# On the Role of Basic Design Concepts in Behaviour Structuring

Dick A.C. *Quartel*[1], Luís *Ferreira Pires*[1], Marten J. *van Sinderen*[1],
Henry M. *Franken*[2], Chris A. *Vissers*[1, 2]

(1) Centre for Telematics and Information Technology,
Tele-Informatics and Open Systems Group, University of Twente
PO Box 217, 7500 AE Enschede, The Netherlands
e-mail: {quartel, pires, sinderen}@cs.utwente.nl
(2) Telematics Research Centre
PO Box 589, 7500 AN Enschede, The Netherlands
e-mail: {H.Franken, C.Vissers}@trc.nl

*Abstract*
This paper presents some basic design concepts for the design of open distributed systems. These concepts should form the basis for the development of effective design methodologies. The paper discusses how design concepts, such as interaction, action and causality relation, can be used for modelling and structuring behaviours of functional entities in a distributed environment. The paper also addresses some consequences of the application of these design concepts such as the choice of language elements and operations to represent behaviour structure, the structuring of the design process, and the definition of design operations for behaviour refinement.

*Keywords*: design concepts, interactions, actions, causality relations, behaviour specification, behaviour structuring techniques, open distributed systems, behaviour refinement, design milestones.

## 1 Introduction

Many research activities on systematic approaches to distributed systems design are being carried out, such as the investigation of design methods, the development of conceptual frameworks (e.g. ODP [5]), and the development of Formal Description Techniques (FDTs). These activities should be carried out in the scope of a comprehensive design methodology, based on common principles and objectives.

The results obtained with a design methodology are largely determined by the choice, correct understanding and precise definition of its basic design concepts ([10]). Basic design concepts model elementary and common characteristics of different system implementations, abstracting from characteristics which are irrelevant to the fundamental purpose of the system. In this way, basic design concepts facilitate the designer to conceive, structure and refine the essential characteristics of a system.

This paper introduces and discusses the concepts necessary to model and structure behaviours of functional entities in a distributed environment. In particular we elaborate on the concepts of interaction, action and causality relation. This paper also addresses some consequences of the introduction of these concepts, such as the choice of language elements and operations to represent behaviour structure, the structuring of the design process, and the definition of design operations for behaviour refinement.

The remaining of this paper is structured as follows: section 2 discusses the interaction concept and its representation by three standard FDTs, section 3 presents some improved architectural insights related to the interaction concept and discusses the basic concepts of action and causality relation, section 4 applies these concepts to define the basic design operations of action refinement and causality refinement, section 5 illustrates our ideas with an example, and section 6 presents some conclusions and directions for further research.

## 2 The interaction concept

The concepts of interaction and temporal ordering are important concepts in the design of open distributed systems. FDTs such as LOTOS, Estelle and SDL [8] should be able to represent these concepts. The key role of the interaction concept can be understood by adopting a design approach in which open distributed systems are designed from the definition of the interaction systems between their distributed parts.

### 2.1 Interaction systems

Our approach to system design is based on a careful consideration of the system concept. A generic definition of a system can be found in Webster's dictionary:

> *A system is a regularly interacting or interdependent group of items forming a unified whole.*

This definition indicates two different perspectives of a system: an integrated and a distributed perspective. The integrated perspective considers a system as a whole or black box. This perspective only defines *what* function is performed by a system. The distributed perspective defines *how* this function is performed by an internal structure defined in terms of system parts and their relationships. Figure 1 depicts both system perspectives.

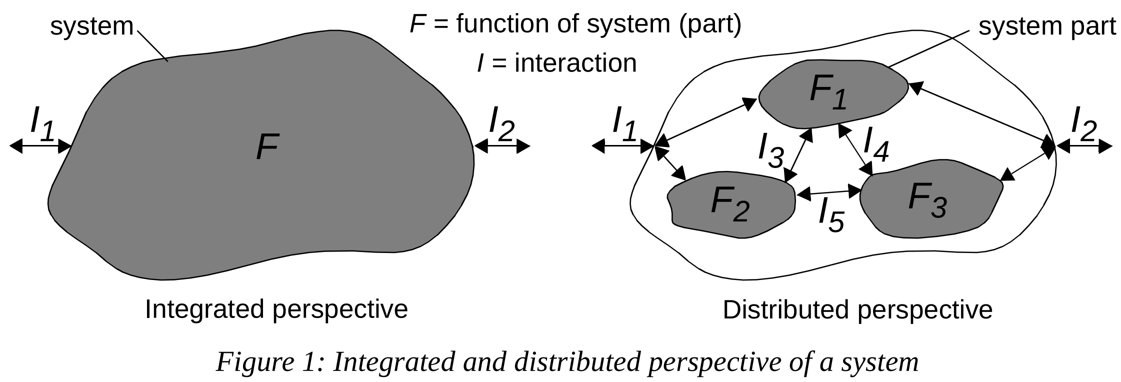



*Figure 1: Integrated and distributed perspective of a system*

Repeated application of these system perspectives provides a basis for a top-down design approach. Initially one defines the system functions. Subsequently, one defines how these system functions can be provided in terms of sub-functions provided by system parts from a distributed perspective. This process can be applied recursively to the identified system parts, until a direct mapping onto available implementation components becomes possible.

This design approach can be directly applied in case the relationships between system parts are rather straightforward, such that interactions between these parts can be defined implicitly by their composition. However, in case the relationships between system parts are more complex, the interactions between these parts should be explicitly designed. The concept of interaction system is introduced for this purpose.

**Definition**

An interaction system is defined as the mechanism that makes the interaction between system parts possible [11]. An interaction system divides each system part in two functions:

- an *application function*, which uses the interaction system to communicate with application functions of other system parts; and
- a *protocol function*, which provides the functionality of the interaction system between system parts in cooperation with the protocol functions of other system parts.

Similarly to systems, interaction systems can be also considered from an integrated and a distributed perspective. Figure 2 depicts both perspectives of an interaction system.

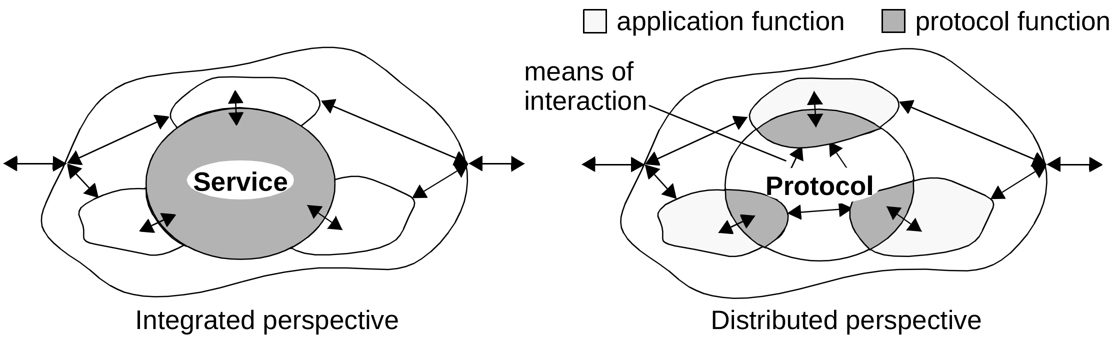



*Figure 2: Integrated and distributed perspective of an interaction system*

The integrated perspective of an interaction system is called a *service*. Traditionally a service is defined as the observable behaviour of the service provider, in terms of the interactions that may occur at the interfaces of the service provider and the relationships between these interactions. In section 3.5 we give a more precise interpretation of the concepts of service and service provider.

The integrated perspective abstracts from the individual protocol functions and their cooperation. The composition of protocol functions is considered as a whole in the integrated perspective, similarly to the system from the integrated perspective as depicted in Figure 1.

