity
ind
Receiving Protocol Entity
req_2
req_3
ack_3'
ind_1
ind_2
ind_3
ack_3
req_1
ack_1'
ack_2'
ack_1
ack_2
Underlying Service Provider

Figure 6.33: Composite behaviour of retransmission mechanism

Figure 6.33 also identifies the functional entities in the composite behaviour. Some alternative mappings of conditions and constraints onto the behaviour of these functional entities are possible. Specific design choices are not discussed here, since it deviates too much from the purpose of this section. Abstracting from actions req_i, ind_j, ack_n and $ack_{m'}$ in the concrete behaviour following the abstraction rules given before we should obtain the original abstract behaviour.

6.4.5 Data Multiplexing

Consider a behaviour consisting of three independent instances of data transfer, where each of the instances is similar to the word transfer service instance of Figure 6.27. This behaviour is represented by the pairs of interactions req_i'', ind_i'', where $i \in \{1, 2, 3\}$. Figure 6.34 depicts this behaviour.

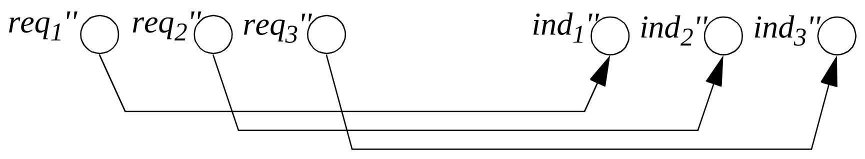


Figure 6.34: Three independent instances of data transfer

Suppose that these three instances of communication have to be multiplexed in a single instance of data transfer at an underlying service. The design objective of this refinement may be to achieve a better use of the underlying communication service, for cost or performance reasons. Figure 6.35 depicts the refinement obtained by performing this design operation.

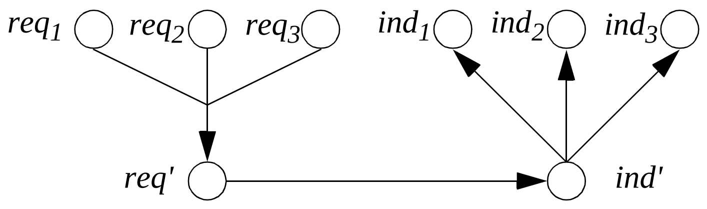


Figure 6.35: Data multiplexing

Pairs <req_i, ind_i> are independent of each other in the original behaviour definition, but not any more in the refinement of Figure 6.35. This refinement imposes that each ind_i only happens in case all req_i's have happened. However, the original relationship between each req_i and ind_i is preserved in this refinement, since ind_i only happens if req_i has happened.

This example shows that correctness notions may depend on specific design objectives of the refinements. This also implies that although standard abstraction or conformance rules can be applied in large instances of design steps, sometimes we must consider more specific rules depending of specific design objectives. The extra condition introduced in this refinement, namely that all req_i's have to happen for ind_i's to happen, should be acceptable, since one cannot achieve the objective of this refinement without creating this extra condition.

Since we have introduced new relationships between the pairs <req_i, ind_i> with respect to the behaviour depicted in Figure 6.34, it is possible to define an intermediate behaviour, in which the actions *req'* and *ind'* are not inserted yet, but that represents these new relationships between the pairs <req_i, ind_i>. This intermediate behaviour is depicted in Figure 6.36.

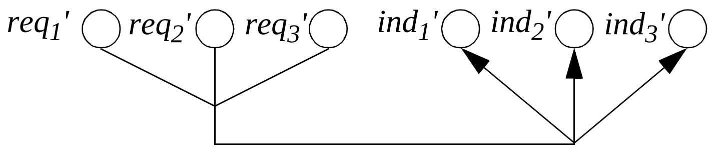


Figure 6.36: Intermediate behaviour for introducing multiplexing

An alternative approach towards the elaboration of this refinement is to replace the behaviour of Figure 6.34 by the behaviour of Figure 6.36 before applying behaviour refinement.

6.4.6 User Identification

Consider a multi-user system in which each user is allowed to use the system once at a time. We concentrate on this requirement in this example, abstracting from the specific facilities provided by the system. User identification is established in a *login* action between the system and its users. We model *login* actions as interleaved actions in the abstract behaviour, such that only user identifiers that are not in use are established each time. In this example we do not model the *logout* actions necessary to release identifiers, for the sake of simplicity.

Figure 6.37 depicts the user identification aspects of this system for three users

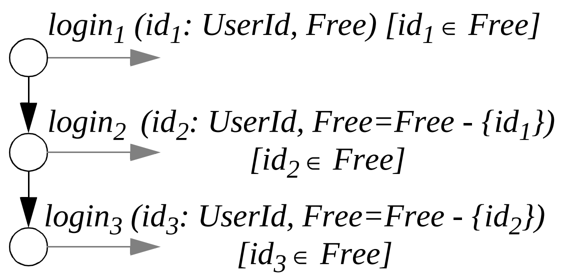

$login_1$ (id_1: UserId, Free) [$id_1 \in$ Free]
$login_2$ (id_2: UserId, Free=Free - {id_1}) [$id_2 \in$ Free]
$login_3$ (id_3: UserId, Free=Free - {id_2}) [$id_3 \in$ Free]

Figure 6.37: User identification

In the behaviour depicted in Figure 6.37 each $login_i$ action makes use of an id_i value which is in the set of free identifiers *Free*. The set *Free* is updated each time an identifier is used, by removing this identifier from the set. A similar example has been discussed in [5].

This system should be able to support geographically distributed users, such that a protocol becomes necessary. Two alternatives for the implementation of the user identification function are discussed below: a distributed implementation and a centralized implementation.

Distributed Implementation

In the distributed implementation the protocol entities determine locally whether the *login* action may or may not take place, by using a token mechanism: only a protocol entity that owns the token is allowed to perform a *login* action. The contents of the set of free identifiers *Free* circulates between the protocol entities together with the token. Set *Free* is used to constrain the *login* actions, and is updated each time a *login* action occurs. The underlying communication service between the protocol entities resembles a virtual ring, where the token is successively sent from a protocol entity to the next protocol entity in the ring.

Figure 6.38 depicts the behaviour refinement with the token mechanism for three users and three possible occurrences of a *login* action.

Figure 6.38 does not represent all conditions and constraints that apply to each action. For example, each *info_resp* happens either caused by *login*, or a certain timeout *T* after *info_ind*. Each action *login* is excluded by *info_resp*, such that no action *login* happens after the timeout *T*. According to this mechanism each *login* gets the chance of being executed when the corresponding protocol entity has the token.

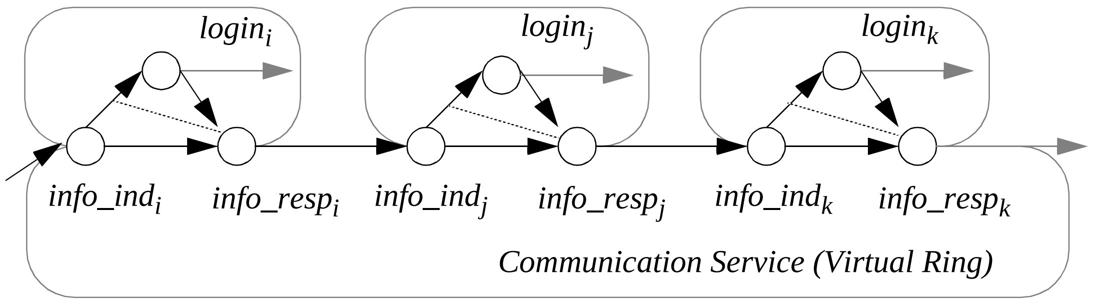


Figure 6.38: Distributed implementation of user identification

A drawback of this solution is the intensive use of the underlying communication service, even if no user is wishing to execute a *login* action.

Intuitively we expect that the distributed implementation yields a correct refinement of the abstract behaviour depicted in Figure 6.37. Each *login* of Figure 6.37 corresponds to a *login* that actually happens in Figure 6.38, and each *login* that actually happens in Figure 6.38 can refer to the set *Free*, which circulates among the protocol entities, such that the conditions and constraints of the original behaviour are preserved. For example, suppose $login_i$ and $login_k$ happen in an execution of the behaviour of Figure 6.38, they correspond to $login_1$ and $login_2$ of Figure 6.37, respectively.

Centralized Implementation

In the centralized implementation a central scheduler functional entity (*Scheduler*) manages the set of free identifiers *Free*. Each *login* action is decomposed in two actions, a *login_req* and a *login_conf*, such that the occurrence of a *login_conf* corresponds to the occurrence of a *login* action in the abstract behaviour.

In order to perform this refinement we are forced to take a design decision concerning which functional entity imposes the user identifier to be established. There are two options: (i) the *login_req* contains a user identifier parameter that is checked by the system against the set *Free*, determining whether this *login_req* can be confirmed or rejected, or (ii) the system determines a free identifier from *Free* for each *login_req*, and informs this identifier to the user in a *login_conf* or rejects the *login_req* if no free identifier exists. We assume option (i) in this example because we consider that each user wants to perform a *login* with a specific user identification.

Action *login_rej* is inserted to model that a *login_req* is rejected in case the identifier offered by the user is not in the set *Free*. Each *info_req* and its corresponding *info_ind* contain an user identifier which is checked against the set of free identifiers of *Scheduler*. Each *info_resp* and its corresponding *info_conf* contain the information of whether the user identifier was in the set of free identifiers or not. This information is used by the local protocol entities to enable a *login_conf* or a *login_rej*.

Figure 6.39 depicts an instance of behaviour of this refinement for three user and three attempts to execute a *login* action.

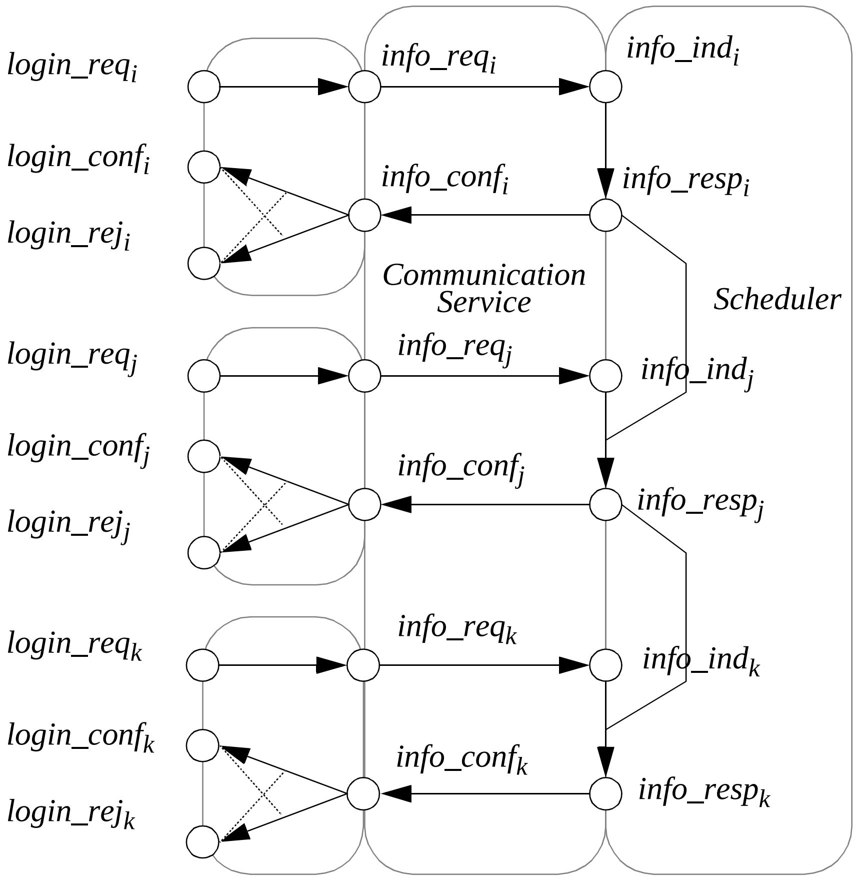


Figure 6.39: Centralized implementation of user identification

A possible advantage of the centralized implementation with respect to the distributed implementation is that in case the service users do not wish to perform a *login login_req* actions are not performed and no use of the communication service is made.

Although this refinement seems to be a correct implementation of the behaviour depicted in Figure 6.37, assessing the correctness of this refinement is not trivial. Firstly each *login* action has been decomposed in a behaviour involving *login_req*, *login_conf* and *login_rej*, such that the occurrence of *login_conf* corresponds to the occurrence of *login*. This design operation alone resembles the *action refinement* design operation treated in Chapter 7 of this thesis. Secondly the behaviour pattern of the abstract behaviour consists of interleaved *login* actions making use and updating set *Free*, which cannot be found back in the relationships between the *login_conf*'s in the concrete behaviour. Even worst, *login_conf*'s are independent of each other, and may even happen at the same time.

However, there are two indications that this refinement can be considered as a correct implementation of the original behaviour: (i) the relationship between *info_resp*'s at the scheduler resembles the original relationship between *login*'s in the abstract behaviour and (ii) the constraint that a *login* only establishes identifiers that were in set *Free*, which is the most important functional requirement of the original behaviour, is guaranteed in this refinement.

By making all actions interleaved, for example using some formal model based on arbitrary interleaving, one could simply try to artificially prove the correctness of this refinement. Intuitively, as *login_conf*'s become interleaved due to the interleaving semantics, they should have the same relationships as the original *login*'s, allowing the correctness of the refinement to be proved. However we strongly doubt the integrity and validity of such a proof strategy, since it is based on properties that the behaviour should not have rather than the properties that the behaviour must have.

6.5 References

[1] E. Brinksma, J.-P. Katoen, R. Langerak, and D. Latella. Performance analysis and true concurrency semantics. Technical Report Memoranda Informatica 94-39 (TIOS 94-10), University of Twente, Enschede, the Netherlands, June 1994.

[2] L. Ferreira Pires, M. van Sinderen, and C. A. Vissers. Advanced design concepts for distributed systems development. In *Proceedings of the Fourth Workshop on Future Trends of Distributed Computing Systems*, pages 419–425, Los Alamitos, USA, Sept. 1993. IEEE Computer Society Press.

[3] P. W. King. Formalization of protocol engineering concepts. *IEEE Transactions on Computers*, 40(4):387–403, April 1991.

[4] R. Langerak. *Transformations and Semantics for LOTOS*. PhD thesis, University of Twente, Enschede, Netherlands, 1992.

[5] J. Schot. *The role of Architectural Semantics in the formal approach towards Distributed Systems Design*. PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[6] M. van Sinderen, L. Ferreira Pires, and C. A. Vissers. Protocol design and implementation using formal methods. *The Computer Journal*, 35(5):478–491, Oct. 1992.

Chapter 7

Action Refinement

This chapter discusses the *action refinement* design operation, which consists of replacing an action by an activity. Since an action can be replaced by different alternative activities, and the choice of a specific activity is determined by specific design objectives, this design operation can not be automated in its totality. However one can determine the correctness of this design operation by checking whether the activity conforms to the replaced action and whether the activity is properly embedded in the context of the replaced action.

This chapter is structured as follows: section 7.1 defines and motivates action refinement, section 7.2 presents the rules that determine which attribute values an action should have in order to be an abstraction of an activity, and section 7.3 presents the rules that determine whether an action is an abstraction of an activity in its context, allowing one to check whether a certain activity is a correct implementation of an action. Section 7.4 applies these rules to check the simultaneous refinement of multiple actions and section 7.5 discusses a design step that can be supported by action refinement: the replacement of an interaction point by a functional entity.

7.1 Definition

Action refinement consists of replacing an action of an abstract behaviour by a conforming activity of a concrete behaviour. Similarly to Chapter 6, all actions of an abstract behaviour are called *abstract reference actions* in the sequel, and are the reference points for assessing the correctness of the concrete behaviours.

In action refinement, a certain abstract reference action is replaced by an activity. An activity is a composition of actions, which are supposed to be more concrete than its corresponding abstract reference action. Activities are defined by behaviours. Activity values are the values of information established by some actions of an activity and referred to by other actions or activities outside the activity.

7.1.1 Behaviour Refinement and Action Refinement

The difference between action refinement, as it is discussed in this chapter, and behaviour refinement, as it is discussed in Chapter 6, is subtle but rather essential. A proper comprehension of this subtle difference helps understanding the purpose of action refinement. Below action refinement is illustrated by an example where only distribution of the action values is considered. The other action attributes are addressed later on.

Example

While in behaviour refinement an abstract reference action should establish the same values of information as its corresponding concrete reference action, in action refinement the activity that replaces an abstract reference action must establish the same values of information as its abstract reference action. However an activity may establish its values in multiple concrete actions, such that only when all these concrete actions occur the values of this activity are available. Therefore a necessary condition for an abstract reference action to properly represent an activity is that this abstract reference action establishes the same values of information as the activity values.

Figure 7.1 depicts an example to illustrate the difference between behaviour refinement and action refinement. In this example we consider an abstract behaviour *B* containing reference actions *a'* and *b'*. We also suppose that values v_{b1} and v_{b2} are established in action *b'*.

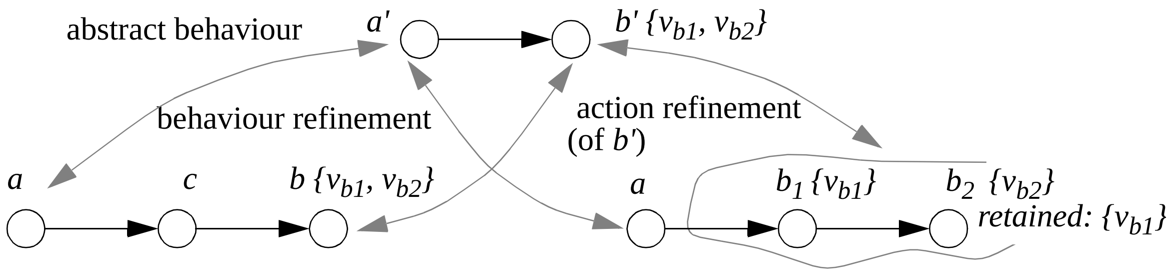


Figure 7.1: Difference between behaviour refinement and action refinement

Performing behaviour refinement we may consider *a'* and *b'* as abstract reference actions, and *a* and *b*, respectively as their corresponding concrete reference actions. Action *c* is inserted between actions *a* and *b*, making it possible for *b* to indirectly refer to attribute values of *a* through *c*. One of the conformance requirements of this design operation imposes that concrete reference actions should have the same attribute values as their corresponding abstract reference actions. Particularly both actions *b* and *b'* should establish values v_{b1} and v_{b2}.

Performing action refinement we may replace the abstract reference action *b'* by a behaviour consisting of actions b_1 and b_2. The pattern of behaviour obtained in this way is similar, in Figure 7.1, to the pattern of behaviour obtained by behaviour refinement. However, in action refinement the values established by the abstract reference action may be established by more than one action in the activity that refines this abstract reference action. In the example of action refinement in Figure 7.1 values v_{b1} and v_{b2} are established by b_1 and b_2, respectively. In action refinement, the abstract reference action models that all values of its corresponding activity are made available. In the example of action refinement in Figure 7.1 this happens when b_2 occurs. Furthermore it is important to notice that b_2 retains value v_{b1} established in b_1, such that the total of values made available when b_2 happens is the same as the total of values established by the abstract reference action *b'*.

7.1.2 Attribute Decomposition

In general the essence of action refinement is the decomposition of at least one of the action attributes of the abstract reference action. This implies that not only action values, but also location, time and probability of an abstract reference action may be distributed over actions of an activity:

- the location attribute of an abstract reference action represents a certain action point, which may be decomposed in more concrete (sub-)actions points, where the actions of the activity that refines the abstract reference action occur;
- the time attribute of the abstract reference action can be decomposed and distributed over the time attributes of the actions of the activity that refines the abstract reference action;
- the probability of occurrence of the abstract reference action can also be decomposed and distributed over the actions of the activity that refines the abstract reference action.

In the instance of action refinement shown in Figure 7.1, for example, the time attribute of the abstract reference action b' could be decomposed in the time attribute values of actions b_1 and b_2. However it is enough to make the time of occurrence b_2 be the same as the time attribute of b', since when b_2 happens all activity values are made available. Furthermore the locations of b_1 and b_2 may be sub-locations of the location of b', and the probability that b' happens given that a' has happened should be the same as the probability that *both* b_1 and b_2 happen given that a has happened, such that the probability attribute of b' is also distributed over the probability attributes of b_1 and b_2.

More general rules for attribute distribution are presented in section 7.2.

Attribute Representation

The representation of the attributes of the abstract reference action plays an important role in the ways distribution of attribute values in action refinement can be performed. Distribution of action values, for example, may be performed in two different ways, depending on how the action value of an abstract reference action is represented:

1. distribution of action values may be straightforward, in case the action value of an abstract reference action is defined in terms of a composition of values;
2. it may be necessary to modify the representation of the action value of an abstract reference action before distribution of values is performed, in order to match specific distribution objectives. In this case action values are first *coded* before being distributed.

In the example of Figure 7.1 the action value of action b' is composed of two values v_{b1} and v_{b2}, which made it possible to directly distribute them on actions b_1 and b_2, respectively. This corresponds to option (1) above. In the same example we could have considered that the action value of b' was a single octet, which would be first decomposed in two hexadecimal digits before these two digits were distributed over b_1 and b_2. This corresponds to option (2) above.

Location attribute values can be defined as hierarchies of locations and sub-locations. This is comparable, for example, to the abstraction levels at which home addresses can be considered: we may start for example by considering that an action occurs in a certain town, distribute this abstract action over actions that occur at different districts of this town, and further distribute these actions over actions that occur at individual home addresses in these districts.

7.1.3 Correctness Requirements

In action refinement we should be able to determine if the activity that replaces an abstract reference action is a correct implementation of this action in its context. This implies that two essential correctness requirements can be identified in action refinement:

1. conformance between an activity and the abstract reference action;
2. proper embedding of an activity in the context of the abstract reference action.

The approach towards requirement (1) is to determine the rules for considering an abstract reference action as an abstraction of an activity, and apply these rules for assessing whether an activity conforms to an abstract reference action. Requirement (1) is supported by the rules of action modelling. These rules determine the attribute values which should be assigned to an abstract reference action in order to consider this action as an abstraction of a certain activity, which characterizes *attribute abstraction*.

The approach towards requirement (2) is to determine the rules for abstracting from the specific ways an activity relates to other activities and actions and apply these rules to determine whether specific activities embedded in the concrete behaviour correctly implement the abstract reference action embedded in the abstract behaviour. Requirement (2) is supported by the rules for abstracting from the specific embedding of an activity in a concrete behaviour, which characterizes *context abstraction*.

Figure 7.2 depicts the relationship between attribute abstraction, context abstraction and the design choice to be taken in action refinement.

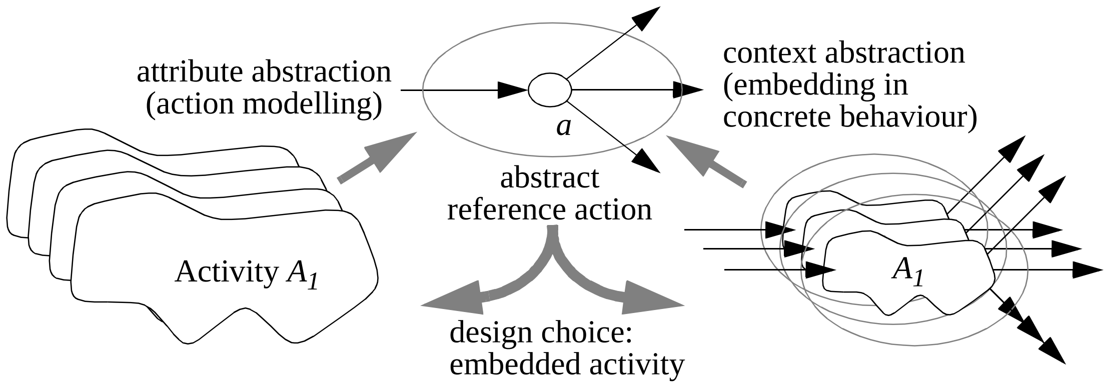


Figure 7.2: Elements of action refinement

Figure 7.2 depicts the refinement of a single action. In general, multiple actions may be refined in a single design operation. The two correctness requirements identified above apply to each individual activity and its corresponding abstract reference action in case multiple abstract reference actions are refined at the same time.

7.2 Attribute Abstraction

An action is a proper abstraction of an activity if it has attribute values that represent the attributes of the activity, namely location, values, time and probability. This correspondence is defined in terms of rules which determine the attribute values that an abstract reference action should have in order to be an abstraction of an activity. General rules, which apply to activities of any form, and specific rules, which apply to certain activity forms, are presented in this section.

7.2.1 Terminology and Notation

Since the textual description of rules for attribute abstraction can be become rather confusing, we introduce some terminology and notation in order to make it more clear and precise. The terminology and notation used from now on is the following:

Action Attributes

- $v(a)$: action values of action a;
- $r(a)$: retained values of action a;
- $f(a)$: functionality of action a, according to the definition $f(a) = v(a) \cup r(a)$[1];
- $t(a)$: time of action a;
- $l(a)$: location of action a;

Activity Definitions

- A: activity, which is a form of behaviour defined in terms of a finite number n of activity actions;
- $v(A)$: activity values, which are the values of information established in (some) actions of activity A and referred to by actions and activities outside A;
- $r(A)$: activity retained values, which are the values of information established in actions or activities outside A, retained by actions of A, and referred to, through actions of A, by other actions and activities outside A;
- $f(A)$: activity functionality, defined as $f(A) = v(A) \cup r(A)$. The activity functionality determines the total of values made available by an activity;
- a_i: activity action, which is an action that belongs to an activity A. We also denote the actions of A by the set $\{a_i \mid i \in \{1,..., n\}\}$.

The definitions above imply that $v(A) \subseteq \bigcup_{i=1}^{n} v(a_i)$ and $r(A) \subseteq \bigcup_{i=1}^{n} r(a_i)$.

1. We consider each action value or retained value attribute as a bag or multi-set, such that the union of these values may contain multiple instances of identical values.

7.2.2 General Rules

Suppose we have an activity *A* that we want to represent by a single abstract reference action *a*. The following rules can be defined for the action values, retained values and location of action *a*, with respect to the characteristics of the activity:

1. *action value*: the abstract reference action establishes the same values as the activity values;
2. *retained values*: the abstract reference action has the same retained values as the activity retained values;
3. *location*: the abstract reference action should have a location that is an abstraction of the location of all actions of the activity, or it should have the same location of all actions of the activity, if they all have the same location. In the former case the locations of the actions in the activity are all contained in the location attribute of the abstract action, which may be represented as a single more abstract location.

The rules presented textually above are defined using set notation in Table 7.4, without considering a possible more abstract representation of value and location attributes in the abstract reference action.

Table 7.1: General rules for attribute abstraction

Rule 1	*Rule 2*	*Rule 3*
$v(a) = v(A) \subseteq \bigcup_{i=1}^{n} v(a_i)$	$r(a) = r(A) \subseteq \bigcup_{i=1}^{n} r(a_i)$	$l(a) = \bigcup_{i=1}^{n} l(a_i)$ $l(a) = l(a_i)$ if $\forall\, a_i, a_j \in A,\ l(a_i) = l(a_j)$

Rules 1 and 2 imply that the functionality of the abstract reference action is the same as the activity functionality, which corresponds to the requirement that the total of values made available by the activity is the same as the total of values made available by its abstract representation.

According to our notation $f(a) = f(A) \subseteq \bigcup_{i=1}^{n} v(a_i) \cup \bigcup_{i=1}^{n} r(a_i)$.

7.2.3 Single Final Action

An activity may have a single final action, such that this activity makes all its values available through this final action.

Figure 7.3 depicts an arbitrary activity with a single final action.

In Figure 7.3, the occurrence of action a_5 indicates that the activity makes all its values available, since a_5 is the last action of this activity. It is possible that not all values of this activity are established in a_5, i.e. parts of these values may be established in actions a_1, a_2, a_3 and a_4, being retained and made available when a_5 happens.

In an activity with a single final action, the final action should allow other actions outside this activity to refer to the activity values. Therefore the attributes of the final action should contain

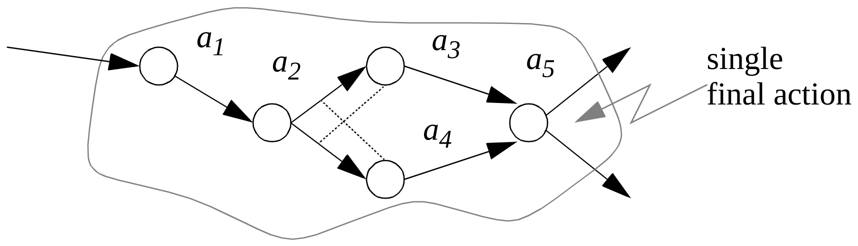


Figure 7.3: Arbitrary activity with single final action

the activity values, either in the action values of the final action, or in the retained values of the final action, or in both action values and retained values of the final action. Activity values contained in the action values of the final action are the values established in the final action, while activity values contained in the retained values of final action have been established in other actions of the activity and retained by the final action for further reference. The final action should also allow other actions outside this activity to refer to the activity retained values, which should be contained in the retained values of the final action.

Considering an activity with a single final action, such as the example in Figure 7.3, we can define more specific rules for attribute abstraction:

1. *action value*: the action values of the abstract reference action must be contained in the functionality of the final action of the activity, being either established values or retained values of this final action. These values may have an abstract representation (coding) in the abstract reference action, in which these values are integrated, for example in case they are represented as a single value of a proper sort;
2. *retained values*: the retained values of the abstract reference action must be contained in the retained values of the final action;
3. *time*: the abstract reference action is expected to occur when all values of the activity are available. This happens when the final action of the activity occurs, which implies that the abstract reference action must have the same time attribute value of the final action;
4. *probability*: the probability attribute of the abstract reference action should be the probability that the final action of the activity occurs, considering that the conditions of the activity are satisfied. This implies that the probability attribute of the abstract reference action cannot be solely inferred from the probability attribute of the final action of the activity without considering the concrete conditions of the activity that correspond to (are an implementation of) the enabling of the abstract reference action.

Rules 1 to 3, presented textually above, are defined using set notation in Table 7.4, considering that $a_k \in A$ denotes the single final action. In rules 1 and 2 we do not consider a possible more abstract representation of values in the abstract reference action a.