The distributed perspective of an interaction system is called a *protocol*. A protocol is a more detailed definition of the service, which defines *how* the interaction system supports the service, in terms of the contribution of the individual protocol functions and their cooperation. Distributed protocol functions cooperate via some communication mechanism, which is called the *means of interaction* of the protocol functions. Figure 2 represents the abstract means of interaction between protocol functions by arrows interconnecting them.

**Design approach**

Repeated application of the service and protocol concepts forms the basis of a top-down design approach for interaction systems. The protocol functions of system parts can be again decomposed into protocol layer functions and lower level protocol functions. The lower level protocol functions, and their means of interaction, constitute one or more lower level interaction systems. Similar to the original interaction system, these lower level interaction systems can be considered as a service and a protocol, and further decomposed.

Figure 3 illustrates a specific example in which an interaction system is decomposed into a layer of protocol functions and a single lower level interaction system. This corresponds to the decomposition of a service into Protocol Entities (PEs) and a Lower Level Service (LLS), such as in the OSI/RM [4]. Recursive decomposition of the lower level service renders a stack of protocol layers, until a lower level service that is supported by available implementation components is reached. In the OSI/RM this lower level service is the service supported by the transmission medium.

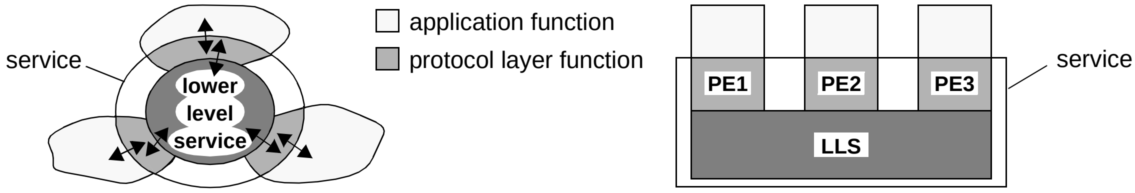



*Figure 3: Alternative representations of service and protocol systems*

A system design approach based on the design of interaction systems is particularly suited for service and protocol systems, in which the interaction between system parts plays an important role.

## 2.2 Interaction characteristics

The interaction concept is a basic building brick for the design of interaction systems. The modelling power of the interaction concept provides a designer with the ability to model the relevant characteristics of interaction systems at suitable abstraction levels. We define an interaction as:

> *a unit of common activity shared by multiple system parts, through which cooperation between these system parts takes place for the purpose of establishing and exchanging information.*

This section discusses the following basic characteristics of interactions: reliability, time and value establishment.

**Reliability**

The interaction concept represents some unit of behaviour at a certain abstraction level that cannot be split at this abstraction level. This property is called atomicity and is inherent to the choice of an interaction as a basic design concept. The atomicity property imposes that an interaction should be implemented reliably, such that:

- either an interaction happens, which means that all involved system parts can refer to the interaction occurrence and to the information values that have been established;
- or an interaction does not happen, which means that none of the potentially involved system parts can refer to the interaction occurrence or to any information values that could have been established in the interaction.

The reliability requirement is based on the fundamental assumption that a design should be considered as a *prescription* for implementation. A designer should be able to assume that some activity on information that is defined at an abstract level, can also be made to happen in the implementation. At an abstract level a designer neither wants to be concerned with the many different ways in which an implementation may provide some prescribed behaviour, nor with the many different ways in which an implementation may fail to provide this behaviour.

Consequently a designer may only define some shared activity by means of a single interaction if this activity can be implemented reliably. If the reliable implementation of an interaction that models an activity cannot be guaranteed, this activity should be modelled by a composition of multiple interactions, making unreliability explicit.

**Time**

The interaction concept only considers the moment of time at which an interaction occurs, which is defined as the first moment of time when all of the involved system parts can refer to the information values that have been established. The interaction concept considered in this paper abstracts from other time aspects that may be associated with an interaction, such as the starting time or duration of an interaction. This is motivated by the need to model *what* is established by all possible implementations of an interaction, and *when* it can be referred to, rather than how this is achieved.

**Value establishment**

An interaction models the establishment of information values that are shared by all the system parts involved in the interaction. Each individual system part may impose conditions on the information values that can be established.

An information value can be established if all conditions that are imposed on this value can be satisfied. Therefore, an interaction models the result of a negotiation between the conditions of all involved system parts. An interaction cannot happen if the superposition of the conditions on its information values is not satisfied by any values.

Considering the negotiation of an individual information value between two system parts, three basic forms of value establishment can be identified:

- *value checking*: when both system parts require that a single prescribed value of a certain information type should be established;
- *value passing*: when one system part requires that some prescribed value of a certain information type is established, while the other system part allows any value of that type to be established;
- *value generation*: when both system parts allow multiple alternative values of a certain information type to be established.

Various combinations of these basic interaction conditions are possible. Furthermore, they can be extended in a straightforward way to the interaction between more than two system parts. For example, one system part may prescribe a single value of a certain information type, a second part may allow a set of values of that information type, while a third part is willing to accept any value of that information type.

In case the superposition of all interaction conditions implies that multiple values are possible, a non-deterministic choice is made between these values.

## 2.3 LOTOS representation

The FDT LOTOS [1], [8] supports the specification of interactions between system parts, and the specification of their temporal ordering and value dependency relationships. The semantics of a LOTOS interaction, which is called an *event*, corresponds to the definition of the interaction concept discussed above.

An event is specified in terms of the synchronization of multiple *event offers*. An event offer of some system part defines the contribution of this system part in terms of its interaction conditions. The matching of event offers from different system parts into an interaction is specified in LOTOS by prescribing that such offers be made at a common abstract location. This abstract location is called the *event gate*.

The following specification illustrates an interaction in LOTOS, which consists of the parallel composition of three event offers:

```
g !3 ?b:Bool; B1 || g ?x:Nat !true; B2 || g ?y:Nat ?b:Bool [(y ge 1) and (y le 5)]; B3
```

In this example, if an interaction at gate `g` happens then values `3` of sort `Nat` (natural number) and `true` of sort `Bool` (boolean) are established.

**Specification of interaction systems**

The service of Figure 2 can be specified in LOTOS in terms of the observable behaviour of the service provider. The observable behaviour of a system part is defined by the event offers of that system part at the interfaces with its environment, and the relationships between these event offers. In this case, the service provider and the application functions constitute each others environment.

LOTOS processes allow the specification of the observable behaviour of system parts. The following LOTOS specification presents a definition of the top-level behaviour structure of the interaction system service. The abstract locations of the interfaces between the service provider and application functions are represented by the event gates `a`, `b` and `c`.

```
specification Service : noexit:=
behaviour
 ServiceProvider[a,b,c]
where ...
endspec (* Service *)
```

The protocol of Figure 3 can be specified in LOTOS in terms of the composition of the observable behaviour of the protocol layer functions or protocol entities, and the observable behaviour of the lower level service. The environment of the protocol layer functions consists of the application functions and the lower level service.

The following LOTOS specification defines the top-level behaviour structure of the interaction system protocol. Since a protocol implements a service, the behaviour of the protocol should be observable equivalent to the behaviour of the implemented service.

```
process Protocol[a,b,c] : noexit:=
 hide x,y,z in  ( PE1[a,x] ||| PE2[b,y] ||| PE3[c,z] )
                |[x,y,z]| LowerLevelService[x,y,z]
where ...
endproc (* Protocol *)
```

## 2.4 Estelle and SDL representation

The FDTs Estelle and SDL [8] support the specification of the concept of *asynchronous* interaction as basic design concept. Asynchronous interactions are called *signals* and *interactions* in SDL and Estelle, respectively.

An asynchronous interaction corresponds to an instance of message exchange between system parts. This implies that system parts do not directly interact, but via some medium. Both Estelle and SDL use *channels* as medium to allow communication between system parts. These channels are infinite queues which avoids deadlock of the synchronous state machines that are used by these languages to model the behaviour of system parts.

Figure 4 depicts an example of an Estelle interaction and of an SDL signal.

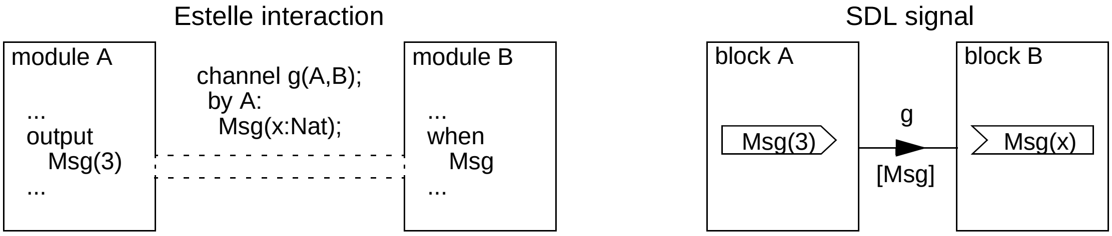



*Figure 4: Examples of asynchronous interaction in Estelle and SDL*

In contrast with asynchronous interaction, the interaction concept of section 2.2 defines *synchronous* interaction between system parts, since synchronization between the system parts involved in an interaction is needed to perform a common activity.

### Synchronous model of asynchronous interaction

Asynchronous interaction between two system parts can be modelled by a synchronous interaction between the sending part and a channel followed by a synchronous interaction between the channel and the receiving part. A LOTOS specification of this model is given below which assumes that a single value (of sort `Nat`) is communicated between a sending and a receiving system part.

```
specification AsynchronousInteraction : noexit:=
behaviour
  SendingPart[g1] |[g1]| Channel[g1,g2] |[g2]| ReceivingPart[g2]
where
  process SendingPart[g1]:noexit:= ... g1!3; ... endproc (* SendingPart *)
  process ReceivingPart[g2]:noexit:= ... g2?x:Nat; ... endproc (* ReceivingPart *)
  process Channel[g1,g2]:noexit:= ... g1?x:Nat; g2!x; ... endproc (* Channel *)
endspec (* AsynchronousInteraction *)
```

In order to model synchronous interaction between distributed parts some composition of multiple asynchronous interactions has to be designed. For example, a handshaking mechanism can be used to implement a synchronous interaction. Therefore, a language which is based on an asynchronous interaction model can only be used to specify specific implementations of synchronous interactions.

### Specification of interaction systems

The observable behaviour of a system part can be specified in terms of a module in Estelle and a block in SDL, by defining the input and output interactions or signals of that system part, and the relationships between these interactions or signals, respectively. This implies that the service of Figure 2 can be specified in Estelle and SDL in terms of a synchronous state machine relating input and output interactions or signals, respectively.

Interfaces between system parts should be defined in Estelle and SDL by means of asynchronous interactions through infinite queues. Therefore implementations that comply to Estelle or SDL specifications must implement sufficiently long queues as the means of interaction between system parts. Figure 5 illustrates the local interfaces between the application functions and the service in terms of their implementation with queues.

The above discussion also applies to the specification in Estelle and SDL of the protocol of Figure 3. The decomposition of the service implies the introduction of channels between the protocol layer functions and the lower level service.

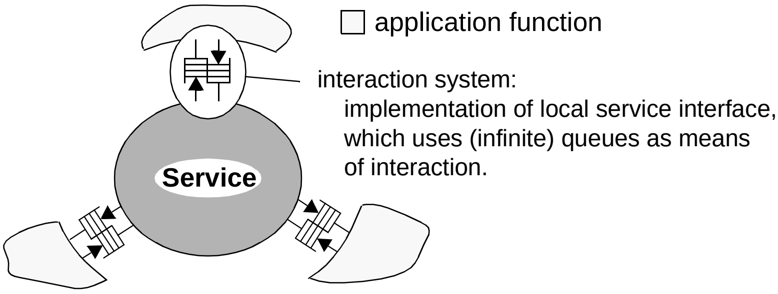



*Figure 5: Specification of service interfaces in Estelle and SDL*

## 2.5 Consequences of interaction representations

The interaction representation of LOTOS and Estelle/SDL have important consequences for the structuring of system behaviours. These are discussed below.

**Modelling power**

A synchronous interaction model has a higher modelling power than an asynchronous interaction model. Asynchronous interactions can be modelled by a composition of synchronous interactions at a corresponding abstraction level, but synchronous interaction can only be implemented by a composition of asynchronous interactions at a more detailed abstraction level.

For example, the relevant characteristics of the administration of connection endpoint identifiers (CEI) can be specified at an abstract level in LOTOS, using the negotiation types of value checking, value passing and value generation. An Estelle or SDL specification can only represent implementations of these negotiation types, which implies that the specification may contain many irrelevant details about the way in which CEI administration is performed.

For example, the flow control by backpressure feature of the OSI Transport Service cannot be represented in Estelle and SDL since the introduction of infinite queues between system parts make it impossible to represent that a system part blocks another at their interface.

**Abstraction levels**

Because of the modelling power of a synchronous interaction model, the FDT LOTOS can be used to represent designs at multiple different and related abstraction levels. This makes LOTOS a broad-spectrum specification language [3].

The FDTs Estelle and SDL are particularly suited for the representation of designs at the lower abstraction levels, such as designs in which an explicit design decision to implement the communication between system parts by means of queues has already been taken.

**Specification styles**

LOTOS makes it possible to express the observable behaviour of some system part in terms of a set of constraints. The synchronous interaction concept allows to express the conditions of an interaction in terms of the conjunction of multiple constraints, characterizing the *constraint-oriented* specification style. Other specification styles supported by LOTOS are the *monolithic*, *resource-oriented* and *state-oriented* specification styles ([12]).

Estelle and SDL specifications can only be written in resource-oriented and state-oriented specification styles. Queues between system parts can be considered as general purpose resources.

**Design approaches**

The choice between an asynchronous or synchronous interaction model is related to the choice of a design approach. Section 2.1 discussed two dual approaches for the design of distributed systems: the design of system parts and the design of interaction systems.

An asynchronous interaction model suits the design of loosely coupled system parts, which communicate by exchanging messages. Therefore, the FDTs Estelle, SDL and LOTOS support the design of system parts. The FDT LOTOS requires, however, the explicit modelling of the communication medium between system parts.

A synchronous interaction model suits the abstract design of strongly coupled system parts. Interaction systems may define complex interaction structures and interaction conditions between system parts. Therefore, the FDT LOTOS supports the design of interaction systems in a more general manner than the FDTs Estelle and SDL. The specification of synchronization between system parts can only be achieved in Estelle and SDL by explicitly defining the mechanism that implements this synchronization, forcing the designer to lower the abstraction level of the specification.

# 3 New developments

The concepts of interaction and temporal ordering allow one to design interaction systems in terms of the observable behaviour of the system parts that contribute to the provision of the interaction system service. The interaction concept does not allow one to define the common behaviour of these system parts as it is discussed in the sequel. Furthermore, the concept of temporal ordering does not allow the definition of some important dependency relationships, such as true concurrency, between interactions.

These limitations have motivated the introduction of the following two basic design concepts: *action* and *causality relation*. Application of these concepts has consequences for the design methodology: more general behaviour structuring techniques are possible, behaviour and entity domains can be identified, and generic design milestones can be defined.

## 3.1 The action concept

An action is defined as:

> *a unit of activity that is performed by a system part, which takes place for the purpose of establishing information.*

The action concept models the relevant characteristics of some activity in the real world. An action is the most abstract model of an activity, and cannot be split at the abstraction level at which the action is defined. Similarly to an interaction, an action should be implemented reliably; i.e. either it happens and reference can be made to all information that is established, or it does not happen and no reference can be made to any information that might have been established.

A more detailed model of an activity can be obtained by decomposing this activity into multiple sub-activities and their relationships. The relevant characteristics of these sub-activities can be modelled by distinct actions. Therefore, the action *concept* is independent of the abstraction level or granularity at which specific activities are modelled.