Table 7.2: Rules for attribute distribution for single final action activities

Rule 1	*Rule 2*	*Rule 3*
$v(a) \subseteq f(a_k) = v(a_k) \cup r(a_k)$	$r(a) \subseteq r(a_k)$	$t(a) = t(a_k)$

The final action should allow the reference to all values made available by an activity and no more values, while the attributes of the abstract reference action should represent all values made available by the activity. This implies that the functionality of the final action and the functionality of the abstract reference action should be always the same. Therefore the following rule also holds:

Functionality rule: $f(a) = f(A) = f(a_k) = v(a_k) \cup r(a_k)$

The retained values of the final action a_k consist of (i) the values that have been established in the other actions of the activity and belong to the activity values, and (b) the activity retained values. In this way the retained values of the final action are a means to allow other actions and activities to refer to these values through this final action. The following rule can also be defined:

Retained values rule: $r(a_k) = (v(A) - v(a_k)) \cup r(A)$

Serial Interface Example

Consider a serial interface where four octets are established. This activity can be described as a behaviour consisting of four sequential actions $byte_1$, $byte_2$, $byte_3$ and $byte_4$, in which the octets o_1, o_2, o_3 and o_4, respectively, are established:

$$\begin{aligned}
B := \{\; & entry \text{ -> } byte_1\,(o_1\text{: } octet),\\
& byte_1\,(o_1\text{: } octet) \text{ -> } byte_2\,(o_2\text{: } octet,\ o_1\text{: } octet),\\
& byte_2\,(o_2\text{: } octet,\ o_1\text{: } octet) \text{ -> } byte_3\,(o_3\text{: } octet,\ o_2,\ o_1\text{: } octet),\\
& byte_3\,(o_3\text{: } octet,\ o_2,\ o_1\text{: } octet) \text{ -> } byte_4\,(o_4\text{: } octet,\ o_3,\ o_2,\ o_1\text{: } octet)\}
\end{aligned}$$

Action $byte_4$ is the final action of the serial interface activity, since when action $byte_4$ occurs the interface makes the four octets o_1, o_2, o_3, o_4 available. Octet o_4 is established in action $byte_4$, while octets o_1, o_2, o_3 are retained values of $byte_4$. The availability of octets o_1, o_2, o_3 is passed to $byte_4$ by the retained values of actions $byte_2$ and $byte_3$. In this way all four octets can be referred to by actions that depend on the occurrence of $byte_4$.

The sequential establishment of four octets can be represented by a single abstract reference action *word*, in which all four octets are established. The action value of *word* must be or represent the four octets o_1, o_2, o_3, o_4. These four octets may be represented in *word* as a single value of a more abstract sort, such that the four octets are just a possible codification of this value. The time of occurrence of action *word* is the time of occurrence of $byte_4$, since when $byte_4$ occurs all action values o_1, o_2, o_3 and o_4 of *word* are available. In this example we suppose that the location of actions $byte_1$, $byte_2$, $byte_3$ and $byte_4$ is the same, and therefore the same as the location of *word*.

The functionality of $byte_4$ is the complete set of the values of information established by the abstract action *word*. Since action *word* does not contain any retained values in this example, we conclude that the functionality of $byte_4$ is indeed the same as the functionality of *word*.

Figure 7.4 depicts the serial interface activity and its corresponding abstract reference action *word*.

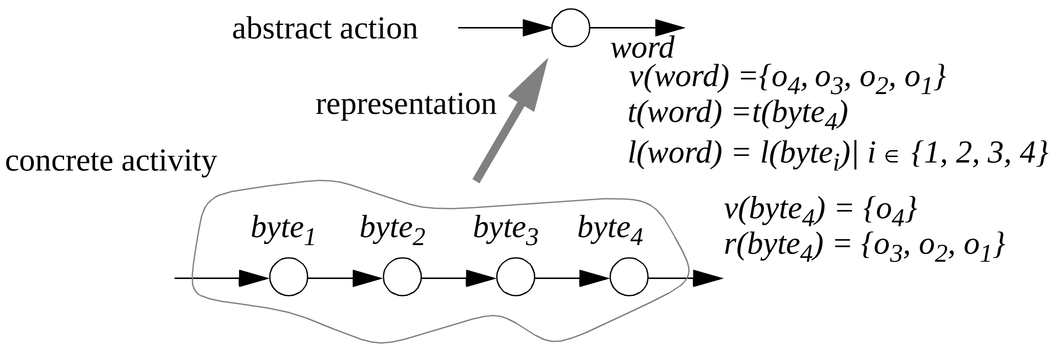


Figure 7.4: Serial interface example

The probability attribute of action *word* cannot be determined directly from the probability of *byte*$_4$. Its value should reflect the probability that all actions *byte*$_1$, *byte*$_2$, *byte*$_3$ and *byte*$_4$ happen given that the conditions that enable the abstract action *word* are satisfied. This probability can only be determined if we know how the conditions that enable *word* are implemented in terms of actions of the concrete behaviour in which the serial interface is embedded.

7.2.4 Conjunction of Final Actions

An activity may have more than one final action, such that this activity makes all its values available when all these final actions occur. Parts of the values of information made available by a final action may be established by actions that cause this final action, and made available in the retained values of this final action, in the same way as in the case of activities with a single final action discussed in section 7.2.3.

These final actions should be independent of each other, and the conjunction of their occurrences determines the completion of the activity. These final actions should not exclude each other, otherwise the activity never finishes. These final actions should not cause each other directly or indirectly either, since in this case their values of information should be passed on to a single final action by their retained values, such as in the example of Figure 7.4.

Figure 7.5 depicts an example of conjunction of final actions.

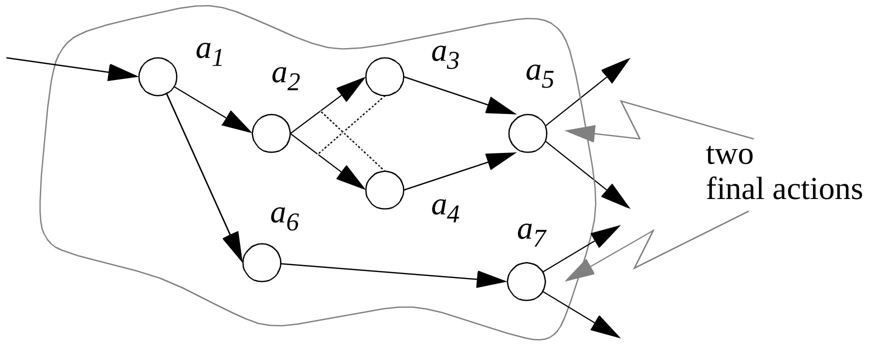


Figure 7.5: Arbitrary activity with conjunction of final actions

In Figure 7.5 both actions a_5 and a_7 have to happen in order to determine the completion of the activity. Values of information established in this activity and made available in both actions a_5 and a_7 may be referred to by other actions and activities.

Considering an activity with conjunction of final actions, such as the example in Figure 7.5, we can define more specific rules for attribute abstraction:

1. *action values*: the action values of the abstract reference action should be contained in the union of the functionality of the final actions;
2. *retained values*: the retained values of the abstract reference action should be contained in the union of the retained values of the final actions;
3. *time*: the time of the abstract reference action should be the time when all final actions have occurred;
4. *probability*: the probability attribute of the abstract reference action is the probability that all final actions occur, given that the conditions of the abstract reference action are fulfilled. This probability cannot be inferred solely from the probability attributes of the final actions, since we also need to consider the actions or activities that implement the conditions of the abstract reference action in order to calculate its probability attribute.

The final actions should allow the reference to all values made available by an activity and no other values, while the attributes of the abstract reference action should represent all values made available by the activity. This implies that the union of the functionality of the final actions should be always the same as the functionality of the abstract reference action.

Rules 1 to 3 presented textually above, are defined using simple set notation in Table 7.3. We consider that F is a sub-set of the actions of the activity, such that each $a_j \in F$ is a final action, and all final actions are in F.

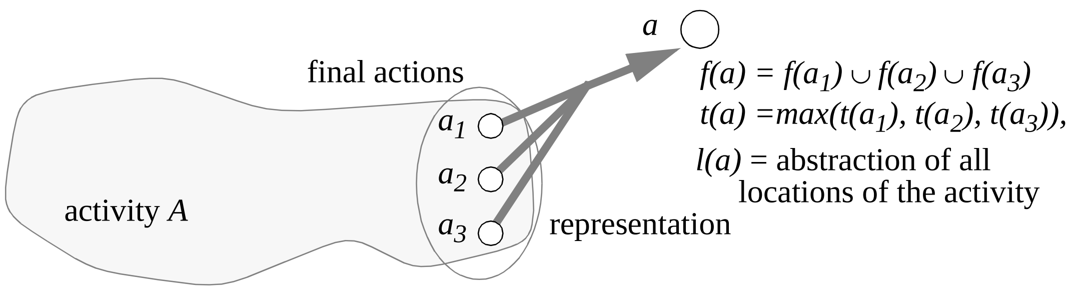
Table 7.3: Rules for attribute distribution for conjunction of final actions activities

Rule 1	*Rule 2*	*Rule 3*
$v(a) \subseteq \bigcup_{a_j \in F} \{ v(a_j) \cup r(a_j) \}$	$r(a) \subseteq \bigcup_{a_j \in F} r(a_j)$	$t(a) = max (\{t(a_j) \mid a_j \in F\})$

$$\textit{Functionality Rule: } f(a) = \bigcup_{a_j \in F} f(a_j) = \bigcup_{a_j \in F} \{ v(a_j) \cup r(a_j) \}$$

Figure 7.6 depicts an activity with conjunction of three final actions and its representation by a single action.

In the abstract reference action the activity values may be represented by a more abstract action value, and the locations of the actions of the activity may be represented by a more abstract location.



a
final actions
$f(a) = f(a_1) \cup f(a_2) \cup f(a_3)$
$t(a) = max(t(a_1), t(a_2), t(a_3))$,
a_1
a_2
a_3
$l(a)$ = abstraction of all locations of the activity
activity A
representation

Figure 7.6: Representing conjunction of final actions by an action

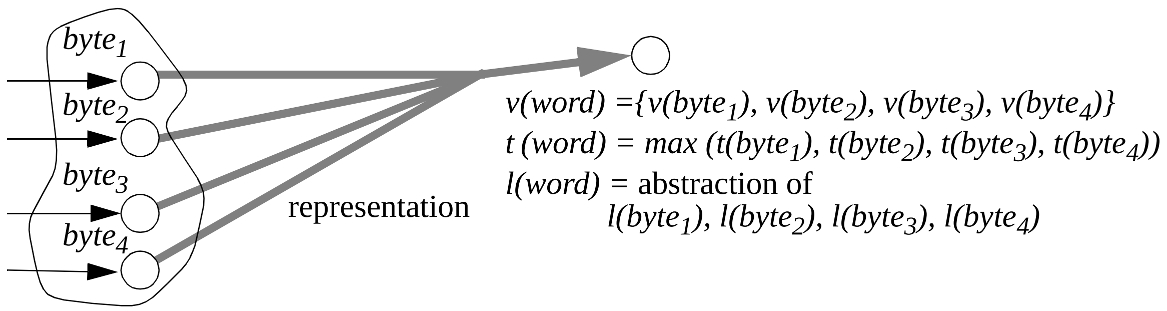
Parallel Interface Example

Consider a parallel interface where four octets are established. This activity can be described as a behaviour consisting of four independent actions $byte_1$, $byte_2$, $byte_3$ and $byte_4$, in which the octets o_1, o_2, o_3 and o_4, respectively, are established.

Actions $byte_1$, $byte_2$, $byte_3$ and $byte_4$ are final actions of this activity, and the occurrence of all of them implies that the activity has made all its values available.

This activity can be represented by an abstract reference action *word*, in which all four octets are established. The values established by action *word* are all the values made available by the final actions of the parallel interface activity. The time of occurrence of action *word* is the time of occurrence of the last action of $byte_1$, $byte_2$, $byte_3$ and $byte_4$. The location of action *word* must be an abstraction of the locations of actions $byte_1$, $byte_2$, $byte_3$ and $byte_4$. Suppose that the location of *word* is a certain action point, this action point is decomposed in (sub-)action points where each of the actions $byte_1$, $byte_2$, $byte_3$ and $byte_4$ occur. The locations of $byte_1$, $byte_2$, $byte_3$ and $byte_4$ may represent, for example, pins of a hardware interface that implements the location of action *word*.

Figure 7.7 depicts the parallel interface activity and its representation as an abstract reference action.



$byte_1$
$byte_2$
$byte_3$
$byte_4$
representation
$v(word) = \{v(byte_1), v(byte_2), v(byte_3), v(byte_4)\}$
$t(word) = max(t(byte_1), t(byte_2), t(byte_3), t(byte_4))$
$l(word)$ = abstraction of $l(byte_1), l(byte_2), l(byte_3), l(byte_4)$

Figure 7.7: Parallel interface example

Comparing the examples of Figure 7.4 and Figure 7.7 we notice that the abstract reference action in both cases establishes a word of four octets. We therefore conclude that serial and parallel interface activities are alternative implementations for such an abstract action, which corresponds to our intuitive expectations.

7.2.5 Disjunction of Final Actions

An activity may have more than one final action, such that this activity makes all its values available when one of these final actions occur. Parts of these values of information made available by a final action may be established by actions that cause this final action, and made available in the retained values of this final action, in the same way as in the case of activities with a single final action discussed in section 7.2.3.

An activity of this form has different alternative final actions, such that the disjunction of the occurrence of these final actions determines the completion of this activity. These alternative final actions may exclude each other, such that the occurrence of one of them may imply that the others do not happen, although this is not absolutely necessary. These final actions should not cause each other directly or indirectly either, since in this case their values of information should be passed on to a single final action by their retained values, such as in the example of Figure 7.4.

Figure 7.8 depicts an example of disjunction of final actions.

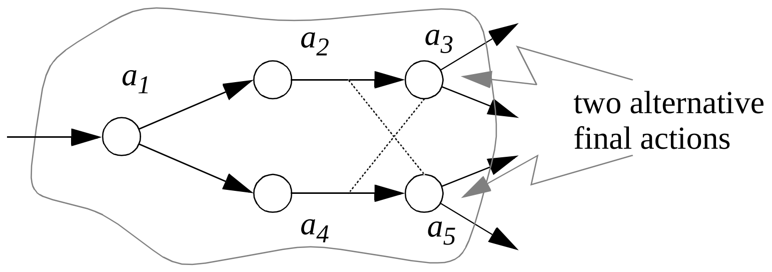


Figure 7.8: Arbitrary activity with disjunction of final actions

In Figure 7.8, actions a_3 and a_5 are alternative final actions, such that the occurrence of one of them determines the completion of the activity. Furthermore a_3 and a_5 exclude each other, which implies that only one of them occurs. Values of information made available by this activity are available at either a_3 or a_5, but not at both of them.

The conditions for considering an abstract action as a proper representation of an activity presented in section 7.2.3 can be generalized to the case of disjunction of final actions, considering that a certain actual final action determines the completion of the activity:

1. *action values*: the action values of the abstract reference action should be contained in the functionality of the actual final action;
2. *retained values*: the retained values of the abstract reference action should be contained in the retained values of the actual final action;
3. *time*: the time of the abstract reference action should be the time when the actual final action occurs. Actually it is not necessary to impose that the actual action is always the first final action to occur. In some cases, however, this may be a sensible decision, since it avoids uncertainty;

4. *probability*: the probability attribute of the abstract reference action is the probability that at least one of final actions occurs, given that the conditions of the abstract reference action are fulfilled. Again this probability cannot be inferred solely from the probability attributes of the final actions, since we also need to consider the actions or activities that implement the conditions of the abstract reference action in order to calculate its probability attribute.

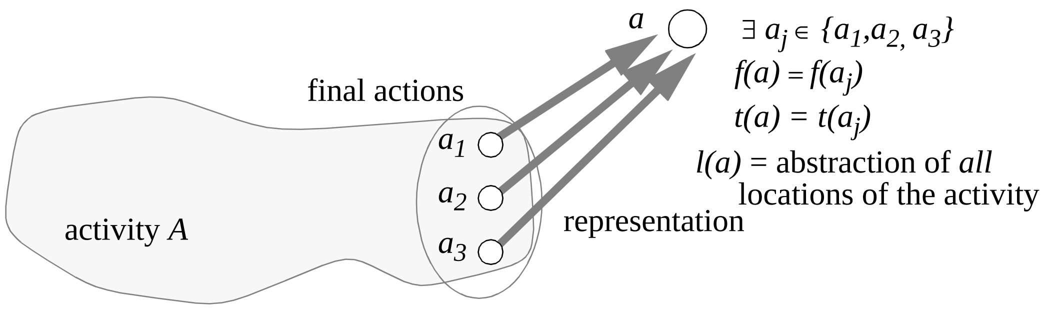
Rules 1 to 3, presented textually above, are defined using simple set notation in Table 7.4. We consider that F is a sub-set of the actions of the activity, such that each $a_j \in F$ is a final action, and all final actions are in F.

Table 7.4: Rules for attribute distribution for disjunction of final actions activities

	Rule 1	*Rule 2*	*Rule 3*
$\exists\, a_j \in F$	$v(a) \subseteq v(a_j) \cup r(a_j)$	$r(a) \subseteq r(a_j)$	$t(a) = t(a_j)$

Functionality Rule: $f(a) = f(a_j) = v(a_j) \cup r(a_j)$

Figure 7.9 depicts the disjunction of final actions and its representation by a single action.



a
$\exists\, a_j \in \{a_1, a_2, a_3\}$
$f(a) = f(a_j)$
$t(a) = t(a_j)$
$l(a)$ = abstraction of *all* locations of the activity
final actions
a_1
a_2
a_3
activity A
representation

Figure 7.9: Representing disjunction of final actions by an action

Sense and Nonsense of Abstraction

The usefulness of representing an activity of this form by an abstract reference action should be questioned. Considering the example of Figure 7.8, suppose a_3 and a_5 cause different set of actions which we denote by A_3 and A_5, respectively. The corresponding abstract reference action should also cause both sets of actions A_3 and A_5, but one may still have to define conditions in these actions to distinguish the original actions, either a_3 and a_5, to which the occurrence of the abstract reference action corresponds. Therefore the abstract reference action is an integrated representation of the activity, but the actions in the context of this abstract reference action still need to refer to two distinct actions a_3 and a_5 of the activity, such that the whole purpose of using abstraction, which is to be able to reason about the activity without considering its internal structure, is ruined. In this case abstraction makes no sense.

Another complication arises if a_3 establish values of different sorts than the values of a_5. In this case the abstract reference action should be able to establish alternative values of different sorts, which is in principle not allowed in our framework.

We conclude that although the identification of activities with disjunction of final actions follows naturally from the generalization of the possibilities for making values available in an activity, the representation of such activities by a single abstract action yields useless abstractions in some cases, and should therefore be carefully considered.

7.2.6 Generalized Activity Forms

Generalizing from the activity forms discussed so far, we can conclude that activities may make their values available by a single final action, a conjunction of final actions, a disjunction of final actions, or by a combination of the last two forms.

Figure 7.10 depicts an example of activity that makes its values available as a combination of conjunction and disjunction of final actions.

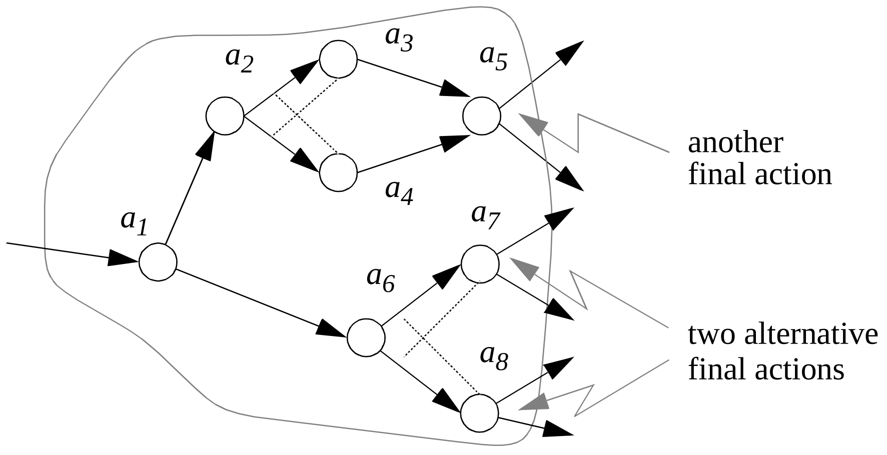


Figure 7.10: Example of generalized activity completion

In Figure 7.10 the occurrence of actions a_7 or a_8 and the occurrence of a_5 determine the completion of the activity. One can determine the attribute values of the abstract reference action that represents this activity by combining the rules presented before in this section.

7.3 Rules for Assessing Context Abstraction

Different alternative activities can be represented by a single abstract reference action, which implies that a single abstract reference action can be implemented by different alternative activities. These different alternative activities can be embedded in a concrete behaviour in different ways, but one still should be able to determine whether the embedding of the activity in the concrete behaviour conforms to the embedding of the abstract reference action in the abstract behaviour.

Similarly to Chapter 6, we assess conformance between a concrete and an abstract behaviour by applying a method for deducing the behaviour which is an abstraction of the concrete behaviour. This method starts with the definition of a reference action for each activity and for each action that is not refined. These reference actions can be defined by considering the rules for attribute abstraction presented in section 7.2. Once the reference actions have been defined, one should have rules to abstract from the remaining actions of the activities. Applying these rules one should obtain a behaviour which consists exclusively of abstract reference actions. This behaviour is the abstraction of the concrete behaviour.

The following steps define a method that can be applied in order to determine the abstraction of the concrete behaviour where activities are embedded:

1. for each activity, identify the condition, in terms of the occurrence of final actions, that correspond to the occurrence of an abstract reference action that models the completion of the activity. The negation of such a condition corresponds to the non-occurrence of this abstract reference action;
2. for each activity, integrate the causality relations of the final actions to define the causality relation of the abstract reference action that models the completion of the activity. The conditions of the abstract reference action can be possibly simplified, e.g. by replacing terms such as $a_i \wedge a_i$ and $a_i \vee a_i$ by a_i;
3. (possibly) replace the conditions defined in step 1 by their corresponding abstract reference actions in the causality relations of other actions;
4. abstract from non-final actions of activities using the abstraction rules of section 6.2;
5. (possibly) simplify the causality relations obtained, e.g. by replacing terms such as $a_i \wedge a_i$ and $a_i \vee a_i$ by a_i;
6. in case non-final actions of activities still remain, go to step 3 and proceed.

The application of this method result in a behaviour defined only in terms of abstract reference actions, such that this behaviour is an abstraction of the concrete behaviour that contains activities. The conditions that have to be defined in step 1 and the integration of causality relations of final actions that has to be performed in step 2 depend on the activity forms. This section discusses each of the activity forms identified before.

This section considers instances of action refinement in which only one action is refined at a time. Refinement of multiple actions is discussed in section 7.4.

7.3.1 Single Final Action

In the case of an activity with a single final action, the occurrence of this final action corresponds to the occurrence of the abstract reference action. The causality relation of the abstract reference action is simply the causality relation of the final action. However, according to the rules for attribute abstraction defined in section 7.2, the functionality of the abstract reference action is the same as the functionality of the final action. Action values and retained values of the abstract reference action can be determined by assessing which values of the functionality of the final action are established in the activity and which values are retained by the activity, respectively.

Some activity examples are discussed below.

Sequential Actions

Figure 7.11 depicts an action *a'* that represents an activity consisting of two sequential actions a_1 and a_2.

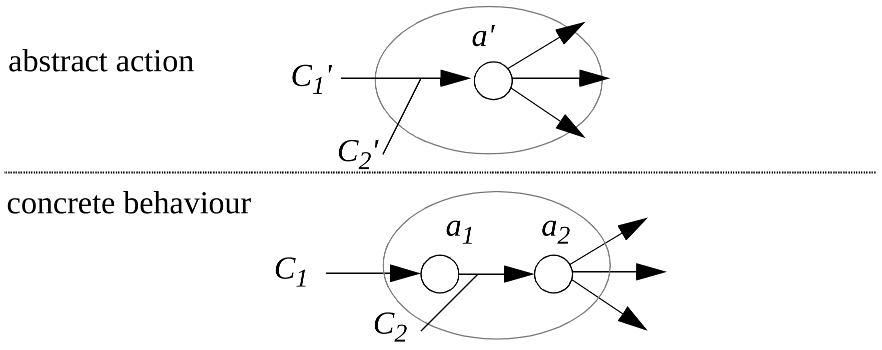


Figure 7.11: Sequential actions represented by an abstract reference action

In Figure 7.11 the concrete behaviour has two causality relations C_1 -> a_1 and $C_2 \wedge a_1$ -> a_2. According to step 1 we consider that the occurrence of a_2 corresponds to the occurrence of an abstract reference action *a*, and according to step 2 the causality relation of *a* becomes $C_2 \wedge a_1$ -> *a*. According to step 3 we replace references to a_2 by *a* in the causality relations of the other actions. According to step 4 we abstract from a_1, replacing it by its conditions, and we obtain a more abstract behaviour containing the causality relation $C_1' \wedge C_2'$ -> *a'*.

In case we only consider one replacement of an action by an activity at a time, actions outside the activity in the concrete behaviour should considered as concrete reference actions. Being a condition defined in terms of concrete reference actions, C_2 is not replaced when the elimination rules are applied, which implies that C_2 may have any form, i.e. causality, exclusion or a combination of both.

In case C_1 only contains causality, we can simply apply abstraction rule 1 and substitute a_1 by C_1 in all occurrences of a_1, and we obtain a correct abstract behaviour. In case C_1 contains exclusion, we have to evaluate which of the abstraction rules applies. Abstraction from the non-occurrence of a_1 as a condition of other actions should also be evaluated in each specific case.

In order to allow the representation of an activity consisting of two sequential actions a_1 and a_2 by a single abstract reference action, the occurrence of a_1 can only be used as a condition for an action if this action is contained in another activity that also refers to the occurrence of a_2. Since this section only addresses the refinement of a single abstract reference action as an activity, we assume here that the occurrence of a_1 cannot appear in the conditions of the concrete reference actions, otherwise no abstraction in terms of a single abstract reference action is possible. This condition is dropped in section 7.4.

Behaviours with Causality

Figure 7.12 illustrates the use of the method to obtain the abstraction of a concrete behaviour with two examples of concrete behaviour that contain an activity consisting of two sequential actions.

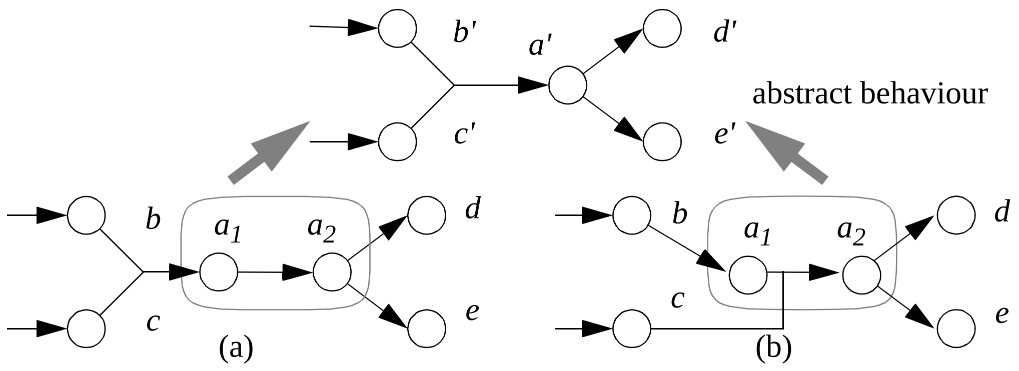


Figure 7.12: Possible refinements of an action by sequential actions

In Figure 7.12 we consider that the occurrence of a_2 corresponds to the occurrence of an abstract reference action a (step 1). The causality relation of a becomes a_1 -> a in concrete behaviour (a) and $c \wedge a_1$ -> a in concrete behaviour (b) (step 2). References to a_2 in the causality relation of d and e are replaced by references to a (step 3). According to step 4 we abstract from action a_1, replacing it by $b \wedge c$ in the concrete behaviour (a) and by b in the concrete behaviour (b). This results in the same abstract behaviour for both concrete behaviours, which means that both concrete behaviours are correct alternative implementations of the abstract behaviour.

The functionality of the final action a_2 has to be the same as the functionality of the abstract reference action a'. In the sequential action examples above, the action values of a_1 and a_2 should compose (part of) the action values of action a'. The time of action a_2 has to be the same as the time of action a'. The location of actions a_1 and a_2 should be either the same or a refinement of the location of action a'.

The probability attribute of action a_2 has to be such that the probability that a_2 happens given that actions b and c happen is the same as the probability attribute of action a', since actions b' and c' represent b and c respectively. For example suppose that the probability attribute value of a_2 in the concrete behaviour (a) is *90%*. In this case the probability attribute value of a_1 must be the probability attribute value of action a' divided by 0.9, such that the probability that a' happens given that b' and c' happen is distributed over the probability attribute values of a_1 and a_2.

Behaviours with Exclusion

Figure 7.13 illustrates the use of the method to obtain the abstraction of a concrete behaviour with three examples of concrete behaviour that contain an activity consisting of two sequential actions and exclusion conditions.

Steps 1 and 2 result in the replacement of a_2 by a reference action a. In Figure 7.13 (a) we abstract from a_1 by simply replacing a_1 by its condition *start* (*true*) in the causality relation of a. The exclusion between b and a_2 implies an exclusion between the abstract reference actions b' and a'. In Figure 7.13 (b) action b disables action a_1, but it still may happen that a_1 occurs before b occurs, such that b may not disable a_2. In this case abstraction rule 2 can be used, such that the non-occurrence of b can be simply ignored, but the probability attribute value of a' should be smaller than the probability attribute value of a_2. In Figure 7.13 (c) we can apply abstraction rule

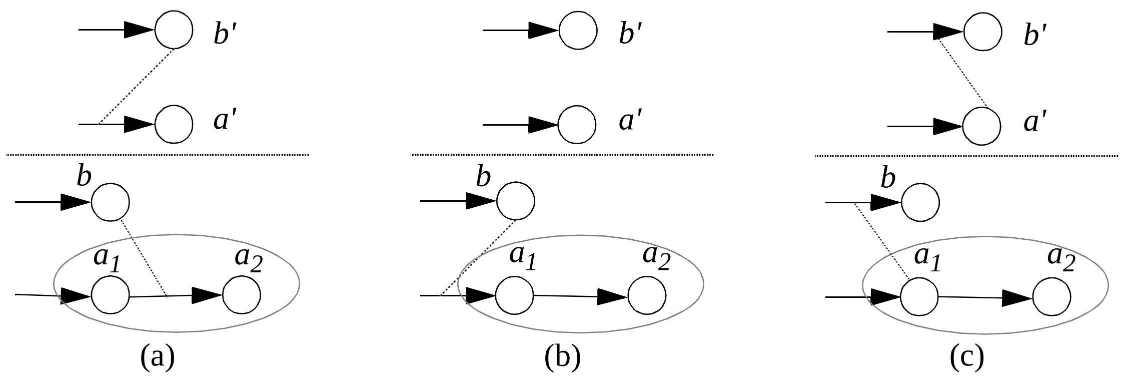


Figure 7.13: Sequential actions and exclusion conditions

3, such that the causality relation of a_2 is inverted to generate a replacement for the non-occurrence of a_1 in the causality relation of b'.

Figure 7.14 depicts three refinement alternatives for two actions in conflict, where one of the actions is replaced by an activity consisting of two sequential actions.

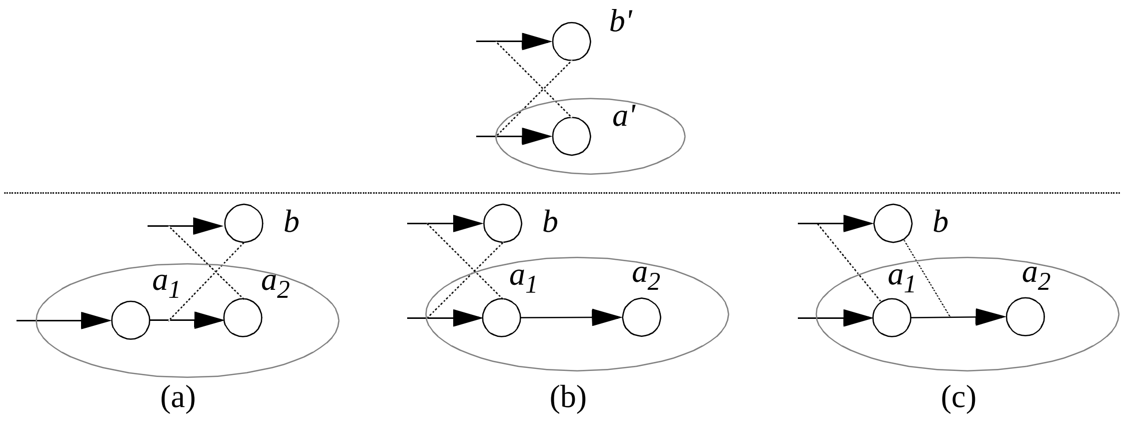


Figure 7.14: Alternative refinements of action choice

In Figure 7.14, the abstraction of concrete behaviour (a) can be obtained by applying abstraction rule 1 in step 4 to abstract from a_1, replacing it by its condition *start*. The abstraction of concrete behaviour (a) can be obtained by applying abstraction rule 4 in step 4, such that the conflict between a_1 and b is inherited by a' and b'. The abstraction of concrete behaviour (c) can be obtained by applying abstraction rule 3 in step 4, where the non-occurrence of a_1 is replaced by the non-occurrence of the reference action a in the causality relation of b.

Figure 7.15 depicts another example in which the abstraction of a concrete behaviour with causality and exclusion conditions is determined.

In Figure 7.15 we apply steps 1 and 2 similarly in the former examples. We apply abstraction rule 1 to abstract from a_1 in step 4, such that the condition a_1 s replaced by *start* in the causality relations of a and b. However this abstraction is only meaningful if one can abstract from the possible references in b to attribute values of a_1, since such references cannot be supported by the abstract

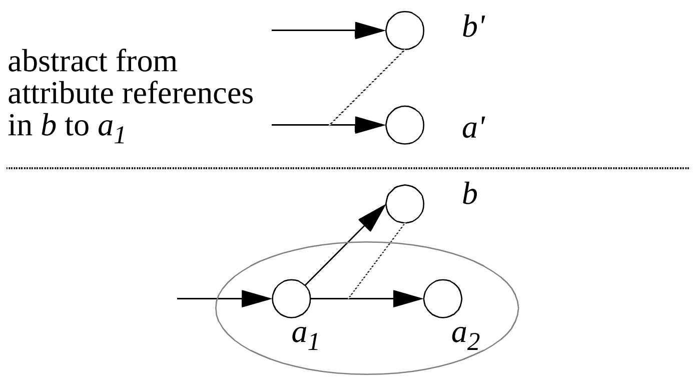


Figure 7.15: Refinement with abstraction of attribute references

behaviour, where a' does not cause b'. In case references to attributes of a_1 by b cannot be ignored this abstraction is not meaningful and should not be performed.

7.3.2 Conjunction of Final Actions

The condition that corresponds to the occurrence of the abstract reference action in case of an activity with conjunction of final actions is the occurrence of all final actions. The conditions of the abstract reference action are defined as the conjunction of all conditions of the final actions. The rules of attribute abstraction defined in section 7.2.4 also apply to the definition of the attributes of the abstract reference action that models an activity with conjunction of final actions.

Some examples of concrete behaviours that contain activities with conjunction of final actions and the deduction of the abstraction of these concrete behaviours are discussed in the sequel.

Two Independent Actions

Figure 7.16 illustrates the representation of an activity with conjunction of two independent actions by an abstract reference action.

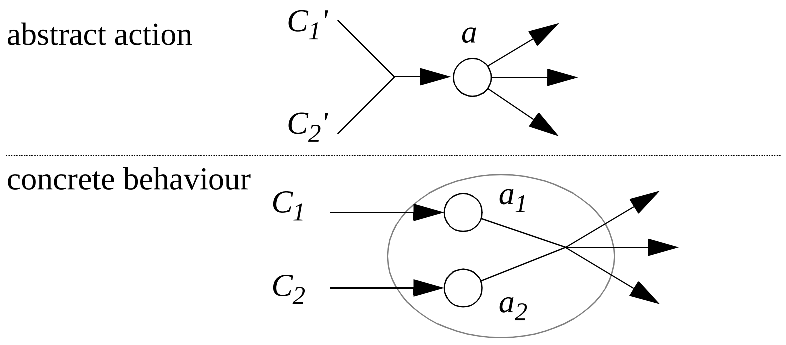


Figure 7.16: Activity with conjunction of two independent actions represented by an abstract reference action

Applying the method to determine the abstraction of the concrete behaviour in Figure 7.16 we identify in step 1 that the condition $a_1 \wedge a_2$ should correspond to the occurrence of an abstract reference action a'. In step 2 we define the conditions of the abstract reference action a' as the conjunction of the conditions of a_1 and a_2, which results in the causality relation $C_1' \wedge C_2'$ -> a', where C_1' and C_2' are the more abstract forms of C_1 and C_2, respectively.

The condition $a_1 \wedge a_2$ in the causality relation of other actions can be replaced by the occurrence of a, while reference to the occurrence of only a_1 or only a_2 may make it impossible to represent the activity as a single abstract reference action. Figure 7.17 illustrates this situation with two examples.

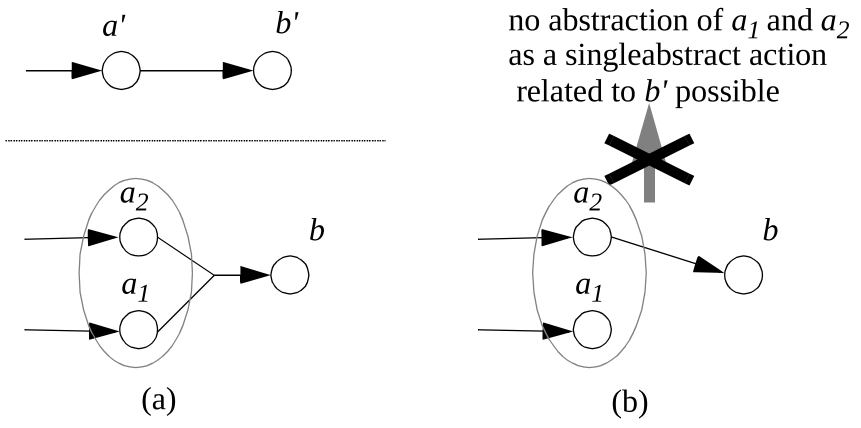

a'
b'
no abstraction of a_1 and a_2 as a singleabstract action related to b' possible
a_2
a_1
b
a_2
a_1
b
(a)
(b)

Figure 7.17: Activity consisting of independent actions as conditions for other actions

In Figure 7.17 (a) we apply steps 1 and 2 to identify $a_1 \wedge a_2$ as the condition that corresponds to an abstract reference action a and to define the causality relation of a as *start* -> a. In step 3 we replace the condition $a_1 \wedge a_2$ by a in the causality relation of b, and we obtain a more abstract behaviour in terms of a' and b'. In Figure 7.17 (b) the occurrence of a_2 in the condition of b makes it impossible to apply step 3, such that no single abstract reference action is capable of correctly represent the occurrence of both a_1 and a_2 in this example. The representation of the activity consisting of a_1 and a_2 by a single abstract reference action might be possible if action b belonged to an activity that also referred to the occurrence of a_1. Such cases are discussed in section 7.4.

Generalized Conjunction of Final Actions

Figure 7.18 depicts an more general example of activity with conjunction of final actions and its abstract behaviour deduced using the method given before.

In Figure 7.18, according to step 1, the condition that corresponds to an abstract reference action that models the completion of the activity is $a_4 \wedge a_5$. In step 2 we define the condition of a as $a_3 \wedge a_3$, which can be simplified to a_3. In step 3 the condition $a_4 \wedge a_5$ is replaced by a in the causality relation of b, yielding a more abstract behaviour. Applying abstraction rule 1 twice we can abstract from a_3 and from a_1 and a_2, obtaining a behaviour that abstracts from all non-final actions.

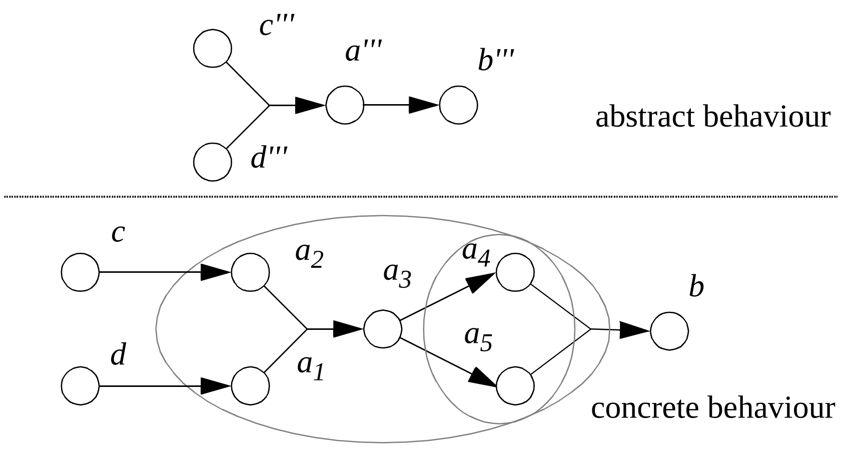


Figure 7.18: Elimination of non-Final actions

7.3.3 Disjunction of Final Actions

The condition that corresponds to the occurrence of the abstract reference action in case of an activity with conjunction of final actions is the occurrence of one of the final actions. The conditions of the abstract reference action are defined as the disjunction of the conditions of the final actions. The rules of attribute abstraction defined in section 7.2.5 also apply to the definition of the attributes of the abstract reference action that models an activity with disjunction of final actions.

Some examples of concrete behaviours that contain activities with disjunction of final actions and the deduction of the abstraction of these concrete behaviours are discussed in the sequel.

Action Choice

Figure 7.19 depicts an action a' that represents a concrete behaviour that contains an activity consisting of two actions a_1 and a_2, such that a_1 and a_2 exclude each other (action choice).

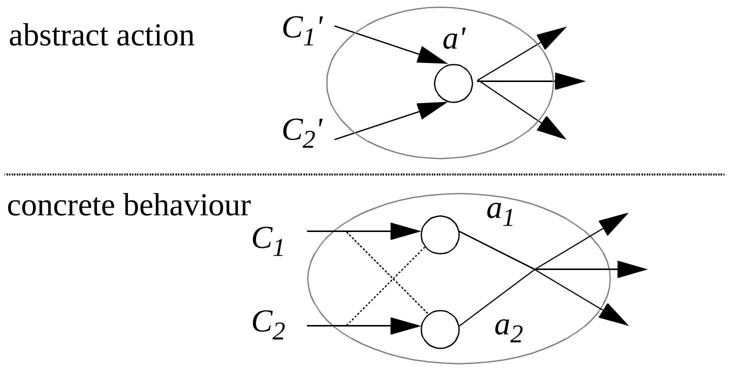
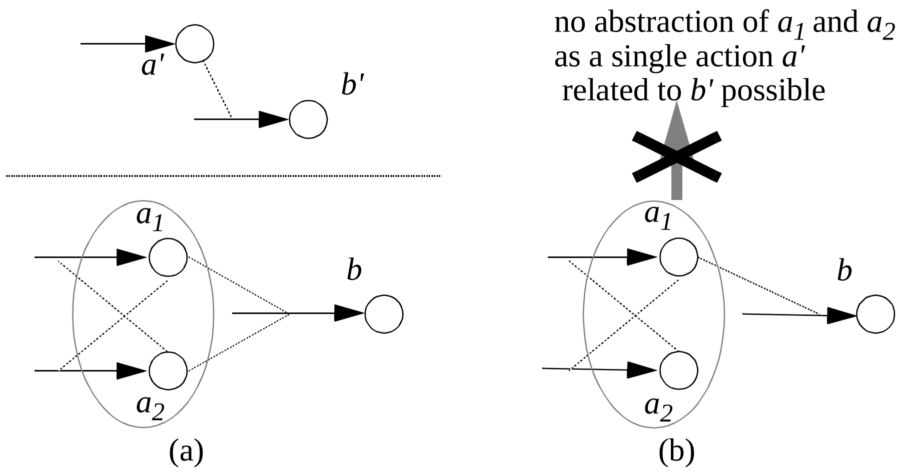


Figure 7.19: Action choice represented by an action

Applying the method to determine the abstraction of the concrete behaviour in Figure 7.19 we identify in step 1 that condition $a_1 \vee a_2$ corresponds to the occurrence of an abstract reference action *a'*. In step 2 we define the conditions of the abstract reference action *a'* as the disjunction of the conditions of a_1 and a_2, which results in the causality relation $C_1' \vee C_2'$ -> *a'*, where C_1' and C_2' are the more abstract forms of C_1 and C_2, respectively.

The condition $a_1 \vee a_2$ in the causality relation of other actions should be replaced by the occurrence of *a*, while reference to the non-occurrence of a_1 or to the non-occurrence of a_2 may make it impossible to represent the activity as a single abstract reference action. Figure 7.20 illustrates this situation with two examples.



no abstraction of a_1 and a_2 as a single action a' related to b' possible
a'
b'
a_1
a_2
b
(a)
(b)

Figure 7.20: Action choices with exclusion dependencies

In Figure 7.20 (a) we apply steps 1 and 2 to identify $a_1 \vee a_2$ as the condition that corresponds to an abstract reference action *a* and to define the causality relation of *a* as *start* -> *a*. In step 3 we replace the negation of condition $a_1 \vee a_2$, i.e. the condition $\neg a_1 \wedge \neg a_2$, by $\neg a$ in the causality relation of *b*, and we obtain a more abstract behaviour in terms of *a'* and *b'*. In Figure 7.17 (b) the non-occurrence of a_2 alone in the condition of *b* makes it impossible to apply step 3, such that no single abstract reference action is capable of correctly represent the occurrence of both a_1 and a_2 in this example.

Figure 7.21 depicts another example to illustrate the alternative concrete behaviours, which contain an activity consisting of an action choice, and their abstractions.

In Figure 7.21, condition $a_1 \vee a_2$ corresponds to an abstract reference action *a* (step 1), and the conditions of *a* (step 2) are $(b \vee c) \vee (b \vee c)$, which can be simplified to $b \vee c$ in concrete behaviour (a), and $b \vee c$ in concrete behaviour (b). In both concrete behaviours the causality relations of *d* and *e* become *a* -> *d* and *a* -> *e*, respectively, after $a_1 \vee a_2$ is replaced by *a* (step 3). A more abstract behaviour is obtained in this way.

The functionality of actions a_1 and a_2 should be the same as the functionality of action *a'*. The possible action values of action *a* can be distributed over the possible action values of actions a_1 and a_2. For example in the concrete behaviour (a) of Figure 7.21 references to values of *b* and *c* are possible for a_1 and a_2, while in the concrete behaviour (b) a_1 can only refer to values of *b* and a_2 can only refer to values of *c*. The time of actions a_1 or a_2 has to be the same as the time of

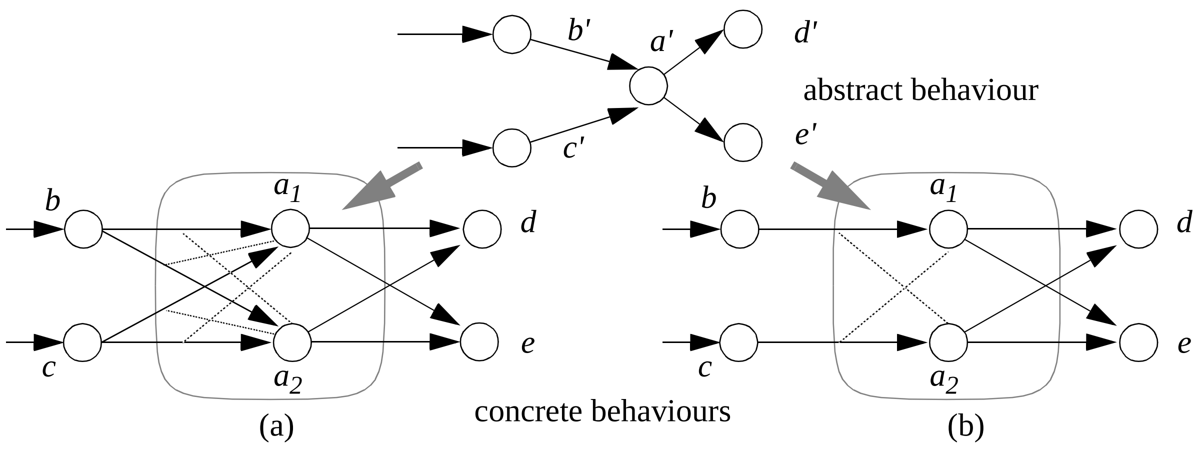


Figure 7.21: Alternative refinements of an action by action choice

action *a'*. The location of actions a_1 and a_2 should be either the same or a refinement of the location of action *a'*.

The sum of the probability attribute values of actions a_1 and a_2 should be equal to the total probability that action *a'* happens given that its conditions are satisfied. Suppose in concrete behaviour (b) of Figure 7.21 that the probability attribute of a_1 is *50%* and that the probability attribute of a_2 is also *50%*, the probability attribute of action *a* is *100%*, since in this case *a* happens either by the occurrence of a_1 or by the occurrence of a_2.

Exclusion Conditions

Consider that an action *a'* excludes another action *b'* in the abstract behaviour. In this case we can refine action *a'* by two independent action a_1 and a_2 such that the occurrence of one of them corresponds to the occurrence of action *a'* (disjunction of final actions) in two ways: (1) by making the non-occurrence of a_1 and the non-occurrence of a_2 a condition for *b*, or (2) by considering that a_1 models the occurrence of *a'* before *b'*, and a_2 models the occurrence of *a'* after *b'*. Figure 7.22 depicts these alternative refinements.

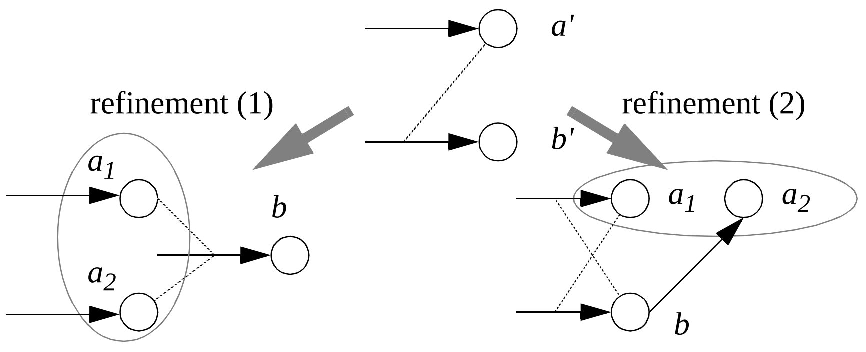


Figure 7.22: Refinements of excluding action

Applying the method for obtaining the abstract behaviour on refinement (1) condition $a_1 \vee a_2$ corresponds to an abstract reference action *a* (step 1), and the conditions of *a* are simply *start* (step

2). The condition $\neg a_1 \wedge \neg a_2$ is replaced by $\neg a$ in the causality relation of *b* (step 3), which results in a more abstract behaviour. Refinement (2) shows a limitation of our this method, since $\neg a_1$ in the condition of *b* cannot be replaced. However, refinement (2) is also a correct implementation alternative if we can abstract from possible attribute references of *b* by a_2, since in case a_1 happens *b* is not allowed to happen, and a_1 and a_2, and *b* do not happen at the same time. Furthermore there is no need to make extra assumptions on the correspondence between the occurrence of action *a'* and the occurrences of a_1 and a_2, since either a_1 happens or a_2 happens, but not both in an execution of this behaviour. Although actions a_1 and a_2 are not directly related, they actually exclude each other through action *b*.

A possible alternative method to deduce the abstract behaviour from refinement (2) is abstract from condition $\neg b$ in the causality relation of a_1, by replacing it by $\neg a_2$, in a similar way as in abstraction rule 3 (inversion of causality). Action a_2 does not happen before b, such that we can also include $\wedge \neg a_2$ in the causality relation of *b*, such that the whole condition becomes *start* $\wedge \neg a$, where *a* is the abstract reference action that models the occurrence of a_1 or a_2. Notice that inversion of causality relations only yields proper relations if no reference to attribute values is used, which is a requirement for proper abstraction mentioned above.

7.4 Multiple Activities

So far we have been discussing the refinement of one action at a time. However, the method for deducing the abstraction of a concrete behaviour has been defined in such a way that it supports the abstraction of more than one activity. Some examples of application of the method in this more general way are presented below.

7.4.1 Causality Between Activities

Consider two activities *A* and *B* consisting of sequential actions a_1 and a_2, and b_1 and b_2 respectively. Figure 7.23 depicts some alternative compositions of *A* and *B*, such that these compositions can be represented as two abstract reference actions *a'* and *b'*, respectively, where *a'* causes *b'*.

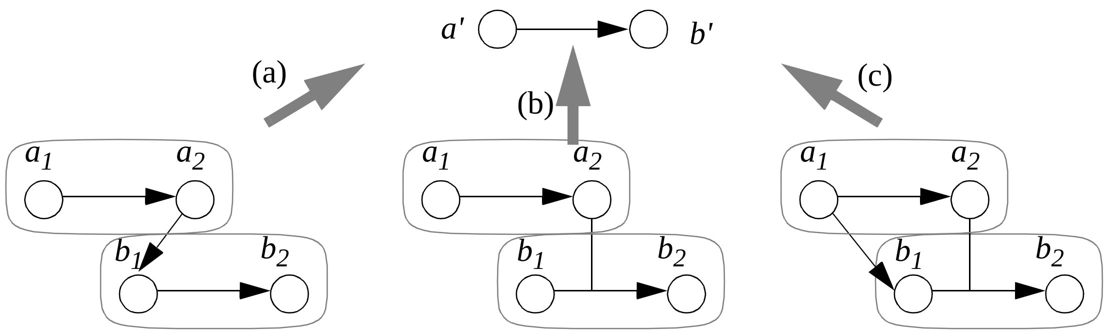


Figure 7.23: Alternative refinements of two actions in causality

Applying the method on the concrete behaviours of Figure 7.23 we identify a_2 and b_2 as the conditions for the abstract reference actions *a* and *b*, respectively (step 1 for *A* and *B*). In concrete behaviour (a) we replace a_2 by *a* in the conditions of b_1 (step 3), and we abstract from a_1 and b_1, by replacing them by their conditions (abstraction rule 1 in step 4), which results in the abstract behaviour *B':= {start -> a', a' -> b'}*. Similarly, in concrete behaviour (b) we replace a_1 and b_1 by *start* in the causality relations of *a* and *b* (step 4), which also results in behaviour *B'*. In concrete behaviour (3) we replace a_2 by *a* in the conditions of *b* (step 3), and we abstract from a_1 and b_1 by replacing them by *start*, which results again in behaviour *B'*.

We conclude that activities *A* and *B* can be correctly represented by actions *a'* and *b'*, respectively, if activity *A* always finishes before activity *B*. Considering that when an activity finishes it makes its *total* of values available, this implies that the total of values of *A* is available before the total of values of *B* is available. However, this does not prohibit some values of *B* to be available before some values of *A*.

Other alternative combinations of activities *A* and *B* cannot be represented as two sequential actions *a* and *b*, since they do not comply to the conditions above. Figure 7.24 shows an example.

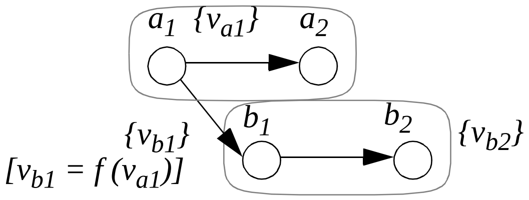


Figure 7.24: Combination of activities A and B without abstraction

In the example of Figure 7.24 it is incorrect to assume that an abstract reference action *a'* that represents activity *A* causes an abstract reference action *b'* that represents activity *B*, since the completion of *A* is completely independent of the completion of *B*. However one of the values established by *B*, namely v_{b1} is a function of value established in *A*, which characterizes value dependency between these activities. Only in case we could abstract from the dependency between the value v_{b1} of *B* and the value v_{a1} of *A* it would be correct to represent these two activities as two actions *a'* and *b'*, but these actions would then be independent of each other due to the lack of time dependencies between a_2 and b_2.

Data Transfer Service Example

Suppose the behaviour of a data transfer service is defined as follows:

$$InOut := \{start \rightarrow in,\ in\ (d_{in}: Data) \rightarrow out\ (d_{out}: Data)\ [d_{out} = d_{in}]\}$$

This behaviour can be refined by considering that *d* has three octets, b_1, b_2 and b_3, which are exchanged in two sets of three consecutive actions, in_1, in_2 and in_3, and out_1, out_2 and out_3. A possible concrete behaviour for *InOut* would then be the following:

$$\begin{aligned} InOut' := \{\quad & start \rightarrow in_1\ (b_1: Byte), \\ & in_1\ (b_1: Byte) \rightarrow out_1\ (b_1': Byte)\ [b_1 = b_1'], \\ & in_1\ (b_1: Byte) \rightarrow in_2\ (b_2: Byte, b_1: Byte), \\ & in_2\ (b_2: Byte, b_1: Byte) \wedge out_1\ (b_1': Byte) \rightarrow \\ & \qquad out_2\ (b_2': Byte, b_1': Byte)\ [b_2 = b_2'], \\ & in_2\ (b_2: Byte, b_1: Byte) \rightarrow in_3\ (b_3: Byte, b_2: Byte, b_1: Byte), \end{aligned}$$

$in_3\ (b_3\text{: Byte}, b_2\text{: Byte}, b_1\text{: Byte}) \wedge$
$out_2\ (b_2'\text{: Byte}, b_1'\text{: Byte}) \rightarrow$
$\quad out_3\ (b_3'\text{: Byte}, b_2'\text{: Byte}, b_1'\text{: Byte})\ [b_3 = b_3']\}$

Figure 7.25 depicts this refinement, without showing value constraints.

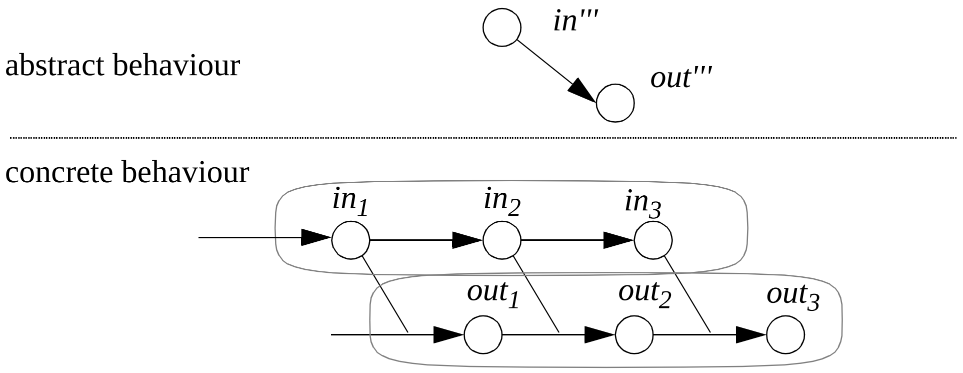


Figure 7.25: Data transfer example

In order to deduce the abstraction of this concrete behaviour we identify in_3 and out_3 with abstract reference actions *in* and *out* (step 1), replace in_3 by *in* in the condition of *out* (step 3), and obtain the following causality relations:

$in\ (b_3\text{: Byte}, b_2\text{: Byte}, b_1\text{: Byte}) \wedge out_2\ (b_2'\text{: Byte}, b_1'\text{: Byte}) \rightarrow$
$\quad out\ (b_3'\text{: Byte}, b_2'\text{: Byte}, b_1'\text{: Byte})\ [b_3 = b_3']$
$in_2\ (b_2\text{: Byte}, b_1\text{: Byte}) \rightarrow in\ (b_3\text{: Byte}, b_2\text{: Byte}, b_1\text{: Byte})$

Abstracting from the non-final actions out_2 and in_2 by replacing them by their conditions (abstraction rule 1 in step 4) we obtain the following more abstract causality relations:

$in'\ (b_3\text{: Byte}) \wedge in_2'\ (b_2\text{: Byte}, b_1\text{: Byte}) \wedge out_1'\ (b_1'\text{: Byte}) \rightarrow$
$\quad out'\ (b_3'\text{: Byte}, b_2'\text{: Byte}, b_1'\text{: Byte})\ [b_3 = b_3']\ [b_2 = b_2']$
$in_1'\ (b_1\text{: Byte}) \rightarrow in'\ (b_3\text{: Byte}, b_2\text{: Byte}, b_1\text{: Byte})$

Abstracting from in_2', out_1' and in_1' in another recursion of step 4, and from in_1'' in the next recursion, we obtain the following causality relations:

$start \rightarrow in'''\ (b_3\text{: Byte}, b_2\text{: Byte}, b_1\text{: Byte}),$
$in'''\ (b_3\text{: Byte}, b_2\text{: Byte}, b_1\text{: Byte}) \rightarrow$
$\quad out'''\ (b_3'\text{: Byte}, b_2'\text{: Byte}, b_1'\text{: Byte})\ [b_1 = b_1' \wedge b_2 = b_2' \wedge b_3 = b_3']$

Under the condition that the concatenation of b_1, b_2 and b_3 corresponds to the original data *d*, we can state that *InOut'* is a correct refinement of the original behaviour *InOut*, since the abstraction of *InOut'* complies to *InOut*. From a technical point of view this means that implementing data transfer by partitioning the data and sending different data parts separately yields a correct refinement. Action out_3 establishes b_3 and keeps b_1 and b_2 in its retained values, making it possible to refer to the whole original data *d* when this action occurs.