Figure 6 depicts an example of an activity that is modelled at two different abstraction levels. At the highest abstraction level the outcome of the activity is modelled by a single action. A more detailed model defines how this outcome is achieved by a composition of four related sub-activities, which are modelled by four distinct actions. Intuitively these representations can be considered consistent if the outcome of sub-activity *a* conforms to the outcome of its corresponding action *a* and to the outcome of the most abstract action *a'*.

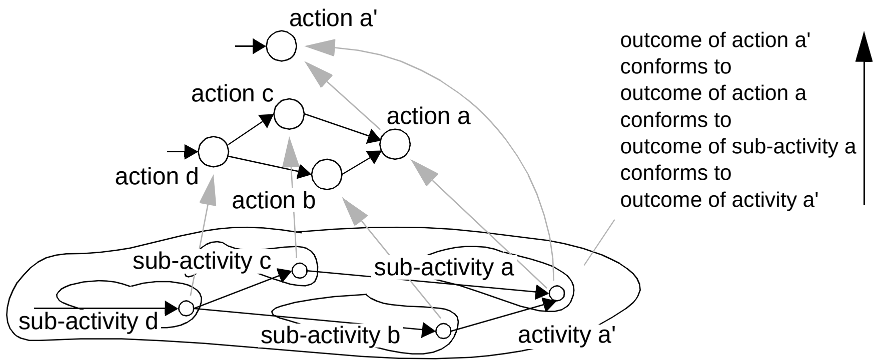



*Figure 6: Example of activity modelling*

We consider each instance of activity to be unique and we assume that we can unambiguously refer to an action by using an action identifier. Other relevant characteristics of an activity are defined by the following attributes:

- *location*: the logical or physical location where an action occurs;
- *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 reference;
- *probability*: the probability that an action occurs once all conditions for its occurrence are satisfied.

Some examples of actions with their attributes are presented below.

a ($t_a$:Time, $v_a$:Nat)
b ($l_b$:Location, $t_b$:Time, $v_b$:Value, $p_b$:Probability)
c ($l_c$:Location, $v_c$:Value, $r_c$:RetainedValue)

### Integrated interaction

An action may represent an *integrated interaction*, which abstracts from the individual interaction contributions or responsibilities of the involved system parts. This implies that an interaction can be considered as a possible implementation of an action. However, some actions are not abstractions of interactions, since some actions may not be distributed over multiple system parts at lower abstraction levels.

Following this more general interpretation of an interaction than the one given in section 2.2, interaction contributions have the same attributes as actions, but they differ in the definition of the constraints on the establishment of the attribute values. Figure 7 depicts an example of an action *a* that is implemented as an interaction *a̲* shared by three behaviours. In order to distinguish interactions from actions, interaction identifiers are underlined.

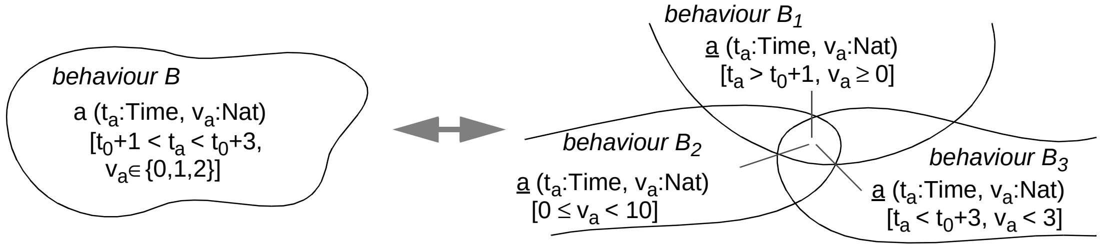



*Figure 7: Example of an integrated interaction*

## 3.2 Behaviour definitions

The occurrence of an action generally depends on the occurrence or non-occurrence of other actions. Causality relations are introduced to represent these dependencies. The causality relation of an action $a_1$ defines the enabling condition of $a_1$ in terms of actions that must have occurred, its enabling actions, actions that must not have occurred, its disabling actions, and conditions on the attributes values of the enabling actions and constraints on the attribute values of action $a_1$. Action $a_1$ is called the *result* action.

Two basic causality relations are distinguished:

- the enabling relation $a_2 \rightarrow a_1$, which defines that the occurrence of action $a_2$ is a condition for the occurrence of action $a_1$. Only in case $a_2$ has occurred $a_1$ is allowed to occur; $a_1$ can refer to the attribute values of $a_2$.
- the disabling relation $\neg a_2 \rightarrow a_1$, which defines that the non-occurrence of action $a_2$ is a condition for the occurrence of action $a_1$. As long as $a_2$ does not occur before or simultaneously with $a_1$, $a_1$ is allowed to occur; if $a_2$ occurs and $a_1$ has not occurred before it, $a_2$ disables or excludes the occurrence of $a_1$.

More complex causality relations can be composed from these two basic relations using the logical operators *and* ($\wedge$) and *or* ($\vee$), which are interpreted according to the rules of boolean logic, once occurrences and non-occurrences of actions are considered as atomic propositions.[1] Some examples of causality relations with two enabling or disabling actions are given below:

- $a_2 \wedge a_3 \rightarrow a_1$: both $a_2$ and $a_3$ must have happened before $a_1$ is allowed to happen. Since both $a_2$ and $a_3$ enable $a_1$, action $a_1$ can refer to the attribute values of both actions;
- $a_2 \vee a_3 \rightarrow a_1$: either $a_2$ or $a_3$ or both must have happened before $a_1$ is allowed to happen. Action $a_1$ is enabled by either $a_2$ or $a_3$, but not both, even in the case that both $a_2$ and $a_3$ have happened before $a_1$. This implies that in an implementation a mechanism that establishes what action actually enables $a_1$ is necessary. This mechanism chooses between $a_2$ and $a_3$ in case both actions have happened before $a_1$. Action $a_1$ can only refer to attribute values of the action that enabled it;
- $a_2 \wedge \neg a_3 \rightarrow a_1$: the occurrence of $a_2$ and the non-occurrence of $a_3$ are both conditions for the occurrence of $a_1$;

1. We stress that $\neg a$ is not equivalent to the negation of $a$ in the boolean sense. Suppose that actions $a$ and $b$ both occur. In this case $a \rightarrow b$ implies that $t_a < t_b$, where $t_a$ and $t_b$ represent the time of occurrence of $a$ and $b$, respectively. Intuitively, the negation of this relation would represent the case in which $t_a \geq t_b$. However, $\neg a \rightarrow b$ implies that $t_a > t_b$. This specific choice of implicit time condition has been motivated by the need to define behaviour patterns such as disabling and choice.

- $a_2 \vee \neg a_3 \rightarrow a_1$: the occurrence of $a_2$ or the non-occurrence of $a_3$ or both are conditions for the occurrence of $a_1$.

Figure 8 depicts the graphical representations of the elementary causality relations discussed above. Actions and causality relations are represented as circles and arrows between circles, respectively. An interaction is represented as a segmented circle, where each segment represents a contribution to the interaction. An example of such a representation can be found in Figure 11.

The logical operators *and* ($\wedge$) and *or* ($\vee$) are represented explicitly, which allows a direct representation of a parentheses structure that may be used in the textual notation to indicate the priority of the logical operators. Furthermore, this notation does not prescribe the use of a conjunctive or disjunctive canonical form. For example, the reported notation in [13] required that a causality condition is written in the form of a disjunction of conjunctions of action occurrences and non-occurrences.