7.4.2 Exclusion Between Activities

Consider again two activities *A* and *B* consisting of sequential actions a_1 and a_2, and b_1 and b_2 respectively. Figure 7.26 depicts two alternative compositions of *A* and *B*, such that these compositions can be represented as two abstract reference actions *a'* and *b'*, respectively, where *a'* excludes *b'*.

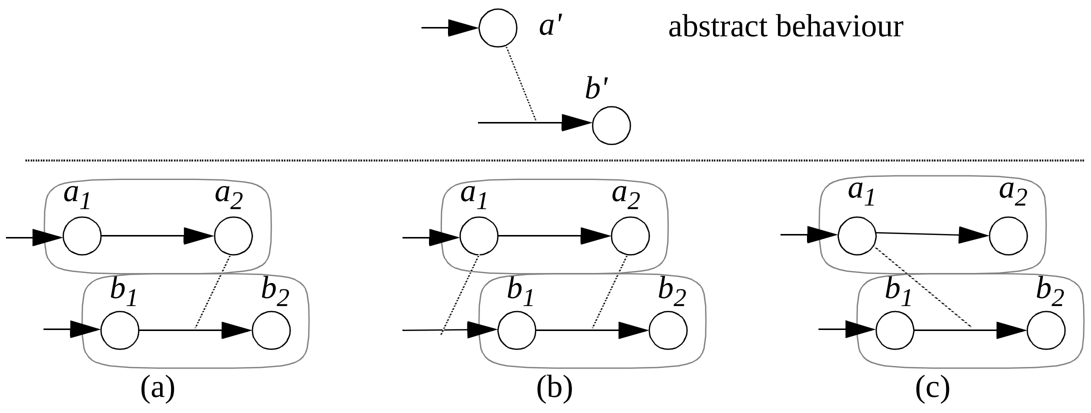


Figure 7.26: Alternative refinements of two actions in exclusion

Applying the method to determine the abstractions of the concrete behaviours of Figure 7.26, we start by identifying a_2 and b_2 as the conditions that correspond to abstract reference actions *a* and *b*, respectively (step 1). In the concrete behaviour (a) we replace $\neg a_2$ by $\neg a$ in the condition of *b* and abstract from a_1 and b_1, by replacing them by their conditions *start*. In concrete behaviour (b) we also replace $\neg a_2$ by $\neg a$ in the condition of *b* and apply abstraction rule 2 to abstract from $\neg a_1$ and b_1 in the condition of *b*. In concrete behaviour (c) we apply abstraction rule 3 to replace $\neg a_1$ by $\neg a$ in the causality relation of *b*. All three concrete behaviours have the same abstraction, which means that all these behaviours are alternative implementations of action *a'* excluding action *b'*.

Action Choice

Consider two activities *A* and *B*, each consisting of two sequential actions. Figure 7.27 shows three alternative compositions of *A* and *B*, such that these compositions can be represented as two abstract reference actions *a'* and *b'*, respectively, where *a'* and *b'* exclude each other.

The abstraction of concrete behaviour (a) in Figure 7.27 can be obtained, after steps 1 and 2, by abstracting from a_1 and b_1 using abstraction rule 1. Similarly the abstraction of concrete behaviour (b) can be obtained by applying abstraction rule 4 twice, and the abstraction of concrete behaviour (c) can be obtained by applying abstraction rule 3. All three concrete behaviour have the same abstraction.

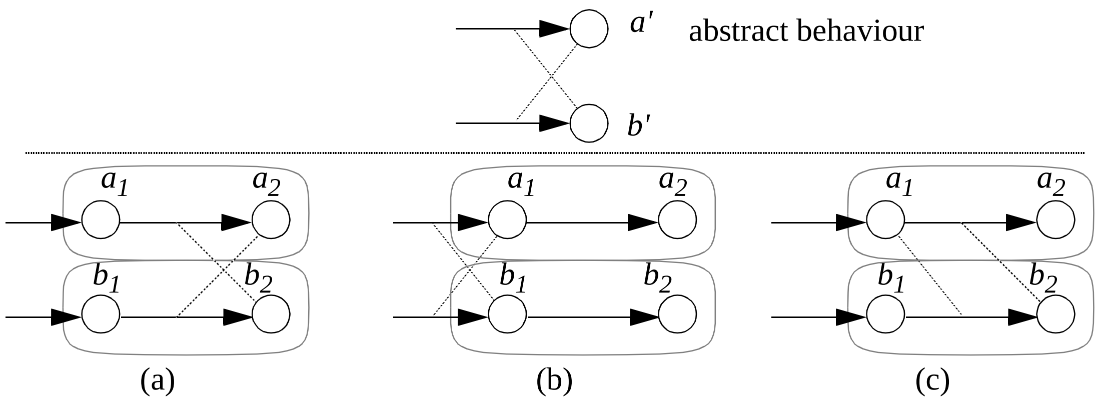


Figure 7.27: Some refinements of action choice

7.5 Interaction Point to Functional Entity

Action refinement, as defined in this chapter, can be used to support a specific design step in which an interaction point is transformed into a functional entity. Figure 7.28 depicts an example that illustrates this design step.

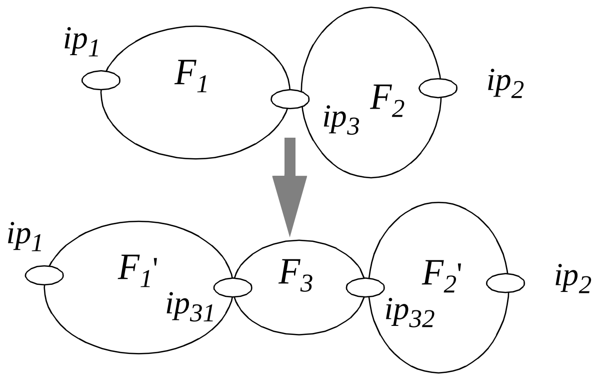


Figure 7.28: Interaction point to functional entity

In Figure 7.28, functional entities F_1 and F_2 originally share interaction point ip_3. The design step consists of replacing interaction point ip_3 by the functional entity F_3, such that the functional entities F_1' and F_2', which correspond to F_1 and F_2, respectively, are not directly connected through ip_3 any more, but are both connected to F_3 through interaction points ip_{31} and ip_{32}, respectively. Functional entity F_3 performs the role of an interface entity, supporting the operation of the original interaction point ip_3.

In the behaviour domain we should replace all interactions that occur at ip_3 by activities represented in terms of interactions occurring at interaction points ip_{31} and ip_{32}. The rules for considering an abstract reference action as a proper abstraction of an activity, and the rules for considering an embedded abstract reference action as an abstraction of an embedded activity which have been discussed before should be applied in the determination of behaviours for F_1', F_2' and F_3 that conform to the original behaviours of F_1 and F_2. Particularly the rule that the location of an abstract reference action is a representation of the locations of the activity applies here, since ip_3 can be considered as an abstract representation of the more concrete interaction points ip_{31} and ip_{32}.

Example

We illustrate the design step in which an interaction point is transformed into a functional entity with an example which has been also presented in [1]. This example consists of two entities A and B which communicate directly, such that they establish a connection, exchange some units of data, and disconnect.

Figure 7.29 depicts these two functional entities and their common behaviour. The common behaviour of the entities A and B corresponds to the behaviour of the interaction service composed by these two entities.

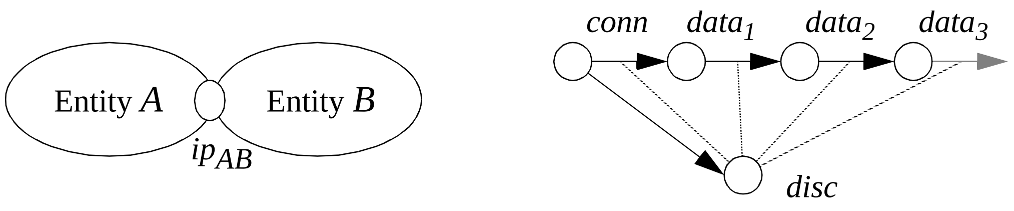


Figure 7.29: Directly interacting entities and their common behaviour

The design step to be illustrated consists of making the direct interaction between entities A and B be performed indirectly through an interface entity. In this case interaction point ip_{AB} becomes the interface entity, and directly interacting entities A and B are replaced by indirectly interacting entities A' and B'. In order to correctly perform this design step we have to refine the actions of the common behaviour between entities A and B, by replacing each abstract action by an activity, which defines the common behaviour between the interface entity and the entities A' and B'.

Figure 7.30 shows the structure of functional entities to be obtained in this design step.

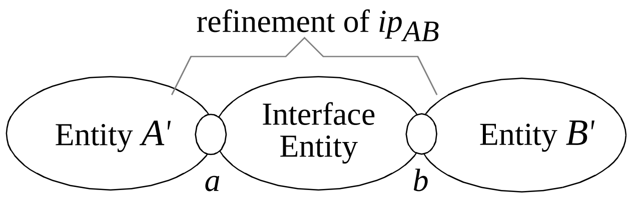


Figure 7.30: Structure of functional entities after distribution

The behaviour of the interface entity resembles the behaviour of a connection-oriented service. We make some assumptions meant to simplify the concrete behaviour. These assumptions are:

- entity A' is a calling user, which initiates connection establishment;
- entity B' is a called user, which accepts or rejects the requested connection;
- only entity A' is allowed to send data, which is received by entity B;
- only entity A' is allowed to start a disconnection activity;

- disconnection generated by the interface entity (provider generated disconnect in the literature) is not supported.

Figure 7.31 depicts the common behaviour between the interface entity and the entities A' and B' which can be obtained under these assumptions. In Figure 7.31 we do not represent value attribute dependencies between actions, but for each action we indicate in which (inter)action point it is supposed to happen.

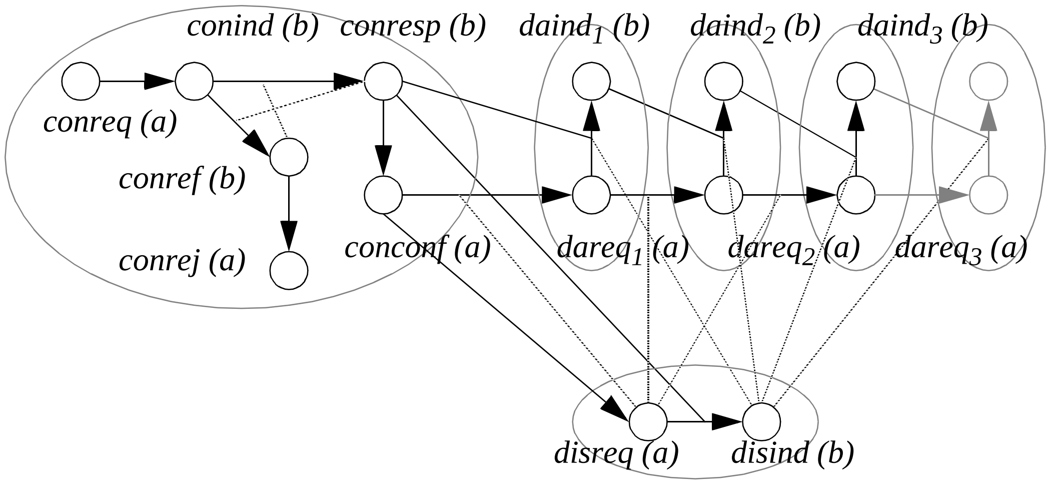


Figure 7.31: Connection-oriented behaviour

Each of the abstract reference actions of Figure 7.29 is replaced by an activity in Figure 7.31. The activity that replaces *conn* contains six actions *conreq*, *conind*, *conresp*, *conconf*, *conref* and *conrej*. The first four actions correspond to the connection establishment service primitives, namely connection request, indication, response and confirm, respectively. The occurrence of *conconf* determines the completion of the connection establishment activity. Actions *conref* and *conrej* stand for connection refusal and reject, respectively. These actions model the fact that the called entity may decide to refuse the connection establishment, which is informed to the calling entity. In most connection-oriented services, connection refusal and reject are performed using disconnect request and indication primitives, respectively. The activities that replace $data_i$ consist of two actions $dareq_i$ and $daind_i$. These activities reach completion when $daind_i$ happen. The activity that replaces abstract action *disc* consists of two actions *disreq* and *disind*. This activity reaches completion when *disind* happen.

The complete demonstration that the behaviour depicted in Figure 7.29 is an abstraction of the behaviour of Figure 7.31 is out of the scope of this example. An intuitive indication that this is the case can be obtained by considering the relationships between the abstract reference actions and how they are implemented in the concrete behaviour. For example *conn* is a condition for *disc* in the abstract behaviour, while in the concrete behaviour *conresp* is a condition for *disind* and *conconf* is a condition for *disreq*. The latter pattern of behaviour is similar to the one shown in Figure 7.23 (c), in which the activities are modelled as two abstract reference actions that cause each other. The causality relations between *conn* and $data_1$, $data_1$ and $data_2$, etc. are also implemented by similar patterns. The disabling of $data_i$ by *disc* is implemented by a behaviour pattern that resembles the example of Figure 7.26 (b), in which the activities could be represented as actions such that one action disables the other.

7.6 References

[1] J. Schot. *The role of Architectural Semantics in the formal approach towards Distributed Systems Design*. PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

Chapter 8

Design of an Interaction Server

This chapter discusses the high-level design of an *interaction server*, which is a general purpose component that supports complex interactions between multiple functional entities. The objectives of this chapter are twofold: to illustrate the use of our design model in a realistic instance of design, and to illustrate the use of pre-defined implementation constructs as an approach towards distributed systems development.

This chapter is structured as follows: section 8.1 discusses the global design choices taken in the development of an interaction server, section 8.2 discusses the support of causality relations by the interaction server, section 8.3 presents the behaviour of a specific interaction server and section 8.4 discusses alternatives for protocols that support a distributed interaction server.

8.1 Global Design Choices

An implementation environment that aims at supporting the implementation of specifications based on entity and behaviour domains has to cope with two important aspects: (i) *entity management*, which consists of creation and deletion of functional entities and (ii) *interaction management*, which consists of interaction scheduling, according to the reliability and atomicity characteristics of interactions.

The interaction server is a general purpose component meant to perform interaction management. The interaction server can be a relevant component in an implementation environment that supports the implementation of specifications using our design model, since it allows one to quickly obtain prototypes of interacting functional entities, without the burden of manipulating the behaviour specifications of these functional entities in order to remove those concepts or constructs that may not have a direct mapping onto concepts or constructs of the available implementation environment.

Some of the aspects to be considered in the development of the interaction server are:

- *support of data types*: implementation of the data values exchanged in interactions are supposed to be available. We abstract from the various ways these implementations can be obtained;
- *generality*: too general components may become inefficient or difficult to implement, or

both. A proper compromise has to be found for the interaction server;

- *correctness*: the behaviour supported by the interaction server has to comply to the behaviour of the interacting functional entities.

Other aspects are neglected for the time being, but are expected to be of importance for the implementation the interaction server:

- *level of parallelism*: functional entities can be implemented as sequential processes of interleaved actions or as a set of parallel threads. Concurrence may exist between different functional entities in a concurrent implementation environment;
- *support to value negotiation*: value negotiation may be hard to implemented directly, since it implies the implementation of functions that: (i) determine allowed data type value sets by superposing all constraints on these values, and (ii) perform a non-deterministic choice of values from these sets. An example of a technique for function (i) is narrowing, which has been implemented in the LOTOS simulator SMILE ([4]);
- *distribution aspects*: the functional entities which interact through the interaction server can in principle be placed in different computer systems or not. An interaction server for interacting functional entities in a single computer system can support the rapid generation of software prototypes, while an interaction server for distributed functional entities needs to be implemented by a proper protocol.

8.1.1 Interaction Structure

Entity management and interaction management are related to each other. Their relation can be represented by their common knowledge on which functional entities interact and through which interactions. This knowledge is called the *interaction structure*.

The interaction structure relates sets of functional entities to interactions, determining the static conditions for interaction. Specific conditions on values of information and timing, and the specific states of the functional entities determine whether they actually interact. These are the dynamic conditions for interaction.

Figure 8.1 depicts an example of three functional entities F_1, F_2 and F_3, and interactions *a*, *b* and *c*. We suppose that F_1, F_2 and F_3 can possibly interact through *a*, F_1 and F_3 can possibly interact through *b*, and F_1 and F_2 can possibly interact through *c*. The interaction structure should relate the sets $\{F_1, F_2, F_3\}$, $\{F_1, F_3\}$ and $\{F_1, F_2\}$ to *a*, *b* and *c*, respectively.

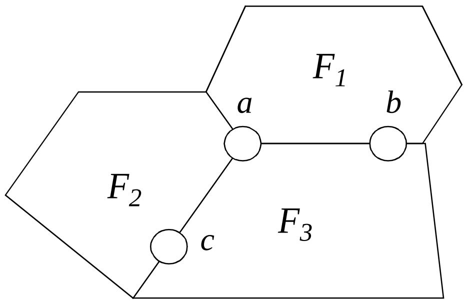


Figure 8.1: Interacting functional entities

We focus on the design of the interaction management component, considering the interconnection structure itself available, but abstracting from the functions that make it available. In this way we abstract from creation and deletion of functional entities and updating of the interaction structure. The definition of an entity management component and the specific ways it updates the interaction structure are not discussed in this design example.

Example

In LOTOS, some processes may correspond to functional entities. Gate definitions in a process and the behaviour expression in which processes are instantiated determine the interaction structure of a specification. Taking the example in Figure 8.1, we could make *a*, *b* and *c* correspond to LOTOS gates. The behaviour expression `(F1[a, b] |[a, b]| F2 [a, c]) || F3 [a, b, c]` represents the interaction possibilities defined above. The interacting processes in LOTOS can therefore be determined by inspecting the behaviour expressions in which these processes are instantiated. This also means that the interaction structure of a LOTOS specification can be determined by some automated algorithm on the specification syntax. This approach has been applied in [3].

8.1.2 Role of Interaction Server

We assume that, in order to interact with other functional entities at some instant in time, each entity has a set of interactions that can be performed. Each interaction in this set may have specific time and value constraints. This set of interactions is possibly modified each time interactions of the set take place.

A functional entity alone cannot decide on the occurrence of an interaction, since the occurrence of an interaction is determined by a functional entity in cooperation with its environment. The environment of an entity consists of other functional entities which participate in interactions with this entity. Since each functional entity in a certain context cannot decide alone on the occurrence of an interaction, it is feasible to define a general purpose component, the interaction server, to cope with the agreement between entities on the execution of interactions. Interactions between functional entities at a certain abstraction level can be grouped and handled by the interaction server. The main function of the interaction server is to guarantee that an interaction only occurs if all functional entities which are allowed to participate in this interaction are wishing to perform it.

Figure 8.2 illustrates the introduction of an interaction server to support the interaction between functional entities F_1, F_2 and F_3.

The service supported by the interaction server defines an interaction system between the original interacting functional entities, and, consequently, it also implements part of these functional entities. This implies that the original functional entities should be transformed in order to interact with each other through the interaction server. In Figure 8.2, F_1, F_2 and F_3 are transformed into F_1', F_2' and F_3', respectively. Transformed functional entities that support their indirect interactions through direct interactions with the interaction server are called *i-entities* (implementation-entities) from now on, in order to avoid misunderstanding. The interaction server can be considered a service provider in the sense of [5], since it interacts directly with its users, the i-entities.

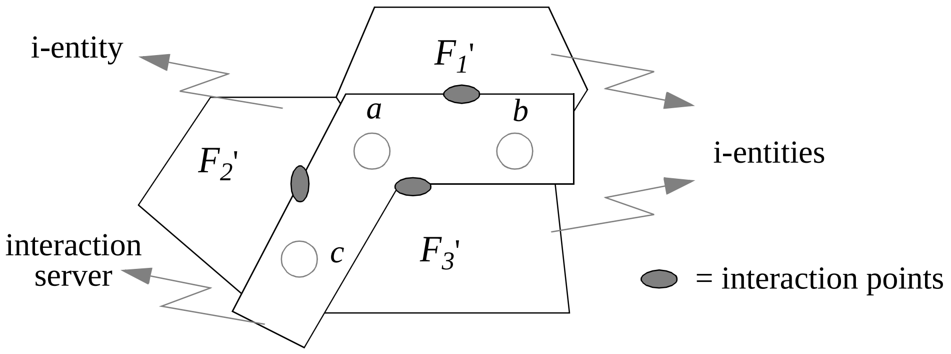


Figure 8.2: Interaction server

A functional entity should not know the structure of its environment in the terms of compositions of functional entities, and should not be able to decide alone on the occurrence of interactions. Therefore the interaction server should have the information on which functional entities exist and how they are interconnected, which correspond to the interaction structure. This is also pictorially shown in Figure 8.2, since the border line between functional entities falls inside the interaction server.

8.2 Behaviour Assumptions

According to our design model, a structure of interacting functional entities actually represents a specific assignment of responsibilities and constraints of a common behaviour to these functional entities. The interaction server can handle part of these responsibilities and constraints. However, depending on the specific choices of responsibility and constraints assigned to the interaction server, we may get specific behaviours for the interaction server and for the i-entities. These choices are investigated in this section.

8.2.1 Conditions and Constraints

In the definition of the interactions between the original functional entities we abstract from the specific ways in which conditions and constraints of each specific functional entity are settled. Conditions on attribute values of interactions that have already occurred can be better handled by the i-entities, since the i-entities have all the information necessary to handle these conditions. Constraints, however, must be superposed at a single place, in order to enable the generation of a single result that conforms to all constraints. Considering that i-entities should be symmetric, we conclude that all constraints should be informed to the interaction server, since this is the only component capable of handling all of them in a general way.

Figure 8.3 depicts an example of distribution of constraints.

In Figure 8.3, the condition $v_1 < 5$ on the value attribute of a_1 is assigned to F_1, and can be handled by its corresponding i-entity, which means that F_1' should only take the initiative to perform a_2 if a_1 has happened with $v_1 < 5$. However, F_1' cannot decide alone on the value of v_2 that complies to all constraints, since F_1' should not know the constraints of other functional entities.

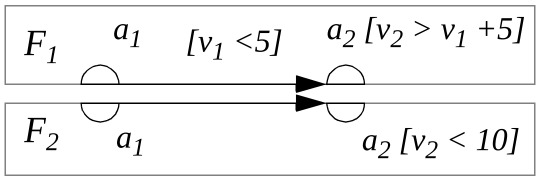


Figure 8.3: Distribution of constraints

F_1 does not know that F_2 imposes that v_2 should be smaller than *10*. Suppose, alternatively, that all functional entities would inform one of them of their constraints. This would yield an asymmetric solution with respect to the behaviour of the i-entities and therefore this alternative is discarded. A general solution to this problem is to assign the selection of values according to all constraints to the interaction server. Applied on this example, the interaction server should get all constraints ($[v_2 > v_1 +5]$ and $[v_2 < 10]$), select a single value if it exists, and inform this value to both functional entities.

8.2.2 Interactions with the Interaction Server

A consequence of treating constraints in the interaction server is that the original direct interactions between functional entities have to be split up in at least two interactions for each i-entity: a *request* for interaction (*intreq*), in which all constraints of the functional entity on the interaction are informed to the interaction server, and a corresponding *confirm* (*intconf*), in which a result that complies to the constraints of all functional entities is informed to this functional entity.

Figure 8.4 depicts an example of introduction of an interaction server for a sequential common behaviour, where each functional entity has a single allowed interaction to each time.

In Figure 8.4, i-entities F_1' and F_2' are both responsible for the causality relations between a_1 and a_2, and a_2 and a_3. The interaction server merely makes the matching between these two i-entities, by collecting their *intreq*s and issuing the corresponding *intconf*s. In this example, the interaction server does not need to know that a_1 causes a_2, and that a_2 causes a_3, but it knows that F_1' and F_2' interact through actions a_1, a_2 and a_3.

Figure 8.4 shows the operation of the interaction server to support interactions between two functional entities. However, the behaviour of the interaction server can be defined in such a way that interactions between more than two functional entities, which characterizes multi-way synchronization, can also be supported.

Figure 8.5 depicts an example where the interaction server supports the interaction of three functional entities.

In Figure 8.5, functional entities F_1, F_2 and F_3 participate in a_1. Introducing the interaction server to support this interaction implies that all three i-entities F_1', F_2' and F_3' have to request interaction a_1, before it can be confirmed to all these i-entities. The behaviour of the interaction server defined in this way is general enough to support the interaction between an arbitrary number of functional entities.

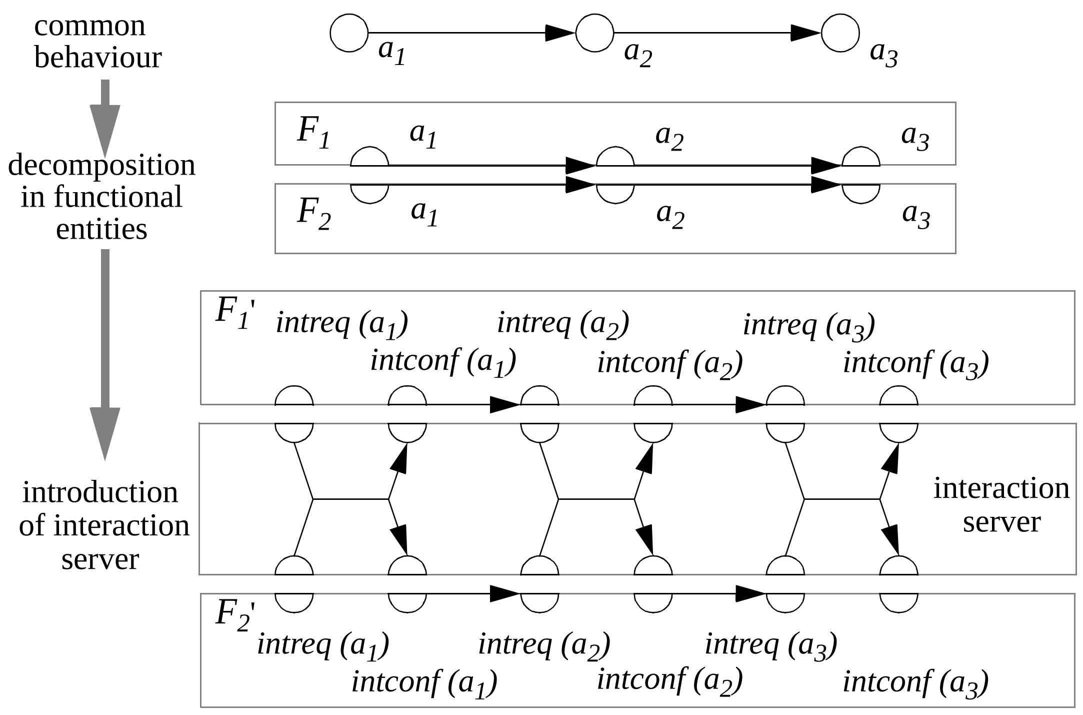


Figure 8.4: Introduction of an interaction server

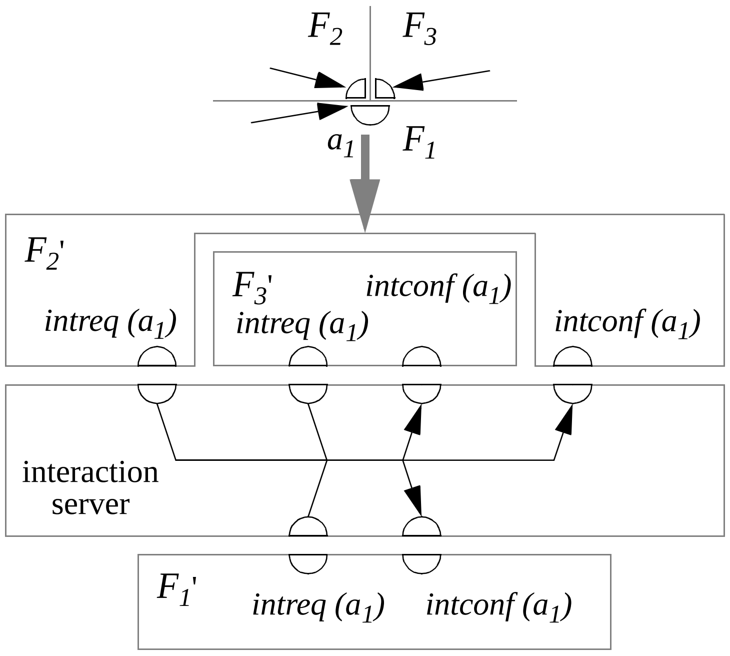


Figure 8.5: Multi-way synchronization supported by the interaction server

Interaction Withdrawal

An important consideration is whether an i-entity should be allowed to withdraw an interaction that it has requested and not yet confirmed. Allowing an i-entity to withdraw interaction requests gives no guarantee that the execution of the interaction is aborted, since it may happen, for example, that an withdrawn interaction has already been confirmed to other i-entities. In case the abortion of an withdrawn interaction is mandatory, roll-back mechanisms may be necessary to erase withdrawn interactions from the history of execution, even if they have already been confirmed to other i-entities. These mechanisms would make the service supported by the interaction server substantially more complicated.

In case interaction withdrawn is supported, some more interactions between the i-entities and the interaction server are necessary, for instance, to allow i-entities to inform the interaction server that they wish to withdraw interaction requests. Interaction withdrawal is not supported in this design example. The consequences of this design decision on the support of timeout conditions and with respect to interactions that are not treated by the interaction server are discussed later on.

8.2.3 Functional Requirements

An interaction between functional entities either happens for all involved entities with the same results, or it does not happen at all. This implies that some functional requirements have to be satisfied by the interaction server:

1. an *intconf* for an interaction may only occur if the *intreq*s of all i-entities that participate in the interaction have occurred (necessary condition);
2. in case one *intconf* occurs, all its corresponding *intconf*s for the other i-entities must occur with the same established values (reliability condition).

In Figure 8.5, for example, requirement 1 should be fulfilled by the causality relations between all *intreq*s and each *intconf*s. Requirement 2 should be fulfilled by the probability attribute of each *intconf* and by assigning the same values attributes to all *intconf*s. For example in Figure 8.5, in case an $intconf(a_1)$ occurs for F_1' it must also occur for F_2' and F_3'. The occurrence of all three *intconf*s should correspond to the occurrence of the original common action a_1.

Global Ordering versus Local Ordering

The correctness of the replacement of each individual interaction by *intreq*s and corresponding *intconf*s could be assessed by considering the interaction in its integrated form, abstracting from the responsibilities of the functional entities. In this way one takes an action, with all the constraints of the individual functional entities, and replace it by an activity consisting of a pattern of *intreq*s and *intconf*s, where *intreq*s occur before *intconf*s for all participating i-entities.

Figure 8.6 depicts the introduction of an interaction server from the point of view of the replacement of an action (integrated interaction) by an interaction server activity involving four i-entities.

The replacement of an action by an activity depicted in Figure 8.6 resembles the case of action refinement by an activity with conjunction of final actions discussed in section 7.2.4. However, the condition that the occurrence of all *intconf*s corresponds to the occurrence of the abstract action, combined with the condition that the temporal ordering imposed by the causality relations of the original common behaviour has to be preserved, may be too severe in some cases.

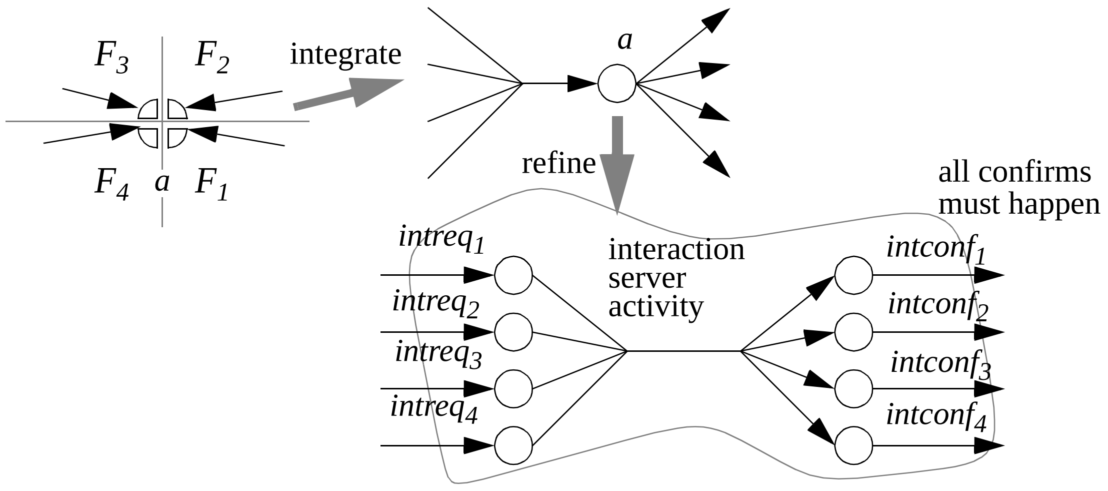


Figure 8.6: Replacement of an integrated interaction by an interaction server activity

Figure 8.7 shows an example where an implementation obtained with the interaction server that should be considered correct does not satisfy the condition of global preservation of temporal ordering.

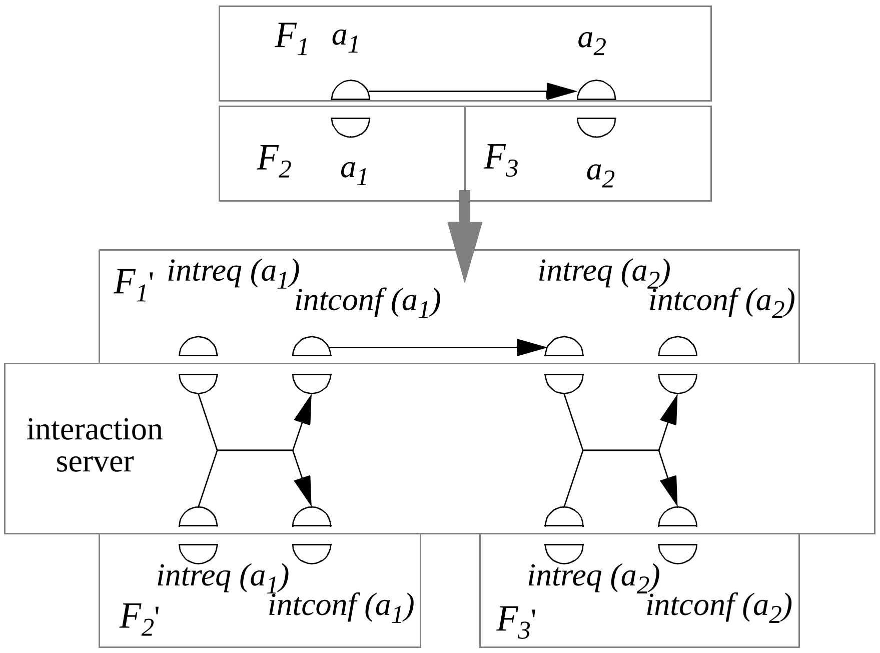


Figure 8.7: Possible non-conforming global ordering

In Figure 8.7, there are three functional entities F_1, F_2 and F_3, where F_1 and F_2 participate in a_1 and F_1 and F_3 participate in a_2. F_1 is responsible for the causality between a_1 and a_2, and F_2 and F_3 do not know of the existence of a_2 and a_1, respectively. In this case, *intconf* (a_1) at F_2' is completely independent of *intconf* (a_2) at F_3', which implies that they may happen in any order. Since they may be the last occurring *intconf*s of a_1 and a_2, considering their occurrence for determining the correctness of this implementation using the interaction server brings the immediate conclusion that this implementation is incorrect.

However, if we consider a more relaxed correctness requirement, namely that for each functional entity the local ordering between interactions imposed by the causality relations assigned to this functional entity is preserved by the *intconf*s of these interactions, this implementation could be considered correct. This more relaxed correctness requirement should only be acceptable if we can guarantee that it does not allow undesirable implementations. Intuitively the fact that globally the ordering of the last *intconf*s of some interactions may not correspond to the ordering of the original actions should only be a problem if the ordering differs at a functional entity that makes use of this ordering. For example in Figure 8.7, in case there is such a functional entity, an *intreq* for a_2 at this functional entity would only be performed after the *intconf* for a_1, and the local ordering would then be correct. The decision to ignore interaction withdrawal also implies that once an interaction has been confirmed somewhere else, it cannot be refused by the other participating functional entities, which implies that although the global ordering of last *intconf*s may not coincide with the original ordering, the local ordering at the functional entities that participate in the interactions is always preserved.

The interaction server activity has always the same behaviour pattern, which means that when the interaction server is introduced, all interactions, from their integrated perspective, are replaced by the same sort of activity. In this way it is enough to determine for which interactions this replacement is correct, and use the interaction server as a general purpose component to support the implementation of these interactions. In this sense the interaction server is considered as a predefined implementation construct.

The interaction server is responsible for the establishment of an interaction, but it may be also necessary to assign some responsibility on the relationships between interactions to the interaction server. In Figure 8.7, for example the causality relation between a_1 and a_2 has been assigned to F_1. I-entity F_1' relates *intconf* (a_1) to *intreq* (a_2), possibly by also relating attribute values of a_2 to attribute values of a_1. The interaction server does not contribute to the establishment of this particular relationship.

8.2.4 Multiple Possible Interactions

A general purpose interaction server must also be able to support interactions between functional entities that may have more than one interaction to be requested at a certain moment in time. We suppose that no interaction should be favoured beforehand, and that the i-entities are symmetric. This means that solutions in which an i-entity would go through a list of possible interactions, making requests and receiving positive or negative *intconf*s, or in which values are established at a certain previously stipulated i-entity should be discarded.

We assume that i-entities inform the interaction server of all their allowed interactions, and the interaction server sorts out which interactions take place. Parts of the causality relations between interactions may also be supported by the interaction server.

Figure 8.8 depicts an example of a choice between two actions a_1 and a_2. This example is considered for the evaluation of the degrees of freedom in assigning causality relations to the interaction server.

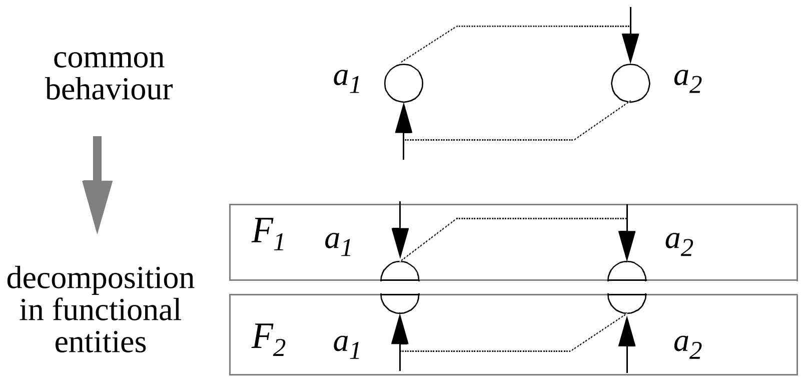


Figure 8.8: Action choice and assignment of conditions to functional entities

Suppose we introduce an interaction server to support the execution of the interactions in the example of Figure 8.8, such that the responsibility for the choice between interactions is assigned to the i-entities. This implies that the i-entities should inform the interaction server that interactions a_1 and a_2 can both occur, but for the interaction server these interactions would be considered independent of each other.

Figure 8.9 depicts a possible consequence of this assumption.

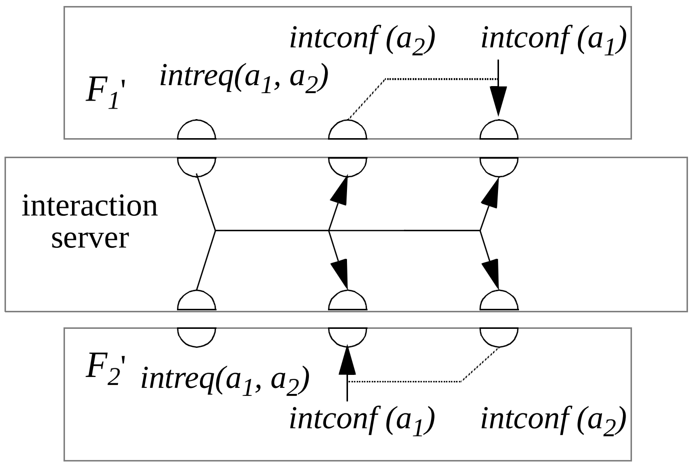


Figure 8.9: Incorrect implementation of choice between interactions

Condition *start* $\wedge \neg a_1 \to a_2$, which has been assigned to F_1, could be applied in F_1' to force that *intconf* (a_2) happens before *intconf* (a_1), since *intconf* (a_1) and *intconf* (a_2) mean the occurrence of a_1 and a_2, respectively, to F_1'. For similar reasons *intconf* (a_1) would happen before *intconf* (a_2) in F_2'. The consequence is that *intconf* (a_1) and *intconf* (a_2), and therefore a_1 and a_2 would both be possible in this implementation, which does not comply to the original common behaviour. This means that assigning conflict conditions to i-entities may result in incorrect implementations. The interaction server must be informed about conflicting interactions in order to handle these interactions in a proper general way.

Alternatively the interaction server can consider the interactions of a set of interactions requested by a functional entity as conflicting, such that only one of these interactions may happen at a time.

In Figure 8.8, for example, either the confirms of a_1 or the confirms of a_2 happen as a consequence of the requests, but not both, which corresponds to the occurrence of a_1 or a_2, but not both. The interaction server is responsible for the treatment of the choice between these two interactions, by only allowing one pair of confirms to happen.

Figure 8.10 illustrates a correct implementation of the choice between two interactions supported by the interaction server.

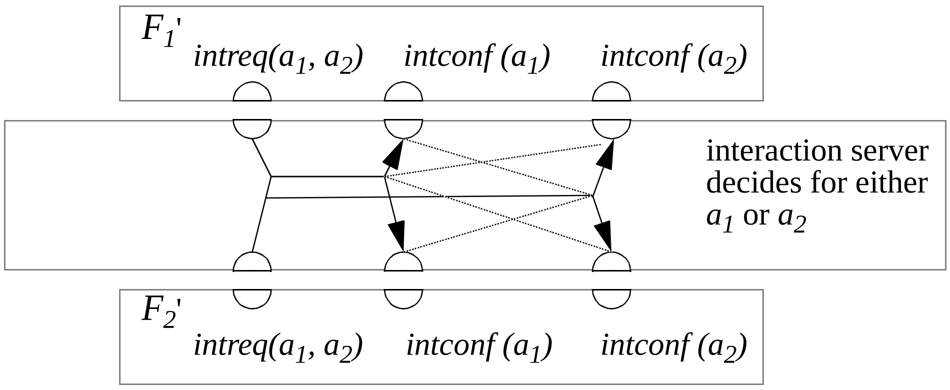


Figure 8.10: Choice of interactions handled by the interaction server

Local Interleaving Between Interactions

Considering all interactions in a request as conflicting interactions seems to be the most conservative way to define the behaviour of the interaction server. One of the benefits of this assumption is that for each *intreq* between an i-entity and the interaction server, only one *intconf* is expected; a serious drawback is that interactions become locally interleaved for each functional entity, even if they were originally independent.

Figure 8.11 depicts an example in which two independent interactions a_1 and a_2 become locally interleaved by considering requested interactions as conflicting.

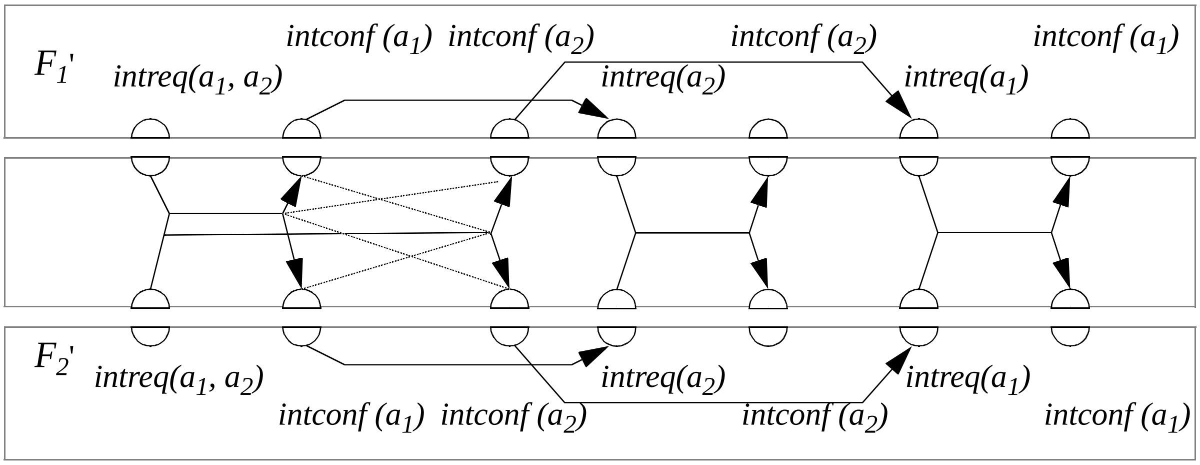


Figure 8.11: Independent interactions becoming locally interleaved

In Figure 8.11 we have assumed that a_1 and a_2 were originally independent actions, which have been assigned to F_1 and F_2. Both F_1' and F_2' request a_1 and a_2, and the interaction server makes a non-deterministic choice between one of them. In case a_1 is confirmed by the interaction server, a_2 is requested by F_1' and F_2', and vice-versa. The interaction server considers requested interactions to be in conflict, and each of the i-entities requests those interactions that are possible at each moment in time. In this way interactions requested by an i-entity will never occur at the same time at the location where this i-entity interacts with the interaction server, which characterizes interleaving. Indeed, if we consider the common behaviour of F_1' and the interaction server, and abstract from the requests, we can deduce that either *intconf* (a_1) causes *intconf* (a_2) or vice-versa.

Since an i-entity does not withdraw requested interactions, we could represent the behaviour of an i-entity that interacts with the interaction server in terms of a set of states and conditional state moves, resembling a state machine. In case a finite set of states can be identified for the behaviour of a certain i-entity, this behaviour can be represented and implemented in terms of *intreq*s and *intconf*s in a systematic and precise way.

8.2.5 Timeout Conditions

The decision of not allowing i-entities to withdraw requested interactions has some consequences on the treatment of timeout conditions. These consequences are discussed below by means of examples.

Figure 8.12 depicts an example of an interaction that is disabled in the behaviour of a functional entity after a timeout.

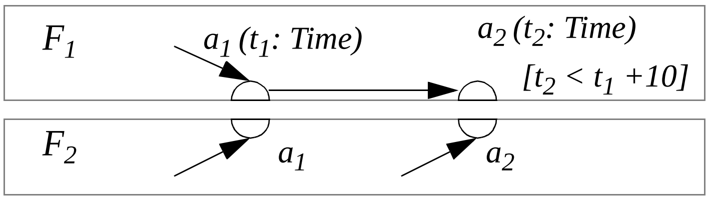


Figure 8.12: Interaction with a timeout condition

In Figure 8.12, F_1 is responsible for the causality between a_1 and a_2 and for the timing constraint of a_2. We suppose that F_2 may have some other constraints on a_2 which determine its occurrence, but we abstract from these specific constraints.

Suppose that interaction a_2 is supported by the interaction server, such that F_1 and F_2 are replaced by the i-entities F_1' and F_2'. According to the assignment of constraints, F_1' should handle the timing constraints of a_2. There are in principle two ways F_1' could handle these timing constraints: (i) by informing these constraints to the interaction server when requesting a_2 or (ii) by withdrawing the request to a_2 after the timeout (immediately after $t_1 + 10$). Since we have already decided that we would not allow request withdrawal, only option (i) is possible. This means that the interaction server should handle constraints on the time of occurrence of interactions in order to support behaviour patterns as the one presented before.

Figure 8.13 depicts the behaviour of the interaction server and the i-entities.

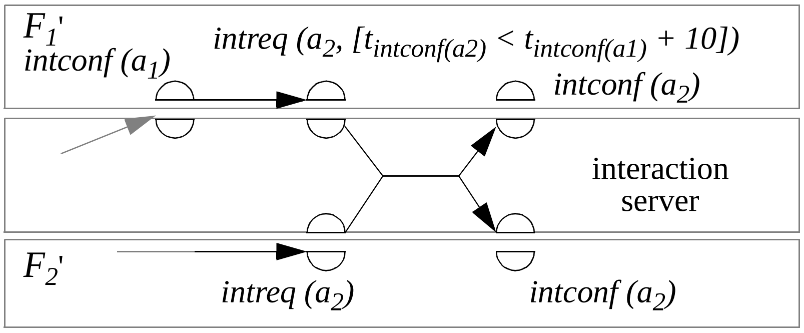


Figure 8.13: Interaction server and timing constraints

In Figure 8.13, *intconf* (a_2) either happens before $t_{intconf\ (a1)}+10$, or it does not happen at all, since *intconf* (a_1) and *intconf* (a_2) correspond to the occurrence of a_1 and a_2, respectively. This condition should be informed to the interaction server as part of the values of information that are established in the *intreq* (a_2) at F_1'.

Often, when an expected action does not happen in a certain time interval, some reaction has to take place. A typical example of this behaviour pattern is data acknowledgement and retransmission, where data is retransmitted if its acknowledgement does not arrive before a certain timeout. Figure 8.14 depicts an example of such a behaviour pattern.

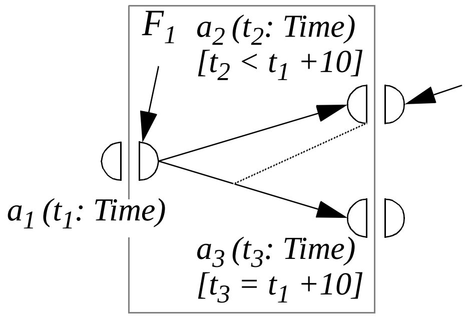


Figure 8.14: Timeout and exclusion

In Figure 8.14, we concentrate on the constraints imposed by F_1, without considering its environment. According to the constraints imposed by F_1, a_2 can only happen before $t_1 + 10$. We also suppose that a_3 must happen at $t_1 + 10$ if a_2 does not happen, which means that $p_3 = 100\%$. Therefore all the other functional entities engaged in the execution to a_3 should be wishing to perform a_3 at the moment $t_1 + 10$.

When we introduce the interaction server to handle interactions a_2 and a_3 we come to the same conclusion already discussed before and illustrated in Figure 8.13: the timing constraints must be informed to the interaction server. However we can transform the behaviour above to an equivalent behaviour in which the symmetric exclusion of a_2 and a_3 is explored to eliminate the timing conditions of a_2.

In Figure 8.14, actions a_2 and a_3 are never both performed in an instance of operation of the behaviour, as a consequence of the timing conditions. Therefore it is correct to replace the asymmetric exclusion between a_2 and a_3 by a symmetric one. By doing this, the timing constraint of a_2 becomes unnecessary if and only if a_3 surely happens at $t_1 +10$ in case a_2 has not happened. Figure 8.15. shows the alternative behaviour obtained in this way.

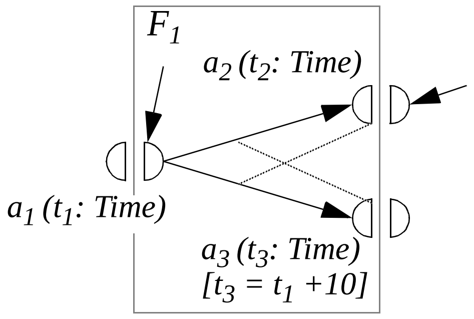


Figure 8.15: Removal of timing constraints

When we introduce the interaction server to handle interactions a_2 and a_3 we can consider two alternatives for the behaviour of F_1': (i) F_1' establishes an *intreq* for both a_2 and a_3 with the timing constraints of a_3 just after a_1 has been confirmed or (ii) F_1' establishes an *intreq* for a_2 just after a_1 is confirmed and another *intreq* for a_3 just before or at $t_1 +10$ if no *intconf* for a_2 has been established.

According to option (i), the interaction server is made responsible for the treatment of timing constraint, but according to option (ii), the timing constraints are actually treated by F_1', through the establishment of an *intreq*(a_3) *10* time units after the establishment of an *intconf*(a_2).

Considering option (ii), since we have supposed that the other functional entities engaged in a_3 should be always ready to execute it at time $t_1 +10$, this implies that a_3 should be already requested by all other participating i-entities at this time. This also implies that a_3 happens as soon as possible after it is requested by F_1', subject to the inherent delays of the interaction server. The composition of i-entities and the interaction server obtained in this way would be considered as an acceptable implementation of the original behaviour, under the condition that the delay between requesting an interaction and having it confirmed to all i-entities engaged in this interaction is much smaller than the timing constraints themselves. Specific performance requirements for the interaction server should be derived from its application, as a function of the timing requirements the interaction server is expected to support.

Figure 8.16 depicts the behaviour pattern of alternative (ii) above, showing how the behaviour of F_1' can be defined, but abstracting from particular assignments of the other requests and confirms to i-entities.

Figure 8.16 shows that in order to apply alternative (ii), i-entities should be allowed to perform more than one request before an interaction is confirmed.

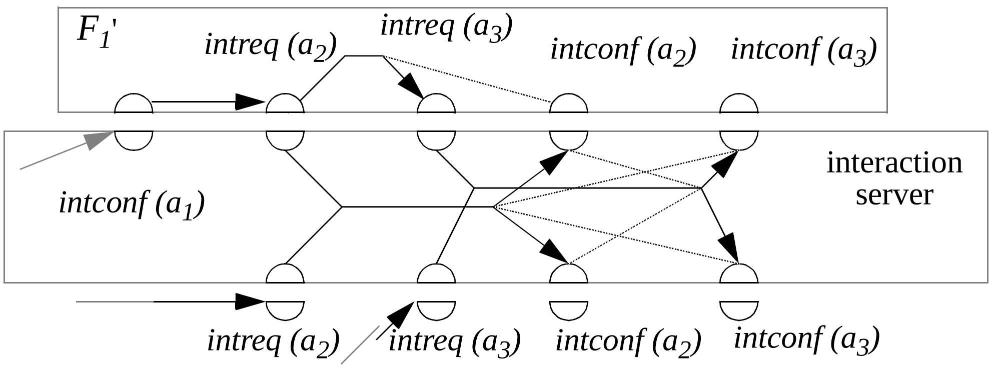


Figure 8.16: Introduction of more than one request per confirm

8.2.6 Actions Outside the Interaction Server

The interaction server is supposed to handle many interactions in a certain environment, but one still has to consider how actions and interactions which are not treated by the interaction server can be related to those interactions that are treated by the interaction server. It is especially important to determine the degrees of freedom to perform actions and interactions outside the interaction server, when the interaction server is used.

Figure 8.17 depicts two examples in which a functional entity has an interaction that is not handled by the interaction server, and an interaction which is handled by the interaction server.

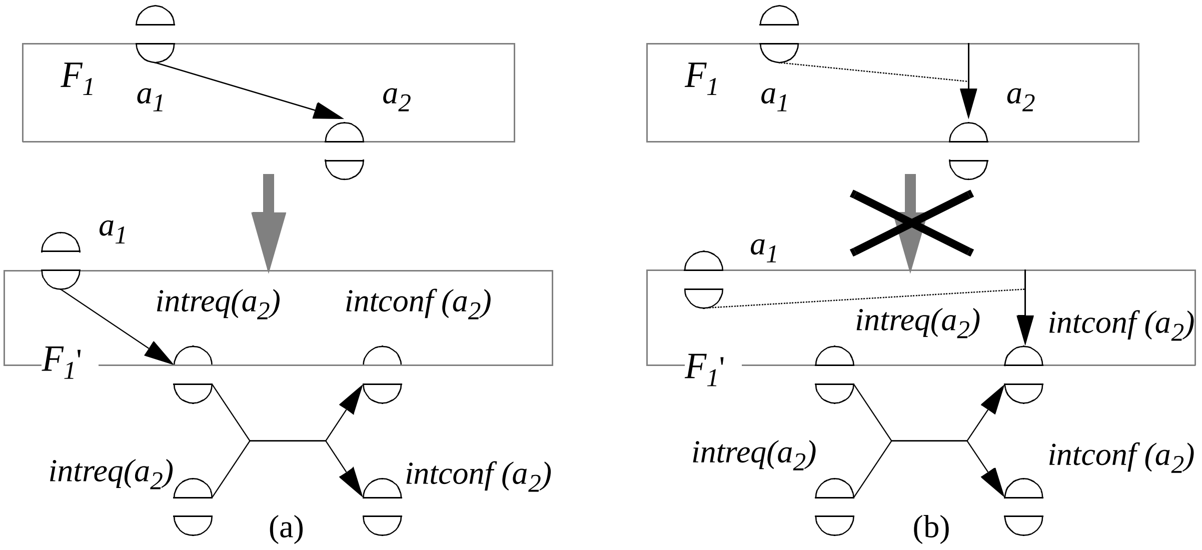


Figure 8.17: Actions not handled by the interaction server

In Figure 8.17 (a) the occurrence of a_1 is a condition for a_2. In this case we can make *intreq (a_2)* in F_1' dependent of the occurrence of a_1, and the original causality between a_1 and a_2 is guaranteed. In Figure 8.17 (b) this does seem to be possible. Making a_1 exclude *intconf (a_2)* does not suffice, since the other *intconf (a_2)* may have happened, and we have defined no mechanism to roll-back interactions that have already been confirmed. It is not possible to make a_1 exclude the other *intconf(a_2)*, since this other *intconf(a_2)* is placed at another i-entity. The decision to disallow interaction withdrawal makes it also impossible to inform the interaction server that an action conflicting with action a_2 has occurred.

We conclude that, under the conditions assumed in the definition of the interaction server actions and interactions that are not handled by the interaction server should not exclude interactions that are handled by the interaction server. A solution to this problem is to refine action a_1 as well, and make it partly happen through the interaction server. In this case we may define a special i-entity *Env* (environment), which corresponds to the environment of the system as seen by the interaction server. Figure 8.18 shows a proper implementation of the example of Figure 8.17 (b) using this solution.

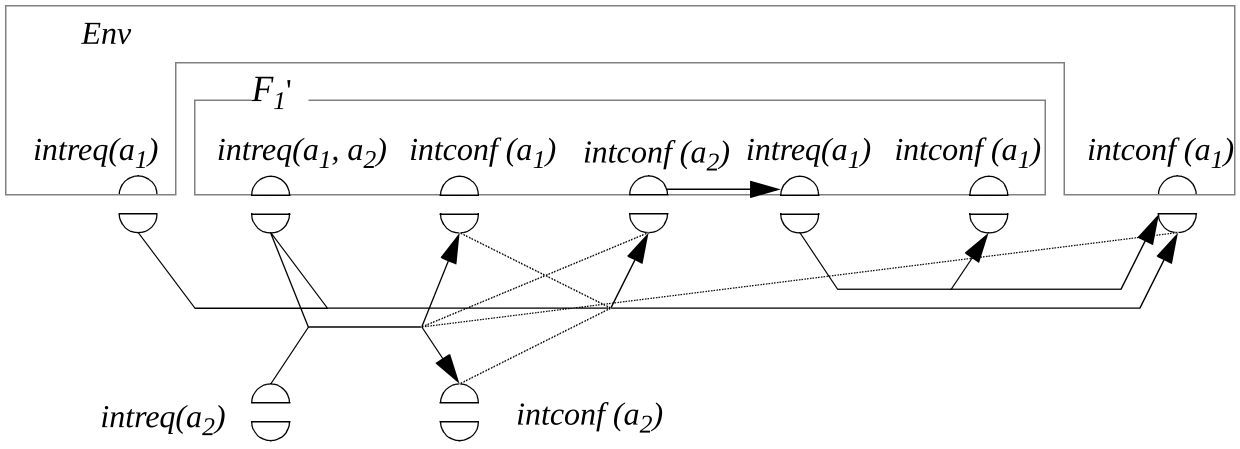


Figure 8.18: Interface between interaction server and the environment

The introduction of a special i-entity *Env* for handling the interface with the environment means actually that interactions with the environment are also treated by the interaction server, and decomposed into requests and confirms in a similar way as the other interactions, otherwise correct implementations cannot be obtained.

8.2.7 Action Instances

In the case of repetitive behaviours it is not feasible to have a unique identifier for each possible interaction. Therefore we consider that the actions can be instantiations of some action types, such that the interaction structure, in the most general case, contains the relationships between action types and functional entities.

Figure 8.19 depicts an example of a repetitive behaviour and an assignment of actions to functional entities that is used to illustrate this more general definition of the interaction structure.

In Figure 8.19, functional entities F_1 and F_2 participate in instances of a_1 and a_2, respectively. Instances of a_1 and a_2 are in conflict with each other, and it is the responsibility of F_3 to handle their conflict. In this case, for example, if F_1 wishes to execute a_1, it does not matter which instance of a_1 is executed.