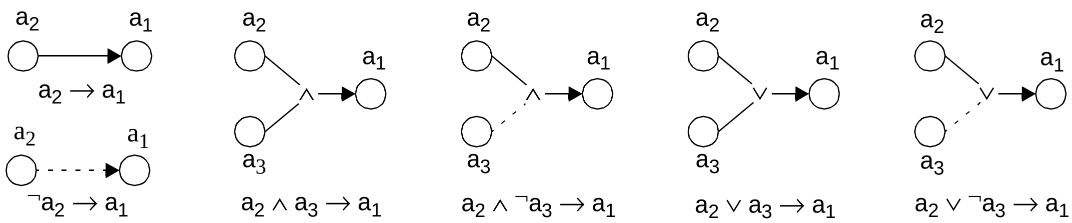



*Figure 8: Elementary causality relations*

A behaviour can be defined by the causality relations of all its actions. Some actions are enabled from the beginning of the behaviour (initial actions), and have a condition *start* which corresponds to true. Figure 9 depicts the graphical representation of some well-known behaviour patterns, and their shorthand notation. The context of these behaviour patterns in terms of possible other conditions for $a_1$ and $a_2$ are not explicitly represented in Figure 9.

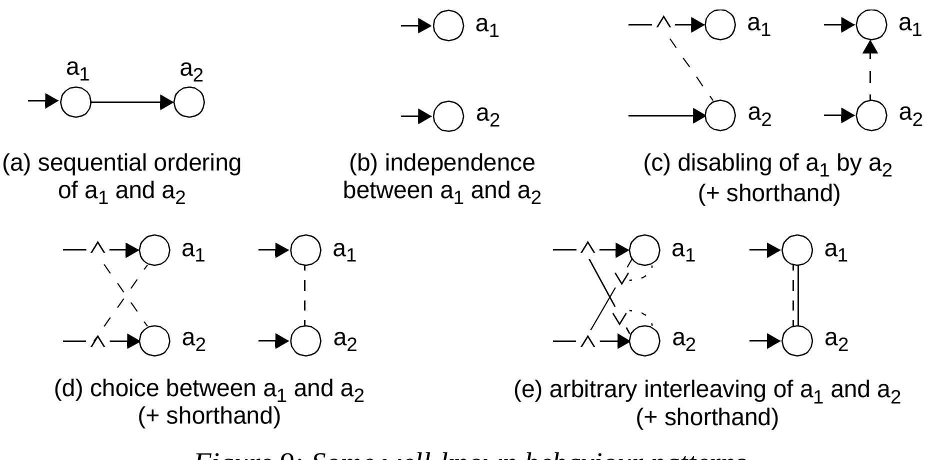



*Figure 9: Some well-known behaviour patterns*

The textual behaviour representation of arbitrary interleaving is given below. The causality relations of actions $a_1$ and $a_2$ represent that either $a_2$ is enabled by $a_1$ or $a_1$ is enabled by $a_2$.

$$
\begin{aligned}
\mathrm{B} := \{\ & a_2 \vee \neg a_2 \rightarrow a_1, \\
& a_1 \vee \neg a_1 \rightarrow a_2\ \}
\end{aligned}
$$

## 3.3 Behaviour structuring

Generally, the purpose of structuring behaviours is twofold: (i) to improve the comprehensibility of the design, or (ii) to be used as a prescription for implementation. The definition of behaviours in the preceding section is limited to the monolithic definition of finite behaviours. Structuring techniques also allow the definition of repetitive behaviours.

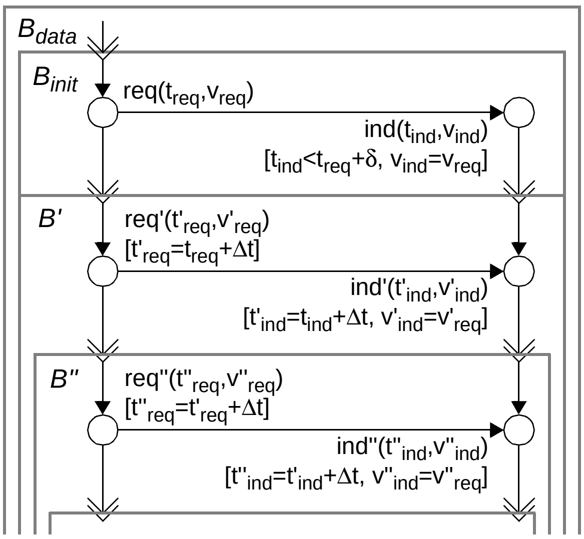
**Causality-oriented behaviour structuring**

The definition of causality relations between actions can be generalized to the definition of causality relations between behaviours. This allows the structuring of a complex behaviour in terms of less complex sub-behaviours and their relationships. Furthermore, predefined sub-behaviours can be reused through instantiation, and repetitive behaviours can be represented through recursive behaviour instantiation. This structuring technique, which is called *causality-oriented structuring*, makes use of:

- *entry points*: which are points in a behaviour from which actions of that behaviour can be enabled by conditions involving actions of other behaviours,
- *exit points*: which are causality conditions in a behaviour that can be used to enable actions of other behaviours.

Behaviours can be composed by relating their exit and entry points, which are indicated by the keywords *exit* and *entry*, respectively. A behaviour may have multiple exit and entry points.

Figure 10 depicts the causality-oriented composition of a unidirectional isochronous datagram service. Actions *req* and *ind* represent a data request and a data indication service primitive, respectively. Sub-behaviour $B_{init}$ models the first instance of communication, in which a transit delay smaller than a certain value δ between the occurrence of a *req* and its corresponding *ind* is established. Sub-behaviour *B'* models subsequent instances of communication such that the interval between two consecutive *req*s (*ind*s) is equal to a certain constant value Δt. This requirement implies that the transit delay is kept constant as well. Repeated instantiation of *B'* is represented by adding an apostrophe to the behaviour and action identifiers.




$B_{data} := \{\ start \rightarrow B_{init}(entry)\ \}$

$B_{init} := \{\ entry \rightarrow req(t_{req},v_{req}),$
$\quad req(t_{req},v_{req}) \rightarrow$
$\quad\quad ind(t_{ind},v_{ind})[t_{ind}<t_{req}+\delta,\ v_{ind}=v_{req}],$
$\quad req(t_{req},v_{req}) \rightarrow B'(entry_1(t_{req})),$
$\quad ind(t_{ind},v_{ind}) \rightarrow B'(entry_2(t_{ind}))\ \}$

$B' := \{\ entry_1(t_{req}) \rightarrow$
$\quad\quad req'(t'_{req},v'_{req})[t'_{req}=t_{req}+\Delta t].$
$\quad entry_2(t_{ind}) \wedge req'(t'_{req},v'_{req}) \rightarrow$
$\quad\quad ind'(t'_{ind},v'_{ind})[t'_{ind}=t_{ind}+\Delta t],$
$\quad req'(t'_{req},v'_{req}) \rightarrow B''(entry_1(t'_{req})),$
$\quad ind'(t'_{ind},v'_{ind}) \rightarrow B''(entry_2(t'_{ind}))\ \}$

*Figure 10: Causality-oriented composition of an isochronous datagram service*

## Constraint-oriented behaviour structuring

The constraint-oriented structuring technique allows the structuring and composition of a behaviour in terms of a conjunction of conditions and constraints on actions, which are defined in separate sub-behaviours. This implies that some actions are represented (or actually implemented) as interactions, which consist of multiple interaction contributions distributed over different sub-behaviours.

Alternative constraint-oriented compositions of the same monolithic behaviour definition are possible:

- an action can be assigned to a single sub-behaviour, or can be shared by more than one sub-behaviour;
- in case an action is shared by sub-behaviours, its conditions and constraints, which are represented by a single causality relation in the monolithic behaviour definition, can be distributed over multiple causality relations of the interaction contributions in different ways. The causality relation of an interaction contribution defines that part of the original conditions and constraint that are assigned to a sub-behaviour.

The choice between alternative decompositions is determined by the design objectives of a design step and by technical and quality criteria. A conformance requirement is that the conjunction of the conditions and constraints of the interaction contributions should be logically equivalent to the conditions and constraints of the (integrated) action.