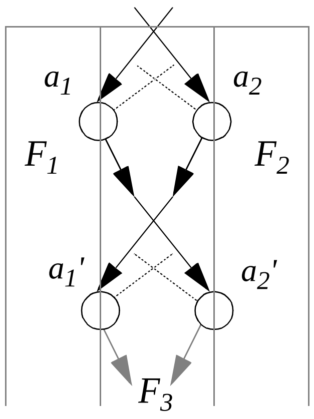


Figure 8.19: Repetitive behaviour and assignment of actions to functional entities

Introducing the interaction server to handle these interactions, it is possible that F_1' requests a_1, F_2' requests a_2 and F_3' requests a_1 and a_2, and a_2 is confirmed to F_2' and F_3'. When F_3' requests a_1 and a_2 again, the request for a_1 by F_1' is still pending, and it can be confirmed to F_1' and F_3' at this point. Therefore the specific instance of a_1 and a_2 that is executed is not relevant at all for the functional entities. The interaction structure in this case simply relates (any instance of) a_1 to F_1' and F_3', and (any instance of) a_2 to F_2' and F_3'.

8.3 Interaction Service Behaviour

Specific choices have to be made with respect to the support of causality relations, which resulted in a specific common behaviour between the interaction server and its users, the i-entities. The most important choices are the following:

- interaction conditions are handled by i-entities, such that an i-entity only requests interactions that are enabled in this i-entity;
- interaction constraints are handled by the interaction server;
- values of information that cannot be deduced from the constraints informed by the i-entities are not generated by the interaction server (no support of value generation);
- each i-entity is allowed to perform one interaction at a time, which results in forced interleaving of interactions at each i-entity;
- limited support to timing constraints, which is only possible by issuing multiple requests;
- actions and interactions not supported by the interaction server cannot exclude interactions supported by the interaction server.

The common behaviour between the interaction server and its users is assigned to a functional entity consistently called interaction service. The behaviour of the interaction service is defined in the sequel.

8.3.1 Service Definitions

The interaction service primitives are the interactions between the interaction server and the i-entities. The interaction service has two primitives:

1. *intreq*, which is a request for interaction. This primitive contains the set of interactions in which a certain process wishes to participate, together with its constraints on values of information to be established. This set of interactions with their constraints is called *offer set*;
2. *intconf*, which is a confirmation of an interaction. This primitive contains a single interaction and the values of information established in this interaction.

Interaction Service Access Points (ISAPs) are physical or logical locations at which i-entities interact with the interaction server. An ISAP identifies an i-entity for the interaction service, such that we can use the same mechanism of functional entity identification to identify ISAPs.

Figure 8.20 depicts the interaction server and three i-entities, and identifies the interaction service.

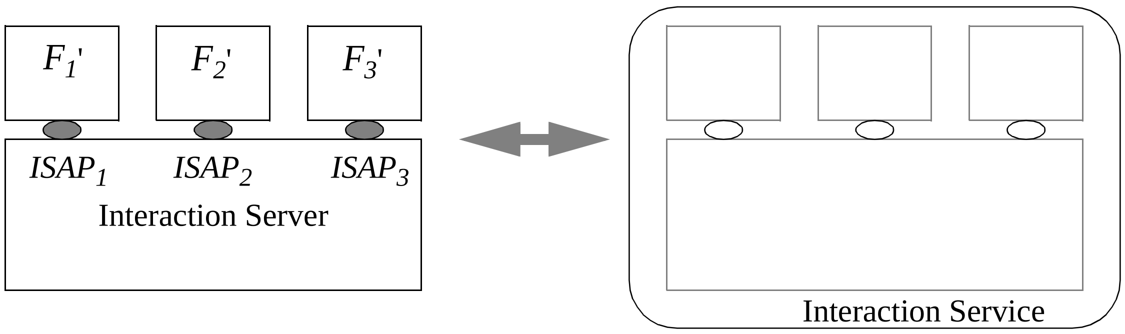


Figure 8.20: Interaction server, i-entities and interaction service

8.3.2 Local Behaviour

The local behaviour of the interaction service at an ISAP should represent that a sequence of *intreq*s causes an *intconf*. This *intconf* must satisfy one of the *intreq*s, such that an interaction can only be confirmed in an *intconf* if it is contained in the offer set of one of the *intreq*, and satisfies the constraints for this interaction as defined in its offer set. An *intconf* informs an i-entity which interaction of the offer set takes place, and with which values of information.

Figure 8.21 shows an instance of the common behaviour between an i-entity and the interaction server, which corresponds to an instance of local behaviour of the interaction service at an ISAP.

In Figure 8.21 each $intconf_i$ is caused by a corresponding $intreq_i$. In case the complete definition of value dependencies would be given, this would represent that each new $intreq_i$ introduces new interactions in the offer set of the i-entity, which implies that more alternative interactions become possible in $intconf_i$ with respect to $intconf_{i-1}$.

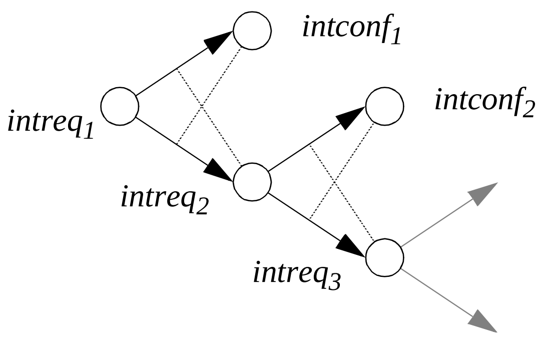


Figure 8.21: An instance of local behaviour of the interaction service

In order to completely represent this local behaviour using causality relations we have to structure it in terms of entries and exits. Figure 8.22 shows the common behaviour between each i-entity and the interaction server in a structured form.

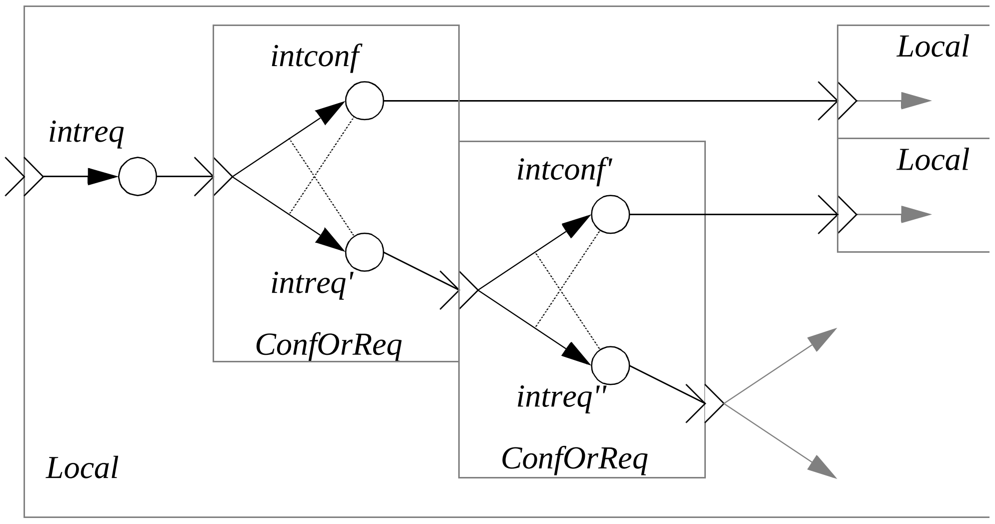


Figure 8.22: A local constraint of the interaction service

In Figure 8.22, behaviour *Local* defines that at least one *intreq* must happen before an *intconf*. After an *intreq* happens, an instance of a choice between an *intconf* and an *intreq* is created. In case an *intconf* happens, an instance of local behaviour is finished, and another one should be started, while in case an *intreq* happens another instance of choice between an *intconf* and an *intreq* is created.

8.3.3 Remote Behaviour

The remote behaviour between ISAPs should conform to the necessary and reliability requirements defined in section 8.2.3. This remote behaviour can be modelled as a set of sequences of *intreq*s, which may cause a set of *intconf*s if a matching of interaction requests occurs. Matching of interaction requests means that there is an interaction such that all i-entities which are supposed to participate in this interaction have informed their offer sets, this interaction belongs to all these sets, and all value conditions of these offer sets are fulfilled by the values established in this interaction. Each sequence of *intreq*s of an i-entity can only be used to cause one set of *intconf*s, which represents the requirement that only one conflicting interaction in an offer set can happen at a time.

At some moment in time there may be more than one interaction in a certain offer set that satisfies the matching of interaction requests, such that these interactions are in conflict. In this case the interaction server makes a non-deterministic choice on one of these interactions, determining that this interaction takes place. The actual interaction that occurs in a conforming implementation of the interaction server depends on the specific internal structure of the implementation.

Figure 8.23 depicts some specific instances of execution of the interaction service remote behaviour, without representing the offer sets explicitly. Only the location of the *intreq*s and *intconf*s (ISAPs) are represented in Figure 8.23, in terms of i-entity identifiers.

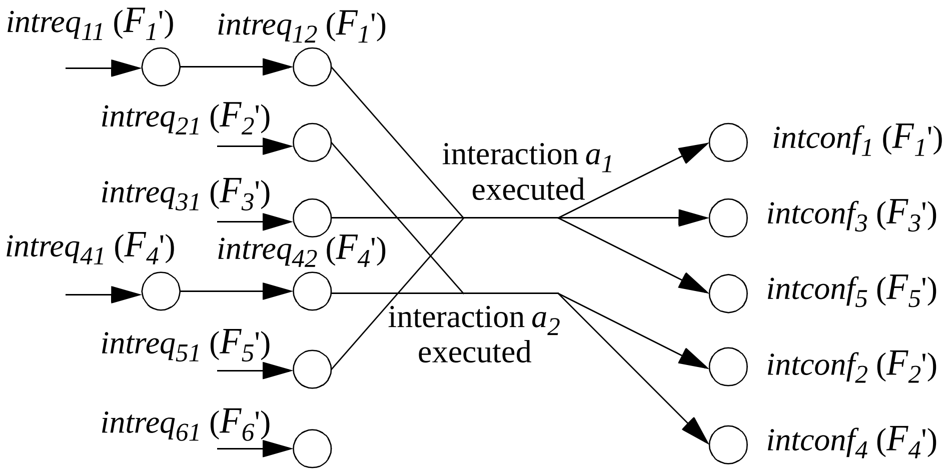


Figure 8.23: Interaction service behaviour definition

In Figure 8.23 we have considered the existence of six i-entities, such that F_1', F_3' and F_5' perform interaction a_1 and F_2' and F_4' perform a_2. The choice of executing a_1 and a_2 is made by the interaction server, and may be a non-deterministic choice on some possible interactions. For example it may be that there is an interaction a_3 involving F_3' and F_6', which has also been requested by these i-entities, but the conflicting interactions a_1 and a_2 are chosen instead.

Since some *intreq*s may not be directly executed, it is possible that some i-entities stay in a deadlock situation waiting for an *intconf*. Deadlocks can be temporary or permanent. Temporary deadlocks are normally desirable situations, such as, for example, waiting for an input device. A permanent deadlock may occur when i-entities that should participate in an interaction have reached states in which they do not participate in interactions any more. In this case, the implementation of the i-entities in permanent deadlock should be removed by some garbage collection mechanism, improving the utilization of resources in the implementation.

In Figure 8.23, for example, functional entity F_6' waits for an *intconf* for $intreq_{61}$, which does not happen because the other i-entities that should participate in the interactions being requested do not wish to interact at that time. It is possible that these i-entities decide to participate in some of the interactions requested by F_6' in the future, or that these i-entities have terminated such that the interactions requested by F_6' will never occur.

Behaviour Definition

The behaviour of the interaction service can only be defined in a general way if we have some notation to handle parameterized definitions, instantiation and definition of conditions that depend on an undetermined number of actions. We introduce below an ad-hoc notation for these purposes. This notation is just a syntactical artifact that makes it possible to write the behaviour definition of the interaction service in a compact way. The following notation is used:

- *par e: E [C_1(e)] in <causality-relation>* defines that an instance of *<causality-relation>* is created for each value *e* of sort *E* that satisfies condition *C_1(e)*;
- *C -> par e: E [C_1(e)] in <action>* defines that an instance of *action* is created for each element *e* of sort *E* that satisfy condition *C_1(e)*;
- *choice a: A, b: B [C_1(a, b)]* defines that if it is possible to choose *a* and *b* of sorts *A* and *B*, respectively, such that they fulfil condition *C_1(a, b)*, this choice can be made;
- *conj e: E [C_1(e)] B(exit (e)) -> a* defines that a conjunction of different instances of *B(exit (e))* are a condition for *a*, for all *e* of sort *E* that satisfy the condition *C_1*;
- *disj B(exit) -> a* defines that a disjunction of the exists of instances of *B(exit)* are a condition for *a*.

The remote behaviour of the interaction service consists of a (possibly infinite) set of possible and alternative causality relations between sets of sequences of *intreq*s and sets of *intreq*s. This causality is established depending on the interaction structure, the offer sets of the i-entities and the internal choices of the interaction service.

This behaviour can be defined in a generic way in a step-wise manner. We start by defining behaviour *Local'*, which is a modified version of behaviour *Local* of Figure 8.24. In *Local'* there is an *exit* which is defined as an disjunction of all *intreq*s that happen in this behaviour. An instance of *Local'* is initially instantiated for each ISAP.

Figure 8.24 depicts the behaviour *Local'*, which is defined in a sub-behaviour of the whole service behaviour.

The disjunction of all *intreq*s that happen in an instance of *Local'* is used as the local contribution to the matching of interaction requests since many *intreq*s are allowed to happen before an *intconf*, but we cannot determine which one actually causes the *intconf*. The actual decision for interaction in the interaction server can be implemented in many ways, for example, in a centralized or distributed fashion, and the behaviour definition should not constrain these possibilities. For example, if two *intreq*s happen at an ISAP there is no guarantee that an *intconf* that follows these *intreq*s is caused by the last occurring *intreq*.

Behaviour *Local'* can be defined using causality relations and the ad-hoc notation introduced before in the following way, where e_i is an ISAP (i-entity) identifier and O_i is an offer set.

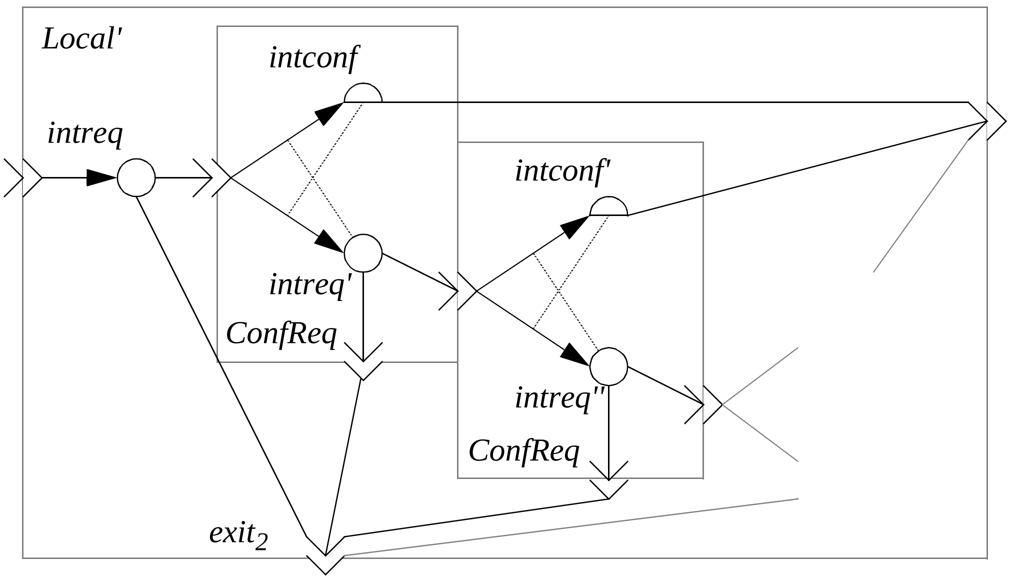


Figure 8.24: The modified local sub-behaviour of the interaction service

```
Local' := {
        entry ($e_i$) -> intreq ($e_i$, $O_i$),
        intreq ($e_i$, $O_i$) $\vee$ disj ConfReq(exit) -> exit$_2$ ($e_i$, $O_i$),
        disj ConfReq(exit$_1$ ($e_i$)) -> Local' (entry ($e_i$))
        intreq ($e_i$, $O_i$) -> ConfReq(entry($e_i$))
        where
        ConfReq := {
                entry ($e_i$) $\wedge \neg$ intreq ($e_i$, $O_i$) -> intconf ($e_i$, o),
                entry ($e_i$) $\wedge \neg$ intconf ($e_i$, o) -> intreq ($e_i$, $O_i$),
                intconf ($e_i$, o) -> exit$_1$ ($e_i$),
                intreq ($e_i$, $O_i$) -> exit$_2$ ($e_i$, $O_i$) } }
```

More than one instance of *Local'* may contribute to the establishment of *intconf*s for an interaction, which represents the requirement that all i-entities that are allowed to participate in an interaction are wishing to do so. Furthermore the individual constraints on the values to be established must match. In this case the matching of interaction requests is fulfilled for some instances of *exit*$_2$ of the some different instances of *Local'* at different ISAPs, the conjunction of these *exit*$_2$s cause *intconf*s at all these ISAPs.

Figure 8.25 shows this behaviour structure, indicating the ISAP identifier of the instances of behaviour *Local'* and where *intconf* happens, and supposing that the matching of interaction requests is fulfilled for three i-entities e_1, e_2 and e_3.

An instance of interaction establishment can be defined using causality relations and the ad-hoc notation introduced before in the following way, where *E* is a set of ISAP (i-entity) identifiers and *is* is a specific interaction structure.

```
IntInstance := {
        conj $e_i$ : ISAP
```

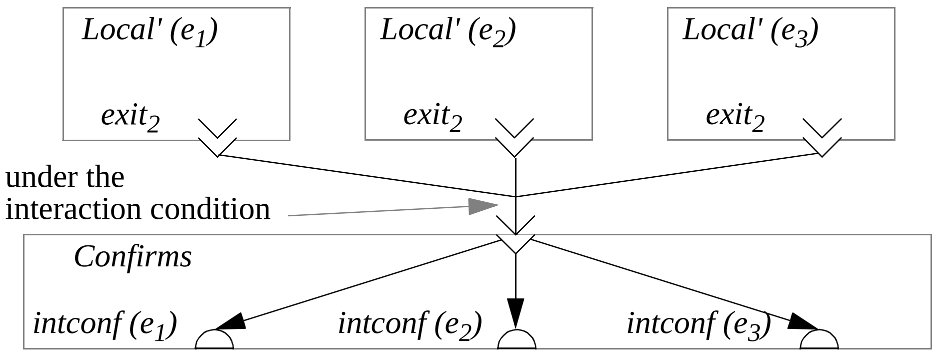


Figure 8.25: Structure of an instance of interaction execution

```
        [choice o: Offer, E: ISAPs [Allowed (o, E, is, $O_i$)] $\wedge$ $e_i \in$ E]
        Local' (exit$_2$($e_i$, $O_i$)) -> Confirms(entry (E, o) )
where
Confirms := { entry (E, o) ->
        par $e_i$ : ISAP [$e_i \in$ E] in intconf ($e_i$, o) } }
```

In this behaviour definition, function *Allowed* determines whether the matching of interaction requests is satisfied by an offer, a set of i-entities, the interaction structure and the offer sets of the i-entities. *intconf*s are generated to confirm this offer to all participating i-entities.

The total remote behaviour of the interaction service is the composition of all possible instances of interaction execution for an interaction structure. There should be only one instance of *Local'* active for an ISAP at a time and each instance of *Local'* is able to cause the execution of only one interaction, causing in this way only one set of *intconf*s. These behaviour requirements follow from the decision of considering all interactions in an offer set as conflicting.

Normally we would represent the service behaviour as a composition of local and remote constraints. However in the case of the interaction service the local constraint has to be almost completely defined in the remote behaviour, such that structuring this behaviour in terms of local and remote constraints yields a redundant specification. Still it is important to define the local behaviour, such as we have done in section 8.3.2, since the i-entities should know how they have to behave in order to interact with the interaction server.

The global behaviour of the interaction service consists of all possible instances of interaction, and can be defined in the following way:

```
IService (is : IS) := {(Local', Confirms: intconf*)
 par $e_i$ : ISAP in start -> Local'(entry ($e_i$)),
par IntInstance (is)
        where ... ) }
```

Instances of behaviours *Local'* and *Confirms* share the *intconf*s that happen at their common ISAPs. An instance of *Local'* is initially created for each *ISAP*. The statement *par <behaviour>* defines that infinite many instances of the behaviour following *par* are unconditionally enabled. This implies that infinite many interaction can be supported by the interaction service.

8.3.4 Abstract Data Type Definitions

The interaction service behaviour is only completely defined when the data sorts and the function *Allowed* are defined. However the definition of these sorts and function *Allowed* implies a lot of design decisions, for example on the data structures of the interaction structure, interaction offers, parameters, conditions on parameters and matching of interaction requests. This section defines these data structures in an abstract way using first-order logic and elementary set theory.

Interaction Structure

Let A be a set of interaction identifiers, each interaction identifier identifying an unique interaction, and $\mathcal{E}$ be a set of i-entity identifier sets, where each entity identifier represents an i-entity, and consequently its ISAP. An $\mathcal{IS}_S$ (interaction structure) of a certain specification S is a relation $A \times \mathcal{E}$ between interaction and i-entity identifier sets, such that $< a_i, E_i> \in \mathcal{IS}_S$ means that i-entities E_i can interact by performing a_i.

The interaction structure determines the logical interface between process management and the interaction server. We consider that $\mathcal{IS}_S$ (or $\mathcal{IS}$ for short) is one of the parameters to be manipulated by the interaction service. The interaction structure can, in principle, be modified during the operation of the system, since i-entities may be terminated or created. In this design example we do not allow the interaction structure to be modified.

In LOTOS one can define the interaction structure of a specification in terms of a relation between sets of processes and gate identifiers. This relation can be obtained by flattening the specification In [3], for example, the structure of interacting processes of a LOTOS specification is dynamically generated and updated. Since we want to have a more flexible way to identify and distinguish events, e.g. not only by its location attribute, but also by some characteristics of its values of information, we consider that interactions are identified, but we abstract from the actual interaction identification mechanism.

Request History

Let E be a set of entity identifiers, each entity identifier representing an entity of a certain specification S, and let $\mathcal{O}$ be a set of offer sets. We consider that each entity is able to compute dynamically its offer set, such that there is a function $OS_{(t)}: E \to \mathcal{O}$ that we call OS for short. This function relates an i-entity to its set of possible interactions.

We define $\mathcal{R}$ (request history) as:

$$\mathcal{R} := \{<e_i, OS(e_i) >| \, i \in I\}$$

The request history represents the *intreq*s performed by all i-entities at some point in time, in terms of their current offer sets.

Interaction Offer

Let *O* be an offer set, where each element of $o_i \in O$ is called an interaction offer, *A* a set of interaction identifiers, *PL* a set of attribute-lists and *CO* a set of value constraints, each constraint defined as a function *co*: *PL -> {false, true, undef}*. We define a function *Interaction*: *O -> A*, which determines which interaction is referred to in an interaction offer, a function *ParamList*: *O -> PL*, which determines which interaction parameters are considered in the interaction offer, and a function *Conditions*: *O -> CO*, which determines which constraints must be satisfied for the interaction to occur.

Interaction attributes of potential interest to the interaction server are location, time and value. We have decided to give a limited support to timing requirements, such that explicit time constraints are not being considered. Location and values of the original interactions can be both treated as values to be agreed upon under the supervision of the interaction server.

We define an interaction offer $o_i \in O$ as the triple:

$$o_i := <Interaction\ (o_i),\ ParamList\ (o_i),\ Conditions\ (o_i)>$$

Conditions is a function on the whole interaction offer resulting in a function *co* on the parameter list. Function *co*, when applied to specific instances of parameter lists, can get the values undefined (*undef*), in case the constraints it represents cannot be evaluated by lack of concrete parameter values, *true*, in case all the constraints are fulfilled by the interaction offer, or *false*, in case the constraints are not fulfilled.

Parameter Values

Values to be agreed upon by the i-entities are defined as a list, such that each of these values has a specific position. The assignment of specific meanings to the values and their positions is outside the scope of this exercise.

Let *D* be a set of parameters, each parameter of the form $d_i := ?\ x_i: t_i$ or $d_i := !E_i$. A parameter list $pl_j \in PL$ has the form

$$pl_j := <d_1, \ldots ,d_k> \mid k \in N \wedge d_1 \in D \wedge \ldots \wedge d_k \in D$$

This means that interaction parameter lists are composed of variables $?x_i: t_i$ or values $!E_i$.

Interactions

We define the function *Interactions*: $\mathcal{O} \rightarrow \mathcal{A}$, where $\mathcal{O}$ denotes a set of offer sets and $\mathcal{A}$ a set of interaction-identifier sets as:

$$Interactions\ (C_i) := \{a_i \mid \exists\ o_j \in C_i,\ a_i = Interaction\ (o_j)\}$$

This function extracts the interaction identifiers of an offer set. When applied to the set of offers of an i-entity this function results in the set of interaction identifiers associated with these offers.

An offer o_i is a *concrete interaction* if

$$\mathit{ParamList}\ (o_i) = <d_{i1}, ...,d_{im}> => \\ (\forall\ d_{ij} \mid j \in \{1,...,m\}, d_{ij} = !\ E_j) \vee \mathit{ParamList}\ (o_i) = <>$$

This definition is necessary since one of the roles of the interaction server is to compute a concrete interaction that matches the patterns of the multiple offer sets. We define the sub-set O^c as the set of concrete offers.

$$O^c = \{\ o_i \in O \mid (\ \mathit{ParamList}\ (o_i) = <d_{i1}, ...,d_{im}> => \\ (\forall\ d_{ij} \mid j \in \{1,...,m\}, d_{ij} = !\ E_j)\ ) \vee \mathit{ParamList}\ (o_i) = <>\ \}$$

Matching Conditions

We define that the function *ValueMatch*(o_i, o_j) of boolean result, where o_i and o_j are interaction offers, is true iff

$$\mathit{ParamList}\ (o_i) = <d_{i1}, ...\ ,d_{im}> \wedge \mathit{ParamList}\ (o_j) = <d_{j1}, ...\ ,d_{jn}> => \\ (\ m = n \wedge \\ (\forall\ k \in \{1,...,m\} \mid d_{ik} = ?\ x_i{:}\ t_i \wedge d_{jk} = ?\ x_j{:}\ t_j\ \Rightarrow t_i = t_j), \\ d_{ik} = ?\ x_i{:}\ t_i \wedge d_{jk} = !E_j \Rightarrow \mathit{Sort}\ (E_j) = t_i, \\ d_{ik} = !E_i \wedge d_{jk} = ?\ x_j{:}\ t_j\ \Rightarrow \mathit{Sort}\ (E_i) = t_j, \\ d_{ik} = !E_i \wedge d_{jk} = !E_j \Rightarrow E_i\ = E_j\ )\ ) \\ \vee \mathit{ParamList}(o_i) = \mathit{ParamList}(o_j) = <>$$

Otherwise the function is false.

This function determines whether the parameter list of an interaction offer matches the parameter list of another interaction offer.

We define that the function *CompleteMatch*(o_i, o_j) of boolean result, where o_i and o_j are interaction offers, is true iff:

$$(\mathit{Interaction}\ (o_i) = \mathit{Interaction}\ (o_j)) \wedge \mathit{ValueMatch}(o_i, o_j) \wedge \\ \mathit{Conditions}\ (o_j)\ (\mathit{ParamList}(o_i)) = \mathit{true}$$

Otherwise the function is false.

This function determines the matching of interaction identifiers, values and conditions.

We define that function *MatchHistory*(o_i, E_i, R) of boolean result, where o_i is an interaction offer, E_i is a set of i-entity identifiers and R is the request history, is true iff

$$\forall\ e_j \in E_i \mid <e_j, OS\ (e_j)> \in R \wedge\ \exists\ o_j \in\ OS\ (p_j) \mid \mathit{CompleteMatch}(o_i, o_j)$$

Otherwise the function is false.

This function determines if the proper *intreq*s have been issued for a certain interaction offer o_i and i-entity identifier set E_i.

We define that the function *InteractionCondition*(*IS*, $\mathcal{R}$) of boolean result, where *IS* is the interaction structure and $\mathcal{R}$ is a request history, is true iff:

$$\exists\, o_i \in O^C, \exists <Interaction\ (o_i), E_i> \in IS \mid MatchHistory(o_i, E_i, \mathcal{R})$$

Otherwise the function is false.

The matching of interaction requests states that there is a concrete interaction offer and a set of i-entity identifiers, such that these i-entities interact through the interaction of the concrete interaction offer, and there is a matching history for this interaction offer and these i-entities.

We define that the function *Allowed(*o_i, E_i, *IS*, $\mathcal{R}$) where o_i is an interaction offer, E_i is a set of i-process identifiers, *IS* is the interaction structure and $\mathcal{R}$ is a request history, is true iff:

$$o_i \in O^C \wedge <Interaction\ (o_i), E_i> \in IS \wedge MatchHistory(o_i, E_i, \mathcal{R})$$

Otherwise the function is false.

A concrete interaction offer that fulfils the matching of interaction requests can be established and confirmed to all i-entities. The minimum sub-set of the request history that fulfil the matching of interaction requests, such that there is no e_j that does not belong to E_i and has an entry in the sub-set, should be used to generate *intconf*s in the interaction service behaviour definition.

8.4 Distributed Implementation

The interaction server is responsible for sorting out the requirements of synchronization between i-entities and conflict between interactions defined in the interaction service. In case these requirements are handled in the implementation of the interaction server by a single centralized functional entity, this functional entity becomes a potential performance bottleneck. Therefore, in order to be able to support many differences patterns of interaction traffic with an implementation of the interaction server, one should develop protocols where synchronization and conflict between i-entities are handled by multiple distributed functional entities.

This section briefly discusses some protocol alternatives for a distributed implementation of the interaction server.

8.4.1 Protocols for Synchronization and Conflict

Interaction scheduling, which consists of deciding which interaction takes place, should be handled by multiple distributed entities in implementations of the interaction server. Distributed implementations allow for independence between interactions that are not in conflict, potentially increasing the number of interactions that can be handled by the interaction service per time unit. The drawback of distribution is the coordination necessary to solve interaction conflicts, meant to guarantee that an i-entity does not interact through more than one of requested interaction at a time.

Some protocols that have been proposed in the literature can be used to support the interaction service. Two protocols in which the support of synchronization between i-entities and conflict between interactions is clearly worked out in their protocol functions are the *event manager protocol* ([2] and [1]) and the *two-phase lock protocol* ([3]). These two protocols are briefly compared below, by considering their most essential characteristics.

In order to compare these two protocols properly, we have chosen a structure of functional entities which is applicable to both protocols. This structure consists of *mediators*, which are protocol entities that directly offer the interaction service to the i-entities, and *coordinators*, which are protocol entities that perform interaction scheduling. A reliable communication service provider is available, allowing mediators and coordinators to exchange messages (protocol data units).

Figure 8.26 depicts an structure functional entities for the interaction protocol, for the case of three i-entities and two coordinators.

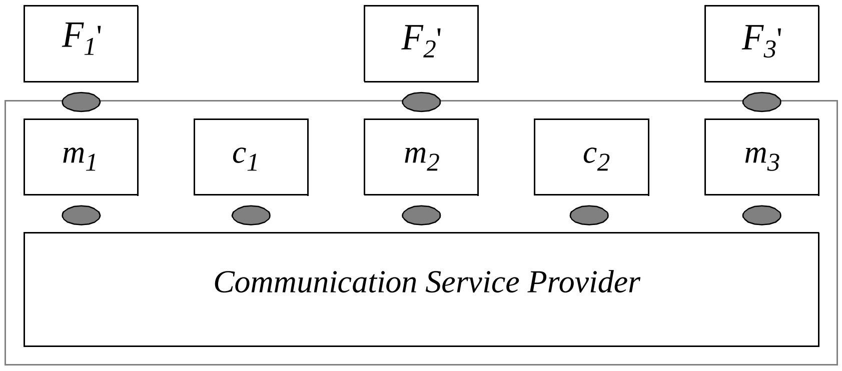


Figure 8.26: Functional entity structure for the distributed interaction server

The mediator of an i-entity F_i' is denoted by m_i, and coordinators are denoted by c_j. Each coordinator is responsible for the execution of a set of interactions.The sets of interactions supported by the different coordinators are disjoint.

Event Manager Protocol

According to the event manager protocol, mediators that have performed *intreq*s inform the coordinators of the interactions contained in their offer sets, by sending them *ready* messages. Coordinators exchange *token* messages in which the status of the mediators are represented. In this way coordinators know whether mediators are ready to participate in interactions or not. The *token* messages are exchanged such that at most one coordinator has a token at a time.

A matching of interaction requests is simply considered as the situation in which there is an interaction which has been requested by all i-entities that participate in this interaction. A coordinator that has detected a matching of interaction requests and has a token is allowed to execute the interaction that satisfies the matching. A coordinator that executes an interaction sends *execute* messages to all mediators involved in the interaction. A mediator that receives an execute message performs an *intconf*s, such that the interaction is confirmed to the i-entity attached to this mediator.

Since at most one coordinator has a token, and only a coordinator that has a token is allowed to execute an interaction, it is not possible that two or more conflicting interactions happen at a time.

Two-phase Lock Protocol

According to the two-phase lock protocol, mediators that have performed *intreq*s inform the coordinators of the interactions contained in their offer sets, by sending them *ready* messages. Coordinators, however, do not exchange messages directly. Since it is possible that more than one coordinator handle conflicting interactions, coordinators must be able to schedule their interactions, in order to avoid that an i-entity executes two or more conflicting interactions at the same time.

The protocol between the coordinators and the mediators is based on a two-phase commit algorithm, where the coordinators cope with the interaction condition and the mediators cope with the conflicting interactions of each i-entity, making sure that a mediator is at most committed to one coordinator. A two-phase commit algorithm is based on the following phases: (1) lock all necessary resources, and (2) unlock all resources as soon as they have been used.

In phase (1) coordinators that have detected a matching of interaction requests for some interaction and some set of i-entities, try to lock the mediators, by sending them *lock* messages. A mediator receiving a *lock* message may either grant it or reject it. A condition for a mediator to grant a lock is that is has not granted any other lock before. When a mediator grants a lock, it sends a *granted* message back to the coordinator that issued this *lock* message, and keeps other *lock* messages to be answered later.

In phase (2) a coordinator receives *granted* messages from all mediators and sends *commit* messages to these mediators. When a mediator that has sent a *granted* receives a *commit* message, it rejects *lock* messages that have been kept, if any, by sending *reject* messages to the coordinators that have sent these *lock* messages, and issues an *intconf* to confirm the interaction to the i-entity. When a mediator that has sent a *granted* receives an *abort* message, it sends a *granted* message to some other interaction for which it has received a *lock*, and waits for a *commit* or an *abort* message again.

Evaluation

Since only a coordinator that has a token can perform an interaction in the event manager protocol, only one interaction can be performed at a time. In the two-phase lock protocol, interactions that are performed by disjoint sets of i-entities can occur independently, since conflict is handled in cooperation with only those mediators attached to i-entities involved in the interactions.

In order to support the interaction service as it has been defined in section 8.3, a protocol should also support value establishment and the evaluation of value constraints. In the two-phase lock protocol value establishment and the evaluation of value constraints can be supported without the need to modify the message structure of the protocol in the following way: a *ready* message can be sent with the values imposed by an i-entity, a coordinator can detect the matching of interaction requests and the existence of a concrete offer in the values that have been imposed by the i-entities, *lock* messages can distribute this concrete value to all mediators, and a *granted* message can inform the coordinator that the concrete offer comply to the constraints of this i-entity. In the event manager protocol extra messages would be necessary in order to guarantee agreement on values of information, since only a *ready* and an *execute* message is exchanged between mediators and coordinators.

These observations influenced our decision of further considering the two-phase lock protocol further in this section. The protocol to considered is a modified version of the protocol presented in [1] in order to support agreement on data values and multiple *intreq*s before an *intconf* at an ISAP.

8.4.2 Interaction Protocol

The two-phase lock protocol that support the interaction service is consistently called interaction protocol. This section discusses and illustrates some instances of behaviour of the mediator and the coordinator protocol entities. The sending and receiving of messages through the communication service provider are modelled by distinct actions, named by the message type that is sent or received.

Mediator Behaviour

A mediator executes *intreq*s, which cause the transmission of *ready* messages to the coordinators. After *ready* messages have been sent, a mediator waits for *lock* messages. The first *lock* message that arrives should be answered with a *granted* message, in case the values contained in the *lock* message fulfil the constraints defined in the *intreq* for the interaction being executed, or with an *reject* message, if the values do not fulfil these constraints. A *granted* message is followed by a *commit* or an *abort* message, meaning that the interaction is either executed or aborted, respectively. Subsequent *lock* messages are only answered after a *commit* or an *abort* are issued, with a *reject* or a *lock* message, respectively

Figure 8.27 depicts an instance of execution of this behaviour, considering that one *intreq* initially occurs for two interactions.

In Figure 8.27 the mediator executes one *intreq*, in which the offer set $\{a_1, a_2\}$ is established. A *ready* message is generated for each one of these interactions. When a *lock* (a_1) arrives it causes the mediator to issue a *granted* (a_1) or a *reject* (a_1). A *granted* (a_1) is followed by either a *commit* (a_1) or an *abort* (a_1), and a *commit* (a_1) finally causes an *intconf* (a_1). In case a *lock* (a_2) has also been received by the mediator and (i) the mediator issues a *reject* (a_1), or (ii) the mediator receives an *abort* (a_1), the mediator issues either *granted* (a_2) or a *reject* (a_2), and so on.

Including values of information, each *ready* message would contain all values and variables of an interaction, and the *lock* message would contain a concrete offer. A *granted* message would be issued if the concrete offer satisfies the local constraints on the values and a *reject* message would be issued otherwise. This implies that the constraints, which may be difficult to code in messages, are not exchanged in this protocol.

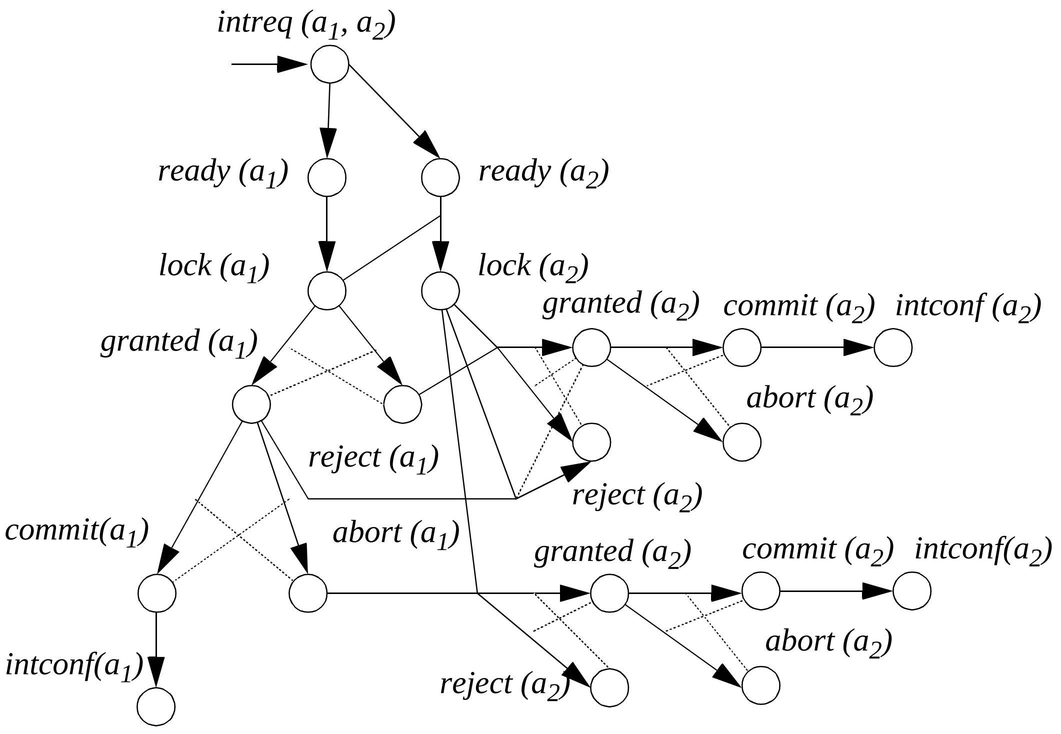


Figure 8.27: An instance of execution of the mediator behaviour

Coordinator Behaviour

A coordinator handles the execution of one or more interactions. In order to handle interactions, a coordinator possesses the part of the interaction structure that concerns to the interaction it handles. We consider the case of a coordinator that handles only one interaction below.

A coordinator waits for a *ready* message from all mediators of the i-entities that participate in an interaction. When all these *ready* messages have arrived, the coordinator generates a concrete offer for this interaction, if possible, and issues a *lock* to all these mediators. After receiving *granted* from all these mediators, the coordinator issues a commit to each one of them.

Figure 8.27 depicts an instance of execution of this behaviour for interaction a_2, which is performed by two i-entities e_1 and e_2.

In Figure 8.27 we observe that the pattern of behaviour between sequences of *intreq*s and *intconf*s that has been identified in the behaviour of the interaction service (see for example Figure 8.25) is repeated twice in this instance of the coordinator behaviour, involving *ready* messages and *lock* messages, and involving *granted* messages and *commit* messages. The former copes with the matching of interaction requests and the latter copes with conflict since each i-entity is allowed to grant only one interaction at a time.

Abstracting from the actions that model message exchange between protocol entities, one should be able to obtain an abstract behaviour that coincides with the interaction service behaviour, by

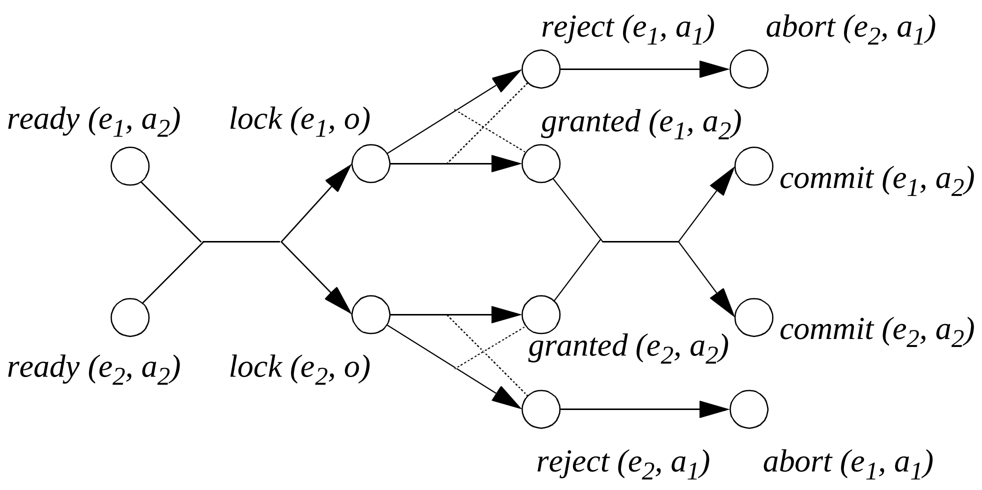


Figure 8.28: An instance of execution of the coordinator behaviour

using the rules for abstracting from inserted actions given in Chapter 6. However, we refrain from doing it in this work, since the protocol behaviour for all possible instances is far too complex.

8.5 References

[1] R. Bagrodia. A distributed algorithm to implement n-party rendez-vous. In *Foundations of Software Technology and Theoretical Computer Science*, volume 287 of *Lecture Notes in Computer Science*, pages 138–152. Springer-Verlag, 1987.

[2] R. Bagrodia. Process synchronization: design and performance evaluation of distributed algorithms. *IEEE Transactions on Software Engineering*, 15(9):1053–1065, Sept. 1989.

[3] P. Sjödin. *From LOTOS Specifications to Distributed Implementations*. PhD thesis, Uppsala University, Uppsala, Sweden, Dec. 1991.

[4] P. van Eijk and H. Eertink. Design of the Lotosphere symbolic LOTOS simulator. In J. Quemada, J. Mañas, and E. Vásquez, editors, *Formal Description Techniques, III*, pages 577–580. Elsevier Science Publishers B.V. (North-Holland), 1991.

[5] C. A. Vissers, L. Ferreira Pires, and D. A. Quartel. *The Design of Telematic Systems*. University of Twente, Enschede, the Netherlands, Nov. 1993. Lecture Notes.

Chapter 9

Conclusions

This chapter draws some important conclusions and identifies some areas in which more research is necessary. This chapter is structured as follows: section 9.1 discusses some general aspects, section 9.2 summarizes the structuring techniques and design operations that have been addressed in this thesis, section 9.3 relates these structuring techniques and design operations to the milestones of the design process and section 9.4 identifies some more subjects for further research.

9.1 General Conclusions

This thesis has shown that addressing design concepts, structuring techniques and design operations from a framework that is more general that any available design language brings an insight in the requirements and problems intrinsic to them that would not be achievable if we had concentrated on a specific design language and its design model.

The approach taken in this thesis has been to make things as general as possible. Some work should be done to select those constructs and results that are more often used, and to introduce shorthand notations for those constructs that have a complex representation in our design model. In some sense there is a need to select the more useful specific results out of the general ones.

Prescription and Description

The techniques and methods for performing design operations addressed in this thesis are meant to support system development, in the sense that they have been defined considering that the system to be designed does not exist. Design specifications are considered as prescriptions for their implementations; this specific requirement, for example, has been one of the reasons for using causality and exclusion as basic concepts for behaviour definition. However, when applying an implementation strategy such as the use of pre-defined implementation constructs discussed in section 1.3.5, one should be able to make models of functional entities that exist, being actually more of a description. Composition of descriptive models should somehow be compared with the prescriptive models generated in early phases of design, for example, in order to assess the correctness of the composition of pre-defined implementation constructs with respect to the prescription of the system to be built. Some research is necessary in order to establish the proper relationships between these models.

Performance Modelling

Another aspect to be considered is performance modelling. Traditionally performance models are descriptive, in the sense that they allow the representation of properties of systems considering that these systems exist. The timing (performance) requirements that can be expressed using causality relations does not consider, for example, the probability distribution of a certain time attribute value. This specific modelling decision has been based on the prescription objectives of our model. However, in later phases of design, when more details about system components are known, we may have the opportunity to couple probabilistic distributions of time attribute values to the prescriptive specification obtained with causality relations. In this way traditional performance modelling techniques could be coupled to causality relations in a more generic design environment. Some research should be dedicated to this coupling.

Specification Styles

Specification styles have been introduced in [1] as a way to structure specifications according to specific design objectives. When considering two distinct but related domains for defining design objectives and characterizing design steps, which are the entity and behaviour domains, we can conclude that some specification styles have been necessary to cope with the impossibility of formally defining the entity structure and the mappings from a behaviour specification onto the functional entities of this structure, at a certain abstraction level.

The constraint-oriented specification style corresponds to defining a behaviour as a composition of sub-behaviours according to the constraint-oriented behaviour composition (Chapter 5), and a mapping of this behaviour onto a single functional entity at a certain abstraction level. The resource-oriented specification style corresponds to defining a behaviour as a composition of sub-behaviours according to the constraint-oriented behaviour composition structuring technique, and a mapping of these sub-behaviours onto functional entities at a certain abstraction levels. The explicit mapping from behaviours to functional entities make the role of specification styles even more clear.

The state-oriented specification style refers to the explicit representation of behaviour states, and is much more related to the behaviour domain and to the representation of behaviours. The monolithic specification style only allows the mapping of a behaviour onto a single functional entity.

9.2 Design Steps and Design Operations

The design framework discussed in this thesis consists of an entity domain and a behaviour domain. Design steps in the design process and their corresponding design operations have been characterized in these two domains throughout this thesis.

Figure 9.1 summarizes the degrees of freedom of the behaviour domain, in terms of possible structuring techniques and design operations that can be applied to a certain behaviour.

A certain monolithic behaviour can be structured in terms of a composition of sub-behaviours in a causality-oriented behaviour composition, in terms of a composition of constraints in a con-

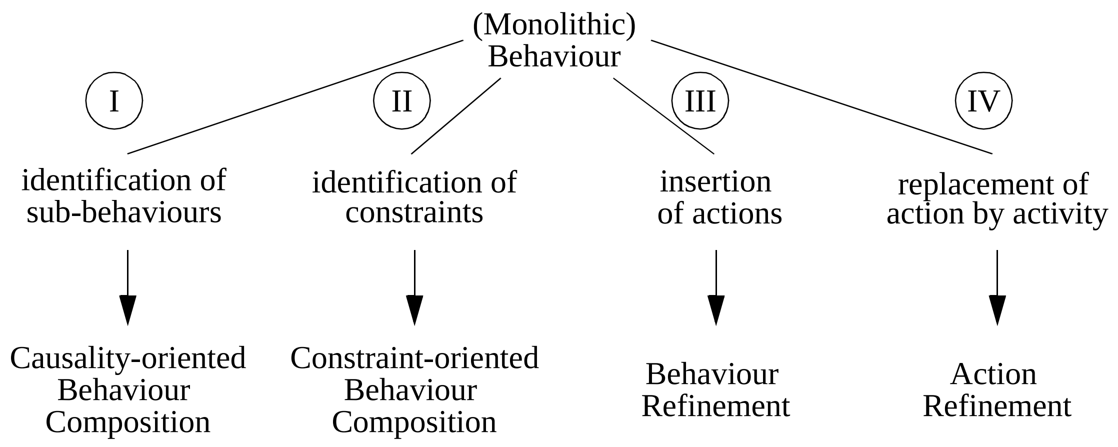

(Monolithic) Behaviour
I
II
III
IV
identification of sub-behaviours
identification of constraints
insertion of actions
replacement of action by activity
Causality-oriented Behaviour Composition
Constraint-oriented Behaviour Composition
Behaviour Refinement
Action Refinement

Figure 9.1: Structuring techniques and design operations in the behaviour domain

straint-oriented behaviour composition, or as a combination of both causality and constraint-oriented behaviour compositions. A behaviour, which may be either monolithic or structured, can also be refined by using the behaviour refinement design operation, in which actions are inserted, or by using the action refinement design operation, in which actions are replaced by an activity. These structuring techniques and design operations can be repeatedly applied in the course of the design process.

9.3 Revisited Design Milestones

Design milestones of a global design process have been identified in Chapter 2, by considering abstraction levels of system representation. Design steps and concrete technical manipulations of designs have been identified from the abstraction gap between milestones. At this point we are able to revisit the milestones and consider more general design trajectories for the elaboration of designs.

Figure 9.2 depicts the revisited milestones of the design process, indicating which structuring techniques and design operations of Figure 9.1 may be necessary in order to move from a milestone to another.

Design steps are performed when designers move from a design milestone to another. Some design steps can be identified in Figure 9.2, by considering the design milestones:

- *From Interaction System to (Integrated) System*: a design step to make this transition consists of assigning responsibilities on the execution of interactions to the system and to its environment. For the assignment of responsibilities to a system and to its environment the following techniques are necessary: (i) constraint-oriented behaviour composition (design structuring technique II) and (ii) assignment of behaviours to functional entities;
- *From (Integrated) System to Distributed System*: a design step to make this transition must define a composition of functional entities that implement the (integrated) system. For the definition of such a composition of functional entities that following techniques may be necessary: (i) behaviour refinement (design operation III), (ii) constraint-oriented behaviour composition (design structuring technique II) and (iii) assignment of sub-behaviours to func-

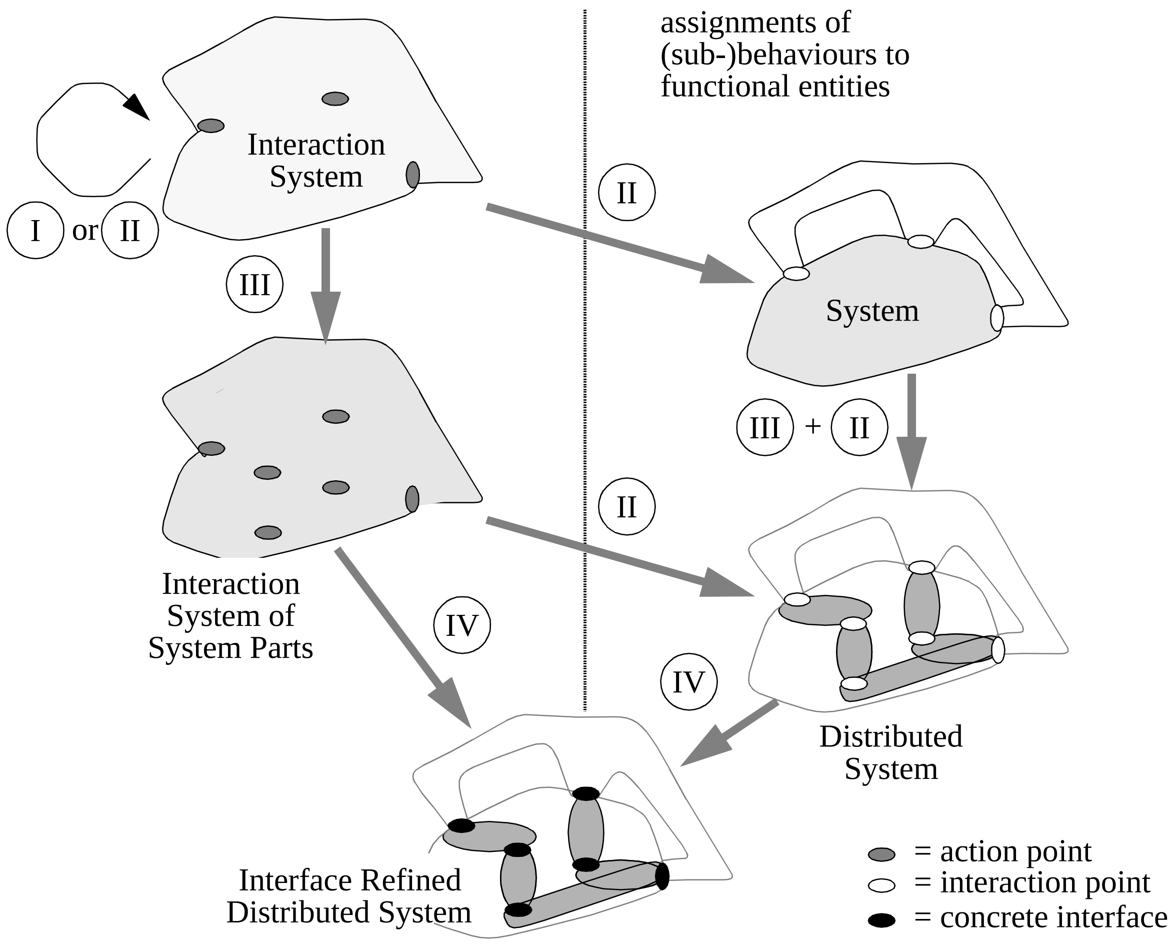


Figure 9.2: Revisited milestones of a global design process

tional entities;

- *Abstract Interface to Concrete Interface*: a design step to make this transition must replace abstract actions by activities consisting of more concrete actions, in which the details of the mechanism that implements these abstract action are revealed. This corresponds to action refinement (design operation IV).

An actions can be considered as an abstract interaction in which the participation of possible multiple functional entities is ignored. This observation actually applies not only for the integrated system with respect to the interaction system, but for any distributed representation of a system with respect to its corresponding interaction system of system parts in the design trajectory, such as represented in Figure 9.2. This implies that an extra dimension can be identified in the design process, in which one reasons exclusively about actions instead of interactions. Therefore one has the choice of defining successive refinements using actions and making loose associations with functional entity behaviours, or defining successive refinements using interactions in which the behaviours of the functional entities are explicitly defined, or a mixture of both.

In some cases we may have already considered some the distribution of action responsibilities at the distributed system milestone, but these may have to be reconsidered when action refinement is applied to define a concrete interface. For these cases it seems that defining specific interaction responsibilities for the functional entities before the definition of their concrete interface is a waste of design effort. More research, for example by studying specific applications, is necessary to determine whether this can be considered as a general rule.

9.4 Further Research

In our design framework we should explicitly define functional entities and their interconnection. In many application areas, such as in mobile communications and Integrated Service Engineering (ISE), one should be able to define systems that can process, create and instantiate functional entities and interaction points dynamically. This kind of meta-level (system that influences itself) introduces some complications that should be further investigated. Considering interaction points as data values of some location sort is expected to help when modelling such applications; more research and possibly an extension of the framework presented in this thesis is necessary.

A method seems to be necessary for specifying protocol functions using causality relations. Traditionally protocol functions are defined using finite state machines, in which normal and exceptional behaviour can be defined at once. Causality relations, on the other hand, enable the definition of behaviour in a sort of step-wise way, in which the exceptions can be defined little by little. It would be interesting if we could develop a method in which exception conditions were specified and incorporated in a normal protocol behaviour, and evaluated, probably with tool support.

The design methodology (model, techniques and methods) discussed in this thesis has been represented using an ad-hoc notation and some graphics. We have consciously decided to leave the definition of a proper design language and its formal semantics, as well as the tool support for further study. The importance of this thesis resides in the depth of the discussion: we have not only considered the representation of a design at a certain abstraction level, but also the design freedom for moving between abstraction levels, and the conformance between designs of consecutive abstraction levels based on a well-considered application of the abstraction principle. The results of this thesis should influence the development of design languages, tool support, implementation strategies and implementation environments.

9.5 References

[1] C. A. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma. Specification styles in distributed systems design and verification. *Theoretical Computer Science*, 89:179–206, 1991.

Appendix A

Formal Models for Behaviour Definitions

This appendix presents some formal definitions that can be useful for defining a formal semantics to support causality relations and implementation correctness notions. This appendix must be considered as a set of preliminary notes which have been preserved because they may contain interesting information for future elaboration.

A.1 Basic Causality Relations

Basic causality relations are causality relations without explicit attribute value conditions and constraints.

Definition A.1: Let A be a finite set of action identifiers, and BA the boolean logic formed by the terminals a_i, $\neg a_i$, **T** and **F** representing occurrence and non-occurrence of $a_i \in A$, *true* and *false* respectively, and by the combinators $\vee$ and $\wedge$.

A behaviour definition B is a pair $<A, \rho>$, where ρ is a function $\rho : A \text{ -> } BA$, the set of *basic causality relation*. For each $a_i \in A$, $\rho\ (a_i)$ defines the conditions for the occurrence of a_i.

A.2 Valuation Functions

Valuation functions can be useful for defining simulation tools for behaviours based on causality relations.

Definition A.2: The valuation function $v_D : A \cup \tilde{A} \text{ -> } \{\mathbf{T}, \mathbf{F}\}$, where $D \subseteq A$ and $\tilde{A} = \{\neg\, a_i, a_i \in A\}$ is defined as follows:

- $v_D\ (a_i) = \mathbf{F}$, if $a_i \in D$
  $\mathbf{T}$, if $a_i \notin D$
- $v_D\ (\neg a_i) = \mathbf{T}$, if $a_i \in D$
  $\mathbf{F}$, if $a_i \notin D$

The valuation function formally represents that an action has not occurred and therefore can still occur. This function can be used to evaluate conditions of an action, and to define the semantics of our basic behaviour definitions in terms of configurations.

Definition A.3: The valuation function $v_D : BA \rightarrow \{\mathbf{T}, \mathbf{F}\}$ is defined as the upgrading of $v_D : A \cup \tilde{A} \rightarrow \{\mathbf{T}, \mathbf{F}\}$ on the boolean logic *BA*. $v_D(C)$, $C \in BA$ corresponds to the application of $v_D(a_i)$ and $v_D(\neg a_i)$ on each a_i and $\neg a_i$ that appear in *C*.

Enabled Action

Definition A.4: An action a_i is enabled in a behaviour definition (A, ρ) if $v_A(\rho(a_i)) = \mathbf{T}$. This means that the conditions for the occurrence of action a_i are satisfied in this behaviour definition.

A.3 Proving Sequence and Configuration

A proving sequence is a possible sequence in which a sub-set of *A* can happen ([1]). Each action of a proving sequence should be enabled by the actions that happened before this action in the sequence.

Definition A.5: A proving sequence is a sequence of actions $a_1, ..., a_n$, where $a_1, ..., a_n \in$ A, such that $\forall\, a_i \mid i = 1,..., n, v_D(\rho(a_i)) = \mathbf{T}$ where $D = A - \{a_1, ..., a_{i-1}\}$

A configuration is a sub-set of *A* that can be executed by a certain behaviour. The definition of configurations makes it possible to consider the partial orders between actions, in place of the total order imposed by trace and interleaving semantics.

Definition A.6: A configuration is a set $C \subseteq A$, if there is proving sequence $a_1, ..., a_n$ such that $C=\{a_1, ..., a_n\}$.