The causality-oriented definition of the isochronous datagram service of Figure 10 abstracts from the assignment of responsibilities on the occurrences of *req*s or *ind*s to the service users and the service provider. Figure 11 depicts a constraint-oriented decomposition of this service in which responsibilities are assigned to sub-behaviours.

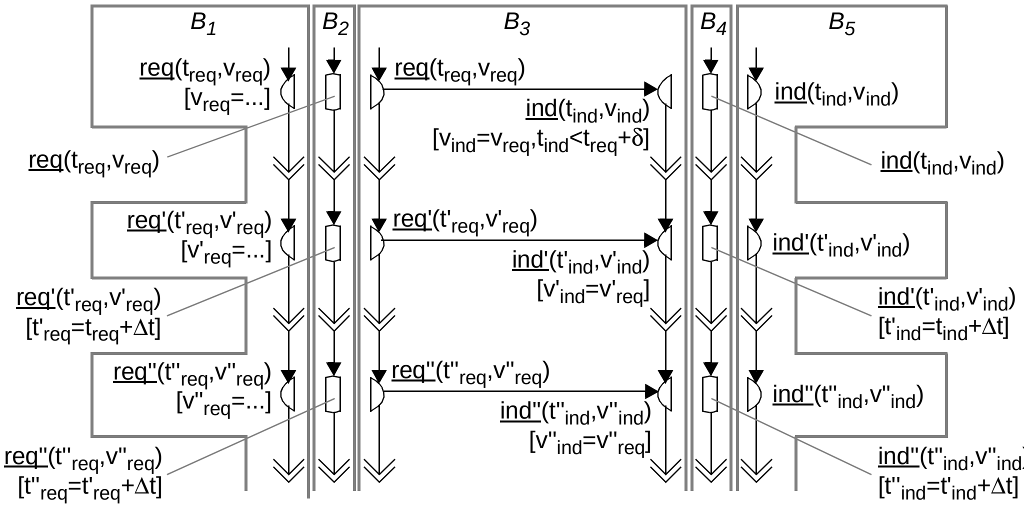



*Figure 11: Constraint-oriented composition of isochronous datagram service*

Sub-behaviour $B_3$ is responsible for the sequential and correct transfer of data unit values across the service provider. Sub-behaviours $B_1$ and $B_5$ are responsible for providing and accepting the data unit values, respectively. Sub-behaviours $B_2$ and $B_4$ must maintain the timing constraints on consecutive occurrences of *req*s and *ind*s, respectively.

Typically, sub-behaviours $B_1$ and $B_5$ define part of the behaviour of the service users and sub-behaviour $B_3$ defines part of the behaviour of the service provider. Sub-behaviours $B_2$ and $B_4$ can be assigned to the service users or to the service provider, or to both.

## 3.4 Behaviour and entity domains

The structure of a behaviour, in terms of a composition of sub-behaviours, is a prescription for implementation if these sub-behaviours are assigned to logical or physical system parts. The term physical system part is used to denote some component that can be identified in the real system. The term logical system part is used to denote some composition of logical or physical system parts, which is considered from an integrated perspective. This means that a logical system part represents an abstraction of many possible compositions of logical or physical system parts. Consequently, in order to obtain a final implementation of a system each of the system's logical parts should be decomposed until a mapping onto real system components is completely defined.

Because the term system part is often associated with real system components, we will use the term entity, or *functional entity*, further on to denote a logical or physical system part.

The concept of functional entity is related to the concepts of *action point* and *interaction point*, which are used to denote the logical or physical locations at which actions and interactions occur, respectively. The following rules apply to functional entities, interaction points and action points:

1. each functional entity is delimited by zero or more interaction points, and each interaction point is shared by two or more functional entities;
2. each functional entity contains zero or more action points, and each action point is contained by a single functional entity;
3. each functional entity is delimited by at least one interaction point or contains at least one action point.

Considering all the design concepts introduced so far, two distinct but related domains for system design can be distinguished:

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

Figure 12 depicts both domains, and their related basic design concepts.

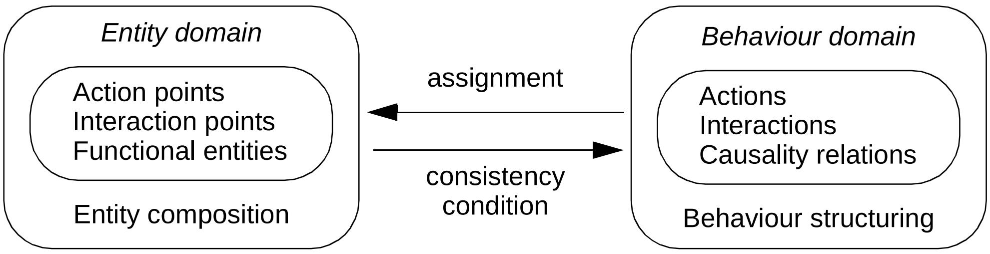



*Figure 12: Entity and behaviour domain*

The entity and behaviour domains are related to each other by an assignment and a consistency condition. A behaviour is assigned to each functional entity, which implies that actions and interactions of a behaviour are assigned to action points and interaction points of a functional entity, respectively. Given a certain assignment of behaviours to functional entities, the consistency condition imposes that:

1. actions of a behaviour happen at action points of the functional entity to which the behaviour is assigned;

2. interactions of a behaviour happen at interaction points which are shared by the functional entities to which the interaction contributions are assigned. Interactions between functional entities can only occur at the interaction points they share;
3. the result action (or interaction) and its conditions and constraints defined in a causality relation are assigned to a single functional entity. This means that a single functional entity is responsible for the conditions and constraints on an action (or interaction) that are represented by a causality relation.

Figure 13 illustrates a possible assignment of the sub-behaviours identified in Figure 11 to functional entities. The depicted interaction points represent the service access points at which the *req* and *ind* service primitives take place.

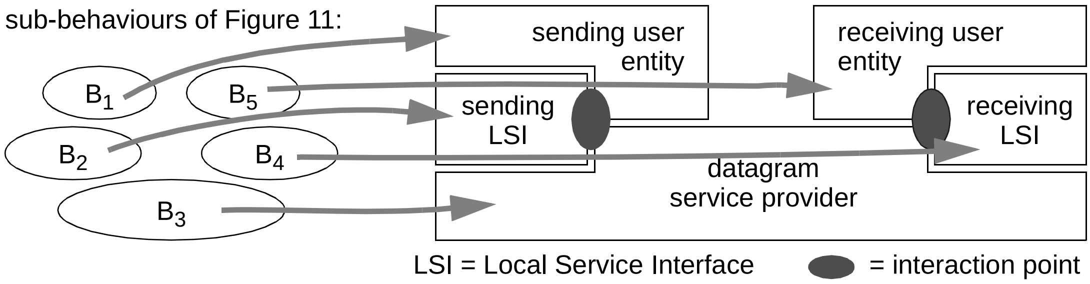



*Figure 13: Assignment of behaviours to entities*

The distinction between an entity and a behaviour domain allows a clear separation of design concerns for the identification and definition of design steps. A design step can be related to the entity domain or to the behaviour domain, if its objectives are defined in terms of manipulations of concepts in the entity domain or in the behaviour domain, respectively (see Figure 12). Examples of designs steps that are related to the entity domain are (functional) entity decomposition and interaction point refinement ([7]). Examples of design steps that are related to the behaviour domain are resolution of non-determinism, behaviour reduction ([7]) and action refinement ([6]).

According to the consistency condition, the entity structure, in terms of functional entities and their action and interaction points, and the behaviour structure, in terms of actions, interactions and their causality relations, are closely related. Since the entity structure of a design is a prescription for implementation, manipulations of this structure should be reflected by corresponding manipulations of the behaviour structure. For example, the decomposition of a functional entity into a composition of sub-entities implies a corresponding constraint-oriented decomposition of the original behaviour into sub-behaviours that can be assigned to these sub-entities. However, this behaviour decomposition is restricted by decomposition rules, which are presented in [2], [9] and [13].

Manipulations in the behaviour domain do not necessarily have consequences for the entity domain. For example, the behaviour of an entity may be defined in more detail without implying any modifications to the entity structure. Furthermore, the structure of a behaviour only becomes a prescription for implementation when the (sub-)behaviours are assigned to functional entities.

Before such an assignment is made, a designer may choose different behaviour structures in order to conceive and understand the characteristics of the behaviour. For example, behaviours may be structured as monolithic, causality-oriented, constraint-oriented, or mixed causality-constraint-oriented structures. In order to prepare the assignment to functional entities, a constraint-oriented, or mixed-causality-constraint-oriented behaviour structure is needed.

## Specification styles in LOTOS

Specification styles have been introduced in [12] as a way to structure specifications according to specific design objectives. When applying the entity and behaviour domains for defining design objectives and characterizing design steps, we can conclude in retrospection that, for example, the resource-oriented specification style is necessary to represent in LOTOS the desired entity structure and the mappings from a behaviour specification onto the functional entities of this structure, at a certain abstraction level.

## Role of the domains in the design process

From the perspective of the entity domain, a pure top-down design process consists of the repeated decomposition of functional entities into compositions of sub-entities, until a mapping onto real system components is achieved. Since functional entities are delimited by interaction points, we have to insert interaction points in a functional entity in order to define a composition of sub-entities from this single functional entity, allowing these sub-entities to be delimited. Therefore, insertion of interaction points is a necessary manipulation to achieve entity decomposition.

When entity decomposition is performed, the behaviour of the original functional entity has to be decomposed into sub-behaviours, such that these sub-behaviours are assigned to the resulting sub-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 behaviour of the resulting functional entities, but have to be indirectly related by interactions occurring at interactions points shared by these functional entities.

Figure 14 illustrates the decomposition of the isochronous service provider of Figure 13 into two protocol entities and a lower level service provider. The decomposition of the behaviour of the isochronous service provider is performed in two steps:

1. two actions are introduced representing the occurrences of a data request and a data indication primitive of the lower level service. This more detailed representation of the service provider is called a protocol (see also section 3.5).
2. both actions, and their corresponding action points, have to be decomposed into interactions and interaction points in order to assign them to the protocol entities and the lower level service provider.

The first step is an example of behaviour refinement, which is discussed in section 4.

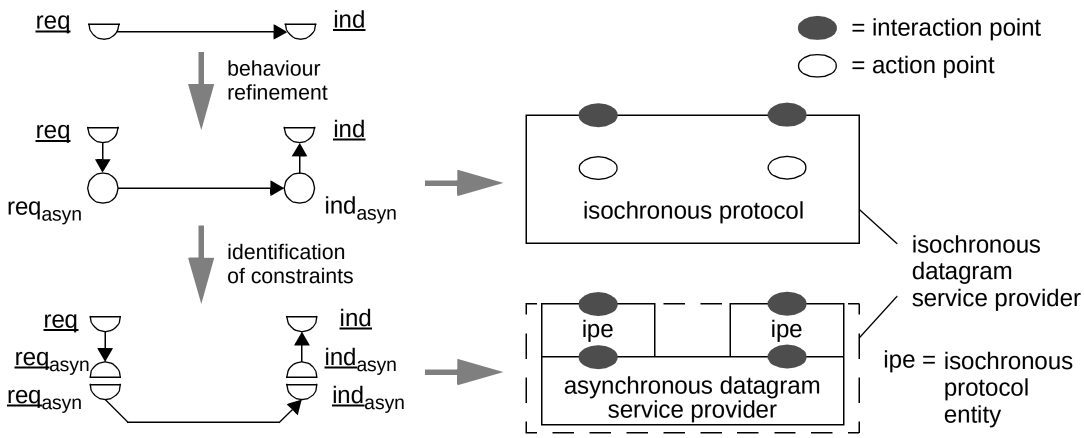



*Figure 14: Decomposition of isochronous datagram service provider*

## 3.5 Design milestones

This section presents some generic design milestones that are relevant for distributed system design, by identifying their objectives and their relative position in a design process. These design milestones have been enabled by the introduction of the action concept, making in this way the design approach presented in section 2.1 more general.

For convenience these design milestones are represented primarily in terms of the entity domain.

### Identification of system and environment

*Objective*: identification of the distributed system and the application environment, in terms of the application entities that use the system and the way these entities cooperate. This design milestone is used to determine the activities of the application environment that should be supported by the distributed system, and the degree of support to be provided.

The requirements on application support to be provided by the distributed system, determines a boundary between the system and its environment. Figure 15 depicts this boundary.

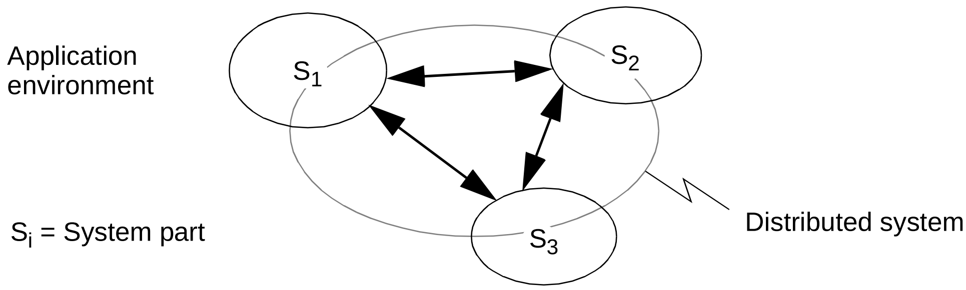



*Figure 15: Identification of system and environment*

### Service definition

*Objective*: definition of the shared boundary between the system and its environment. This design milestone defines the common behaviour of the system and its environment, which is called the *service*, and abstracts from the many different ways in which the responsibilities and constraints for providing the service may be distributed between the system and the environment. A service is defined in terms of (common) actions and their causality relations.

Because the individual contributions of the system and the environment to the service are not defined, both entities are not distinguished at this abstraction level. The service is therefore assigned to a single functional entity, which is called the *interaction system between the system and its environment*. This interaction system only comprises that part of the environment that is relevant for the definition of the service.

Figure 16 (a) depicts this design milestone.

### Definition of service provider and service users

*Objective*: definition of the behaviour of the system, which is also called the *service provider*, as it is observed by its environment. At this abstraction level responsibilities and constraints for performing the service are assigned to the service provider and to its environment, using the constraint-oriented structuring technique. In this way the observable behaviour of the service provider is defined as well as part of the observable behaviour of the environment, which consists of the service users or application entities.

This design milestone is useful to delimit the functionality of the service provider. The internal structure of the service provider is not considered at this abstraction level.