A.4 Operational Semantics

This operational semantics defines the remainder of a behaviour after the execution of a configuration. In this way one can implement simulators that execute steps consisting of more than one action.

Definition A.7: The remainder $B[C]$ of a behaviour definition $B = <A,\rho>$ after a configuration *C* is defined as follows:

Let $B[C] = (A', \rho')$

- $A' = A - C$
- $\rho' = \rho \ \lceil A' [\forall\, a_i \in C\ (a_i \setminus \mathbf{T}, \neg a_i \setminus \mathbf{F})]$

We could have also removed the elements in $\{ (\rho(a), a) \mid \rho(a) = \mathbf{F}\}$ from the causality function, but then there can be actions in A' that have no value in the causality function.

From this definition, it is not difficult to show that $B[C]$ is also a behaviour definition. According to the definition, ρ': $A' \text{->} BA'$, for elements boolean algebra involving a_j, $\neg a_j$, T, F and the combinators $\vee$ and $\wedge$, where $a_j \in A'$. ρ' has domain A', and all references to a_i and $\neg a_i$ belonging to C have been replaced by **T** and **F** respectively, which corresponds to evaluating all conditions involving actions of the configuration (causality or exclusion).

Alternatively we could remove those actiosn that are made impossible by the configuration:

Let $B[C] = (A', \rho')$

- $A' = A - C - \{a_j \in A \mid \rho(a_j)\ [\forall\ a_i \in C\ (a_i \setminus \mathbf{T}, \neg a_i \setminus \mathbf{F})] = \mathbf{F}\ \}$
- $\rho' = \rho \ \lceil\ A'\ [\forall\ a_i \in C\ (a_i \setminus \mathbf{T}, \neg a_i \setminus \mathbf{F}), \forall a_j \in A\ (\rho(a_j)\ [\forall\ a_i \in C\ (a_i \setminus \mathbf{T}, \neg a_i \setminus \mathbf{F})] = \mathbf{F})$
  $(a_i \setminus \mathbf{F}, \neg a_i \setminus \mathbf{T})$

It should be possible to show that if C is aconfiguration of B, and $D \subseteq A$ such that $C \cap D = \varnothing$, if D is configuration of $B[C] \Leftrightarrow C \cup D$ is configuration of B

A.5 Partial Order Semantics

This partial order semantics for basic causality relations considers a behaviour as a set of partial ordered sets, such that each partial ordered set of a behaviour defines a possible execution of this behaviour, and the partial order of thuis execution defines the forced timing relations between actions.

Definition A.8: An execution p of a behaviour B is a truple $<A, t, \angle>$, where $A \subseteq A_B$, A_B is the set of actions of B, t is a total function t: $A \text{->} \Re$, where $\Re$ is the set of real numbers, and $\angle \subseteq A \times A$ is defined such that $a_1 \angle a_2 \Rightarrow t(a_1) < t(a_2)$. A execution p defines a possible set of actions, their time of occurrence and forced timing dependencies.

A_p, t_p, $\angle_p$ refer to A, t, $\angle$ of a specific execution $p = <A_p, t_p, \angle_p>$

Forced timing dependencies are not always identified to causality, since exclusion also causes forced timing dependencies. For example, in any execution of a behaviour that contains the causality relation $start \wedge \neg\ b \text{->} a$ in which both a and b occur, $a \angle b$. Therefore forced timing dependencies can also be generated by exclusion.

Definition A.9: The semantics of a behaviour B can be defined as a set of executions P of this behaviour. This semantics defines all the possible sets of actions that can occur and their associated time of occurrence.

The semantics of behaviours defined as causality relations in terms of sets of executions can be given in terms of two conditions:

- *safety*: in case an action happens, its conditions were satisfied at the moment of its execution. This implies that the history of execution before the occurrence of this actions has enabled it;
- *liveness*: if it is possible to have an execution of the behaviour in which the conditions of an action are satisfied, then there is at least one execution of the behaviour in which this action

occurs.

Both conditions must hold. The semantics of causality relations are defined in terms of the conditions that these causality relations impose on the sets of executions P of a behaviour. For example:

- $start \text{ -> } a =_{\text{def}} \exists\, p \in P, a \in A_p$ *(* liveness *)*
- $a \text{ -> } b =_{\text{def}} (\forall\, p \in P, b \in A_p \Rightarrow a \in A_p \wedge a \angle_p b$ *(* safety *)* $) \wedge$
  $\exists\, p \in P, a \in A_p \Rightarrow b \in A_p$ *(* liveness *)*
- $\neg\, a \text{ -> } b =_{\text{def}} (\forall\, p \in P, b \in A_p \Rightarrow (a \in A_p \Rightarrow b \angle_p a)$ *(*safety *)* $) \wedge$
  $\exists\, p \in P, b \in A_p$ *(* liveness *)*
- $a \wedge b \text{ -> } c =_{\text{def}} (\forall\, p \in P, c \in A_p \Rightarrow (a \in A_p \wedge a \angle_p c) \wedge (b \in A_p \wedge b \angle_p c)$ *(* safety *)* $) \wedge$
  $\exists\, p \in P, (a \in A_p \wedge b \in A_p) \Rightarrow c \in A_p$ *(* liveness *)*
- $a \vee b \text{ -> } c =_{\text{def}} (\forall\, p \in P, c \in A_p \Rightarrow (a \in A_p \wedge a \angle_p c) \vee (b \in A_p \wedge b \angle_p c)$ *(* safety *)* $) \wedge$
  $\exists\, p \in P, (a \in A_p \vee b \in A_p) \Rightarrow c \in A_p$ *(* liveness *)*
- $a \wedge \neg\, b \text{ -> } c =_{\text{def}} \forall\, p \in P, c \in A_p \Rightarrow (a \in A_p \wedge a \angle_p c) \wedge (b \in A_p \Rightarrow c \angle_p b)$ *(* safety *)* $\wedge$
  $\exists\, p \in P, b \in A_p$ *(* liveness *)*
- $a \vee \neg\, b \text{ -> } c =_{\text{def}} \forall\, p \in P, c \in A_p \Rightarrow (a \in A_p \wedge a \angle_p c) \vee (b \in A_p \Rightarrow c \angle_p b)$ *(* safety *)* $\wedge$
  $\exists\, p \in P, a \in A_p \Rightarrow b \in A_p$ *(* liveness *)*

The semantics of a behaviour is defined in terms of all sets of executions that satisfy these conditions.

Liveness clauses can be alternatively defined in the form $\exists\, p \in P, b \in A_p$ (or $c \in A_p$)

Examples

B := { start -> a, start -> b, a $\wedge$ *b -> c }* has the following possible executions:

nothing happens: $< \emptyset, \emptyset, \emptyset>$
only *a* happens: $<\{a\}, \{<a, t(a)>\}, \emptyset>$
only *b* happens: $<\{b\}, \{<b, t(b)>\}, \emptyset>$
only *a* and *b* happen: $<\{a, b\}, \{<a, t(a)>, <b, t(b)>\}, \emptyset>$
a, b and *c* happen: $<\{a, b, c\}, \{<a, t(a)>, <b, t(b)>, <c, t(c)>\}, \{a \angle c, b \angle c\}>$

B := { start -> a, start -> b, a $\vee$ *b -> c }* has the following possible executions:

nothing happens: $< \emptyset, \emptyset, \emptyset>$
only *a* happens: $<\{a\}, \{<a, t(a)>\}, \emptyset>$
only *b* happens: $<\{b\}, \{<b, t(b)>\}, \emptyset>$
only *a* and *b* happen: $<\{a, b\}, \{<a, t(a)>, <b, t(b)>\}, \emptyset>$,
only *a* and *c* happen: $<\{a, c\}, \{<a, t(a)>, <c, t(c)>\}, \{a \angle c\}>$
only *b* and *c* happen: $<\{b, c\}, \{<b, t(b)>, <c, t(c)>\}, \{b \angle c\}>$
a, b and *c* happen, *a* causes *c*: $<\{a, b, c\}, \{<a, t(a)>, <b, t(b)>, <c, t(c)>\}, \{a \angle c\}>$
a, b and *c* happen, *b* causes *c*: $<\{a, b, c\}, \{<a, t(a)>, <b, t(b)>, <c, t(c)>\}, \{b \angle c\}>$

$B := \{ start \rightarrow a, start \rightarrow b, a \wedge \neg b \rightarrow c \}$ has the following possible executions:

nothing happens: $< \varnothing, \varnothing, \varnothing>$
only *a* happens: $<\{a\}, \{<a, t(a)>\}, \varnothing>$
only *b* happens: $<\{b\}, \{<b, t(b)>\}, \varnothing>$
only *a* and *b* happen: $<\{a, b\}, \{<a, t(a)>, <b, t(b)>\}, \varnothing>$
only *a* and *c* happen: $<\{a, c\}, \{<a, t(a)>, <c, t(c)>\}, a \angle c>$
a, *b* and *c* happen: $<\{a, b, c\}, \{<a, t(a)>, <b, t(b)>, <c, t(c)>\}, \{a \angle c, c \angle b\}>$

The partial orders of each execution should be the transitive closure of the $\angle$ relation, since each execution has a linear time scale. For example, the behaviour definition $\{start \rightarrow a, a \rightarrow c, c \rightarrow b\}$ has the following executions:

nothing happens: $< \varnothing, \varnothing, \varnothing>$
only *a* happens: $<\{a\}, \{<a, t(a)>\}, \varnothing>$
only *a* and *c* happen: $<\{a, c\}, \{<a, t(a)>, <c, t(c)>\}, a \angle c>$
a, *b* and *c* happen: $<\{a, b, c\}, \{<a, t(a)>, <b, t(b)>, <c, t(c)>\}, \{a \angle c, c \angle b, a \angle b\}>$

In this case, if we want to find the abstraction of this behaviour when ignoring action *c*, which has been characterized in Chapter 6 by elimination rule 1, we can simply remove references to *c* in the executions, which results in the executions:

nothing happens: $< \varnothing, \varnothing, \varnothing>$
only *a* happens: $<\{a\}, \{<a, t(a)>\}, \varnothing>$
a and *b* happen: $<\{a, b\}, \{<a, t(a)>, <b, t(b)>\}, a \angle b>$

These are the executions of $\{ start \rightarrow a, a \rightarrow b \}$, which is exactly the abstract behaviour obtained using elimination rule 1.

A.6 Modality or Probability

Modality can also be expressed using set of executions. We simplify the notation that represents modality in the following way:

$$a \rightarrow_{\Diamond} b =_{def} a \rightarrow b\ [p_b = x]\ , \text{ where } x < 100\%$$

$$a \rightarrow_{\Box} b =_{def} a \rightarrow b\ [p_b = 100\%]$$

We define the conditions on the sets of executions of a behaviour containing these causality relations in the following way:

$$a \rightarrow_{\Diamond} b =_{\text{def}} (\forall p \in P, b \in A_p \Rightarrow a \in A_p \wedge a \angle_p b) \wedge \exists p \in P, a \in A_p \Rightarrow b \in A_p$$

$$a \rightarrow_{\Box} b =_{\text{def}} (\forall p \in P, b \in A_p \Rightarrow a \in A_p \wedge a \angle_p b) \wedge \forall p \in P, a \in A_p \Rightarrow b \in A_p$$

A.7 Execution Equivalence

Two behaviour definitions can be considered equivalent if they have the same sets of executions.

Example

In [2] we find the following proposition: Asymmetric Conflict = Symmetric Conflict + Causality.

Using causality relations this implies that, for example, $B: = \{ start \rightarrow a, start \wedge \neg a \rightarrow b\}$ (only asymmetric conflict) is equivalent to $B: = \{ b \vee \neg b \rightarrow a, start \wedge \neg a \rightarrow b\}$ (symmetric conflict + causality $b \rightarrow a$)

Inspecting the sets of executions generated by both behaviours we come to the conclusion that both behaviours generate:

nothing happens: $< \emptyset, \emptyset, \emptyset>$
only *a* happens: $<\{a\}, \{<a, t(a)>\}, \emptyset>$
only *b* happens: $<\{b\}, \{<b, t(b)>\}, \emptyset>$
a and *b* happen: $<\{a, b\}, \{<a, t(a)>, <b, t(b)>\}, \{b \angle a\}>$

A.8 References

[1] R. Langerak. *Transformations and Semantics for LOTOS*. PhD thesis, University of Twente, Enschede, Netherlands, 1992.

[2] A. Rensink. *Models and Methods for Action Refinement*. PhD thesis, University of Twente, Enschede, The Netherlands, 1993.

Appendix B
Additional Consulted References

This appendix presents a list of references that have been consulted during the development of this thesis but have not been mentioned in any chapter:

L. Aceto and M. Hennessy. Adding action refinement to a finite process algebra. In J. Leach Albert, B. Monien, and M. Rodríguez Artalejo, editors, *Automata, Languages and Programming*, volume 510 of *Lecture Notes in Computer Science*, pages 506–519, Germany, 1991. Springer-Verlag.

T. Bolognesi, F. Lucidi, and S. Trigila. From timed Petri nets to timed LOTOS. In L. Logrippo, R. Probert, and H. Ural, editors, *Protocol Specification, Testing and Verification X*, pages 377–406. Elsevier Science Publishers B.V. (North-Holland), 1990.

E. Brinksma. From data structure to process structure. In *CAV'91 - Participants Proceedings*, 1991.

E. Brinksma. What is the method in formal methods? In *FORTE'91*, 1991.

E. Brinksma, B. Jonsson, and F. Orava. Refining interfaces of communicating systems. In S. Abramksy and T. Maibaum, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development, TAPSOFT'91*, volume 494 of *Lecture Notes on Computer Science*, pages 297–312. Springer-Verlag, 1991.

R. Coelho da Costa and J.-P. Courtiat. A true concurrency semantics for LOTOS. In M. Diaz and R. Groz, editors, *Formal Description Techniques, V*, pages 347–378. North-Holland, 1993.

I. Czaja, R. J. van Glabbeek, and U. Goltz. Interleaving semantics and action refinement with atomic choice. In G. Rozenberg, editor, *Advances in Petri Nets 1992*, volume 609 of *Lecture Notes in Computer Science*, pages 80–107, Germany, 1992. Springer-Verlag.

E. Dubuis. An algorithm for translating LOTOS behaviour expressions into automata and ports. In S. T. Voung, editor, *Formal Description Techniques, II*. North-Holland, 1990.

L. Ferreira Pires and W. L. de Souza. Step-wise refinement design example using LOTOS. In J. Quemada, J. M. nas, and E. V'azquez, editors, *Formal Description Techniques, III*, pages 255–262, The Netherlands, 1991. Elsevier Science Publisher B.V. (North-Holland).

L. Ferreira Pires and J. Schot. Systematic design of a network gateway using the FDT LOTOS. In *IEEE Infocom'91*, volume 3, pages 1344–1352. IEEE Communications Society, 1991.

L. Ferreira Pires and J. Schot. Design and implementation strategies. In T. Bolognesi, E. Brinksma, and C. A. Vissers, editors, *Third Lotosphere Workshop and Seminar, Workshop Proceedings*, volume 2, 1992.

R. Gotzhein. The formal definition of the architectural concept 'interaction point'. In *Participant's Proceedings of FORTE'89*, pages 84–98, 1989.

R. Gotzhein. Temporal logic and applications – a tutorial. *Computer Networks and ISDN Systems*, 24:203–218, 1992.

J. Gunawardena. Causal automata I: Confluence $equiv$ {and,or} casuality. In M. Kwiatkowska, M. Shields, and R. Thomas, editors, *Semantics for Concurrency, Leicester 1990*, pages 137–156, Great-Britain, 1990. Springer-Verlag.

J. Gunawardena. Gemeotric logic, causality and event structures. In J. Baeten and J. Groote, editors, *CONCUR'91, Second International Conference on Concurrency Theory*, volume 494 of *Lecture Notes on Computer Science*, pages 266–280, Berlin, 1991. Springer-Verlag.

J. Gunawardena. Causal automata. *Theoretical Computer Science*, 101:265–288, 1992.

M. P. Herlihy and J. M. Wing. Reasoning about atomic objects. In M. Joseph, editor, *Formal Techniques in Real Time and Fault-Tolerant Systems*, volume 331 of *Lecture Notes in Computer Science*, pages 193–208. Springer-Verlag, 1988.

G. Karjoth. Implementing process algebra specifications by state machines. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing, and Verification, VIII*, pages 47–60. North-Holland, June 1988.

G. Karjoth. A compilation of algebraic processes based on extended-action derivation. In J. Quemada, J. M. as, and E. Vazquéz, editors, *Formal Description Techniques, III*, pages 127–140. North-Holland, 1991.

L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

L. Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, January 1989.

R. Langerak. Decomposition of functionality: a correctness preserving LOTOS transformation. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Protocol Specification, Testing and Verification, X*, pages 229–243. North-Holland, 1990.

R. Langerak. Even structures for design and transformation using LOTOS. In K. Parker and G. Rose, editors, *Formal Description Techniques, IV.* North-Holland, 1992.

R. Langerak. Bundle event structures: a non-interleaving semantics for LOTOS. In M. Diaz and R. Groz, editors, *Formal Description Techniques, V*, pages 331–346. North-Holland, 1993.

G. Leduc. An upward compatible timed extension to LOTOS. In K. Parker and G. Rose, editors, *Fourth International Conference on Formal Description Techniques*, 1991.

J. Mañas, T. de Miguel, J. Salvachúa, and A. Azcorra. Tool support to implement LOTOS formal specifications. *Computer Networks and ISDN Systems*, 1992.

J. Mañas and J. Salvachúa. Λβ: a virtual LOTOS machine. In K. Parker and G. Rose, editors, *Fourth International Conference on Formal Description Techniques, Participant's Proceedings*, pages 445–460, Sydney, Australia, November 1991.

R. Milner. *Communication and concurrency*. Prentice Hall International Series in Computer Science. Prentice Hall International, Great-Britain, 1989.

X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In *CAV'91*, July 1991.

D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053 – 1058, December 1972.

J. Parrow and P. Sjödin. Multiway synchronization verified with coupled simulation. In W. Cleaveland, editor, *CONCUR'92, Third International Conference on Concurrence Theory*, volume 630 of *Lecture Notes In Computer Science*, Germany, 1992. Springer-Verlag.

V. Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.

J. Quemada and A. Azcorra. Structuring protocols using exceptions in a lotos extension. In *Protocol Specification Testing and Verification, XII*, 1992.

J. Quemada and A. Fernandez. Introduction of quantitative relative time into LOTOS. In H. Rudin and C. West, editors, *Protocol Specification, Testing, and Verification, VII*, pages 105–121, The Netherlands, 1987. Elsevier Science Publishers B.V. (North-Holland).

E. Rechtin. The art of systems architecting. *IEEE Spectrum*, pages 66–69, October 1992.

R. Sisto, L. Ciminiera, and A. Valenzano. A protocol for multirendezvous of LOTOS processes. *IEEE Transactions on Computers*, 40(1):437–447, April 1991.

L. Svobodova. Implementing OSI systems. *IEEE Journal on Selected Areas in Communications*, 7(7):1115–1130, September 1989.

K. J. Turner. A LOTOS-based development strategy. In *Formal Description Techniques II*, the Netherlands, 1990. North-Holland.

K. J. Turner, editor. *Using Formal Description Techniques. An Introduction to Estelle, LOTOS and SDL*. John Wiley & Sons, 1993.

G. v. Bochmann, G. W. Gerber, and J.-M. Serre. Semiautomatic implementation of communication protocols. *IEEE Transactions on Software Engineering*, SE-13(9):989–1000, September 1987.

F. Vaandrager and N. Lynch. Action transducers and timed automata. In W. Cleaveland, editor, *CONCUR'92. Third International Conference on Concurrency Theory, Stony Brook, NY, USA, August 1992, Proceedings*, volume 630 of *Lecture Notes in Computer Science*, pages 436–455, Germany, 1992. Springer-Verlag.

W. H. van Hulzen, P. A. Tilanus, and H. Zuidweg. LOTOS extended with clocks. In *FORTE '89. Participant's Proceedings*, 1989.

M. van Sinderen, L. Ferreira Pires, and C. A. Vissers. Protocol design and implementation using formal methods. *The Computer Journal*, 35(5):478–491, Oct. 1992.

C. A. Vissers. Standardisation of formal description techniques for communication protocols. In H.-J. Kugler, editor, *Information processing 86*, pages 321–328. North-Holland, 1986.

C. A. Vissers. FDTs for open distributed systems, a retrospective and a prospective view. In *Protocol Specification, Testing and Verification X*, 1990.

C. A. Vissers and L. Logrippo. The importance of the service concept in the design of data communications protocols. In M. Diaz, editor, *Protocol Specification, Testing, and Verification, V*, pages 3–17, Netherlands, 1986. Elsevier Science Publishers B.V. (North-Holland).

C. A. Vissers, J. van de Lagemaat, and L. Ferreira Pires. Formal description techniques for distributed computing systems: the challenges for the 1990's. In *Future Trends 90's, Second IEEE Workshop on Future Trends of Distributed Computer Systems*, pages 465–471. IEEE Computer Society Press, 1990.

W. Vogler. Bisimulation and action refinement. *Theoretical Computer Science 114*, pages173–200, 1993.

D. Weber-Wulff. Selling formal methods to industry. In J. Woodcock and P. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*, pages 671–678, Berlin, 1993. Springer-Verlag.

S. White, M. Alford, J. Holtzman, S. Kuehl, B. McCay, D. Oliver, D. Owens, C. Tully, and A. Willey. Systems engineering of computer-based systems. *Computer*, pages 54–65, Nov. 1993.

G. Winskel. An introduction to event structures. In J. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Logic, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 364–397, Germany, 1989. Springer-Verlag.

P. Zave. The operational versus the conventional approach to software development. *Communications of the ACM*, 27(2):104–118, February 1984.

Index

Summary

This thesis develops a framework of methods and techniques for distributed systems development. This framework consists of two related domains in which design concepts for distributed systems are defined: the entity domain and the behaviour domain. In the entity domain we consider structures of functional entities and their interconnection, while in the behaviour domain we consider behaviour definition and structuring.

An interaction in which we abstract from the particular responsibilities of the participating functional entities is considered as an action. Behaviours consist of actions, interactions and their relationships. Relationships between actions and interactions are defined in terms of causality relations. In each causality relation the conditions and constraints for an action or interaction to occur are defined. Two important behaviour structuring techniques have been identified from the possible ways causality relations can be distributed: causality-oriented behaviour composition and constraint-oriented behaviour composition.

Causality-oriented behaviour composition consists of placing some conditions of an action and the action itself in different sub-behaviours. Constraint-oriented behaviour composition consists of placing parts of the conditions and constraints of an action in different sub-behaviours, such that this action is shared by these sub-behaviours.

This thesis identifies milestones in the design process of distributed systems, as well as the design steps to move from one milestone to another. These design steps are characterized using the concepts of the entity and the behaviour domain. We identified two crucial design operations of the behaviour domain that support these design steps: behaviour refinement and action refinement.

Behaviour refinement consists of introducing (internal) structure in the causality relations of reference actions of an abstract behaviour, but preserving their causality and exclusion relationships and their attribute values. Action refinement consists of replacing abstract actions by activities, such that the completion of these activities correspond to the occurrence of the abstract actions. One important characteristic of action refinement is the possibility of distributing attribute values of the abstract actions over actions of the activities that replace them in the concrete behaviours.

The area of research, scope and objectives of this thesis are discussed in Chapter 1. The concept of design culture and its elements is introduced in this chapter in order to provide an overview of the important aspects of the design process. Entity domain, behaviour domain, and design milestones are introduced and discussed in Chapter 2. This chapter also discusses the global objectives of design steps, and the abstraction obtained by considering interactions between cooperating functional entities as actions of the interaction system between these entities. Action, action attributes, causality and exclusion are discussed in Chapter 3. This chapter shows how a behaviour can be defined in terms of the causality relations of its actions in a monolithic form.

Causality-oriented behaviour composition is discussed in Chapter 4. Entries and exits of a behaviour are the mechanisms that make it possible to assign parts of a condition of an action and the action itself to different sub-behaviours. Constraint-oriented behaviour composition is discussed in Chapter 5. Decomposition possibilities of monolithic behaviours are systematically studied in this chapter.

Behaviour refinement is discussed in Chapter 6. This chapter defines a method to obtain an abstraction of a concrete behaviour. This method can be used to check whether the concrete behaviour corresponds to a certain abstract behaviour. Action refinement is discussed in Chapter 7. This chapter identifies some activity forms, and define the rules for considering these activities as implementations of an abstract action. These rules are used in a method to derive an abstraction of a concrete behaviour in which the abstract actions are implemented as activities. This method can be used to check whether the concrete behaviour corresponds to a certain abstract behaviour.

Chapter 8 discusses a design example that is meant to illustrate the use of our design concepts. The example is an interaction server, which is a component that supports the interaction between multiple functional entities. Chapter 9 draws some conclusions and revisits the design milestones of Chapter 2, showing alternatives for the design trajectory which have been created with the use of actions and interactions in a single framework.

Samenvatting

In dit proefschrift wordt een raamwerk ontwikkeld, dat bedoeld is om het ontwerpen van gedistribueerde systemen te ondersteunen. Dit raamwerk bevat twee gerelateerde domeinen waarin concepten voor gedistribueerde systemen gedefinieerd kunnen worden: het entiteit-domein en het gedragsdomein. In het entiteit-domein worden de structuren van functionele entiteiten en hun koppeling beschouwd, terwijl in het gedragsdomein worden gedragingen en gedragstructuren beschouwd.

Een interactie waarin wij abstracteren van de verantwoordelijkheden van de deelnemende functionele entiteiten is beschouwd als een actie. Gedragingen bestaan uit acties, interacties en hun relaties. Relaties tussen acties en interacties zijn gedefinieerd in termen van causaliteit relaties. In elke causaliteit relatie zijn de condities en beperkingen voor het gebeuren van een actie gedefinieerd. Twee belangrijke structureringstechnieken voor gedragingen zijn geïdentificeerd: causaliteit-georiënteerde gedragscompositie en 'constraint'-georiënteerde gedragscompositie.

Causaliteit-georiënteerde gedragscompositie bestaat uit het splitsen van causaliteit relaties zodanig dat sommige condities van acties en deze acties zijn geplaatst in verschillende sub-gedragingen. 'Constraint'-georiënteerd gedragscompositie bestaat uit het splitsen van causaliteit relaties zodanig dat condities en beperkingen van acties zijn geplaatst in verschillende sub-gedragingen. In dit geval zijn deze acties gezamenlijk uitgevoerd door de sub-gedragingen.

Dit proefschrift identificeert mijlpalen in het ontwerpproces van gedistribueerde systemen, en ook de ontwerpstappen die bedoeld zijn om van een mijlpaal naar een volgende te gaan. Deze ontwerpstappen zijn gedefinieerd met behulp van de concepten van het entiteit- en gedragsdomein. Wij identificeren twee cruciale ontwerpoperaties die ontwerpstappen ondersteunen: gedragsverfijning en actieverfijning.

Gedragsverfijning bestaat uit het introduceren van (interne) structuur in de causaliteitsrelaties van referentie-acties van een abstract gedrag, waarin zowel causaliteit en uitsluiting eigenschappen van deze acties als de waarde van de attributen van deze acties behouden. Actieverfijning bestaat uit het vervangen van abstracte acties door activiteiten, zodanig dat de voltooiing van deze activiteiten correspondeert met het gebeuren van de abstracte acties. Een belangrijke eigenschap van actieverfijning is de mogelijkheid om attribuut waarden van abstracte acties te distribueren over acties van de activiteiten die de abstracte acties implementeren.

Het onderzoekgebied, de grenzen en de doelstellingen van dit proefschrift zijn besproken in Hoofdstuk 1. Het concept van ontwerpcultuur is daarin geïntroduceerd om een overzicht te geven van de belangrijke aspecten van het ontwerpproces. Entiteit-domein, gedragsdomein en ontwerpmijlpalen zijn besproken in Hoofdstuk 2. Dit hoofdstuk bespreekt ook de globale doelen van ontwerpstappen en de abstractie die verkregen wordt door het beschouwen van interacties tussen samenwerkende functionele entiteiten als acties van hun interactiesysteem. Acties, actieattributen, causaliteit en uitsluiting zijn besproken in Hoofdstuk 3. Dit hoofdstuk toont aan hoe een gedrag gedefinieerd kan worden in termen van de causaliteit relaties van zijn acties op een monolithische manier.

Causaliteit-georiënteerde gedragscompositie is besproken in Hoofdstuk 4. ‘Entries’ en ‘exits’ van gedragingen zijn de mechanismen die het mogelijk maken om delen uit een conditie van een actie en een actie zelf toe te wijzen aan verschillende sub-gedragingen. ‘Constraint’-georiënteerde gedragscompositie is besproken in Hoofdstuk 5. Decompositiesmogelijkheden voor monolithische gedragingen zijn systematisch bestudeerd in dit hoofdstuk.

Gedragsverfijning is besproken in Hoofdstuk 6. Dit hoofdstuk definieert een methode voor het vaststellen van de abstractie van een concreet gedrag. Deze methode kan gebruikt worden om te controleren of een concreet gedrag een abstract gedrag implementeert. Actieverfijning is besproken in Hoofdstuk 7. Dit hoofdstuk identificeert activiteit vormen, en definieert de regels voor het beschouwen van een activiteit als een implementatie van een actie. Deze regels zijn gebruikt in een methode voor het vaststellen van de abstractie van een concreet gedrag. Deze methode kan gebruikt worden om te controleren of een concreet gedrag een abstract gedrag implementeert.

Hoofdstuk 8 bespreekt een ontwerpvoorbeeld bedoeld om de ontwerpconcepten van de voorgaande hoofdstukken te illustreren. Dit voorbeeld is een interactie ‘server’, d.w.z. een component waarmee interacties tussen functionele entiteiten ondersteund worden. Hoofdstuk 9 presenteert onze conclusies en bespreekt alternatieve ontwerptrajecten die geschapen zijn met het gebruik van acties en interacties in hetzelfde raamwerk.

Curriculum Vitae

7 april 1961:	geboren te São Paulo, Brazilië
1979 - 1983:	studie Elektrotechniek ‘Instituto Tecnológico de Aeronáutica’, São José dos Campos, Brazilië
1984 - 1989:	master programma ‘Escola Politécnica da Universidade de São Paulo’, São Paulo, Brazilië
1982-1984:	stagiair Instituto Nacional de Pesquisas Espaciais São José dos Campos, Brazilië
1984 - feb’ 1988:	ingenieur ITAUTEC Informatica S.A. São Paulo, Brazilië
maart 1988 - sept’ 1992:	medewerker onderzoek vakgroep Tele-Informatica en Open Systemen Universiteit Twente
oct’92 - heden:	universitair docent vakgroep Tele-Informatica en Open Systemen Universiteit Twente