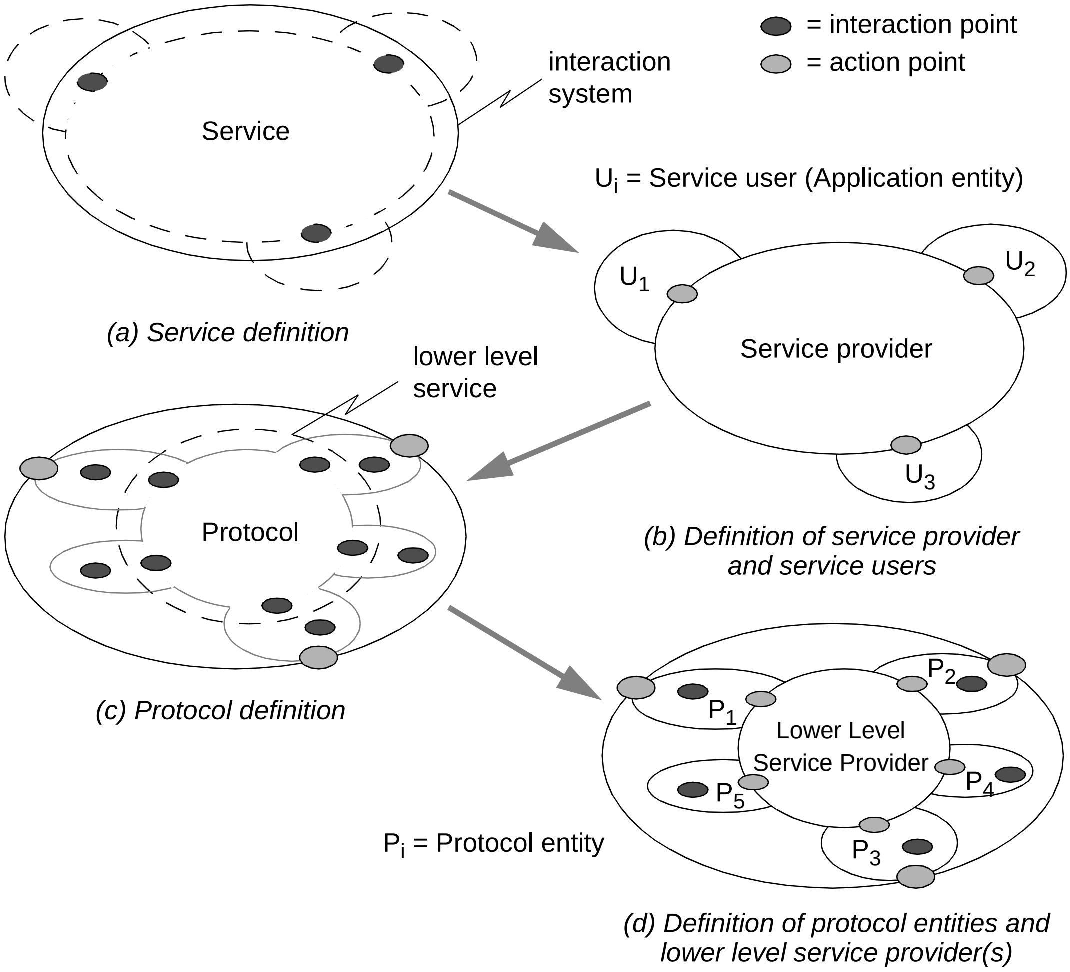



*Figure 16: Some generic design milestones*

Figure 16 (b) depicts this design milestone.

**Protocol definition**

*Objective*: definition of how the observable behaviour of the service provider is offered, while abstracting from possible decompositions of the service provider. Therefore, the internal behaviour of the service provider is defined in terms of a monolithic or causality-oriented behaviour structure, which is called the *protocol*.

The definition of the internal structure of the service provider, in terms of the logical distribution of actions and associated action points, should anticipate on the design objectives of the next design milestone. This implies that the designer should already have some decompositions in mind.

Figure 16 (c) depicts this design milestone.

**Definition of protocol entities and lower level service provider(s)**

*Objective*: definition of the internal structure of the service provider in terms of a composition of distributed protocol entities which are interconnected by one or more lower level service providers. At this abstraction level responsibilities and constraints for performing the protocol are assigned to protocol entities and lower level service provider(s), using the constraint-oriented structuring technique.

The common behaviour of the protocol entities and the lower level service provider(s) is defined by the protocol. This implies that a protocol definition provides the functional requirements for the definition of the lower level services and the definition of how they are used to provide the observable behaviour of the service provider.

Figure 16 (d) depicts this design milestone.

**Interface decomposition**

The presentation of the milestones of Figure 17 (b) and (d) may suggest that the responsibilities for performing the actions at the interfaces between the service provider and service users, or between the lower level service provider and protocol entities, respectively, have to be assigned to the involved entities. However, in some cases it is better to defer the assignment of (part of) the responsibilities to later design steps; for example to prevent that an early assignment has to be reconsidered.

An example of this is illustrated in Figure 13, where part of the local constraints on the occurrences of actions *req* and *ind* have been assigned to the entities sending LSI and receiving LSI, respectively. The assignment of local service constraints to a separate entity is particularly useful in the standardization of distributed systems, where the distribution of these constraints over the service users and the service provider can be left to the implementer.

## 4 Behaviour refinement

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 more 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 each abstract reference action has one or more corresponding *concrete reference actions* in the concrete behaviour. By assuming that, it is 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 behaviour for assessing conformance.

Two different types of the behaviour refinement design operation are considered:

- *causality refinement*, in which the causality relations between the abstract reference actions are replaced by causality relations involving their corresponding concrete reference actions and some inserted actions;
- *action refinement*, in which an abstract reference action is replaced by an activity involving multiple concrete reference actions and their causality relations.

There are refinements that cannot be strictly characterized as causality refinement or action refinement. In the examples we have studied so far, these refinements can be considered as a combination of causality and action refinement.

Figure 17 illustrates the causality refinement and action refinement design operations. The abstract behaviour consists of the abstract reference actions $a'$, $b'$, $c'$ and $d'$. The concrete behaviour obtained by causality refinement consists of the concrete reference actions $a$, $b$, $c$ and $d$, and the *inserted actions* $e$, $f$ and $g$, which characterize the modifications performed in this design operation. The concrete behaviour obtained by action refinement consists of the concrete reference actions *a*, *b*, $c_1$, $c_2$, $c_3$ and *d*, where actions $c_1$, $c_2$ and $c_3$ form an activity that refines the abstract reference action *c'*. In this example, $c_2$ and $c_3$ are both concrete reference actions which correspond to abstract reference action *c'*.

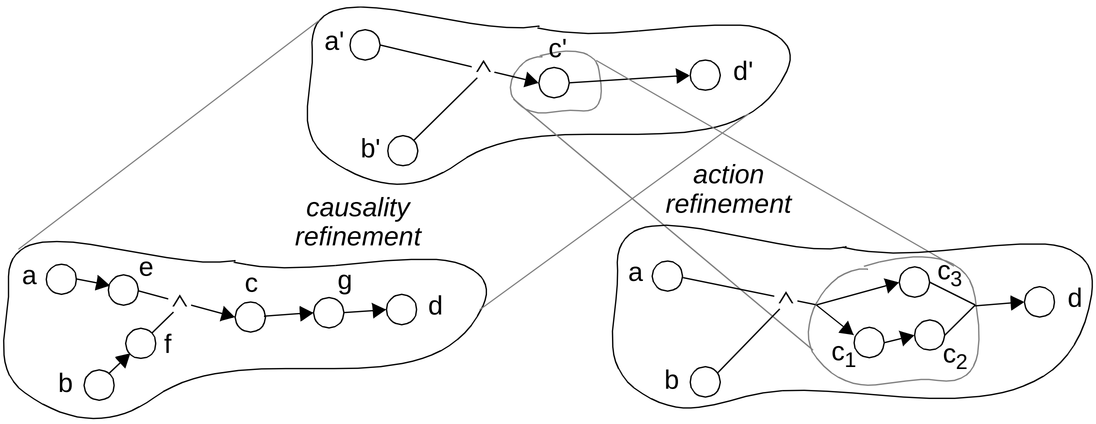



*Figure 17: Causality and action refinement*

Since an abstract behaviour can be replaced by many different alternative concrete behaviours and the choice of specific concrete behaviours is determined by specific design objectives, the behaviour refinement 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.

## 4.1 Causality refinement

The following activities have to be performed in an instance of causality refinement:

1. delimitation of the abstract behaviour;
2. elaboration of the concrete behaviour;
3. determination of the abstraction of the concrete behaviour.

**Delimitation**

The abstract behaviours that are considered 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. Delimitation is important to make it feasible to refine infinite or large behaviours.

**Conformance**

Causality refinement generally consists of replacing direct references to attribute values of abstract reference actions by indirect references to attribute values of concrete reference actions via attribute values of inserted actions. An instance of causality 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 enabling and disabling relations*: enabling and disabling relationships between abstract reference actions defined in the abstract behaviour are preserved in the concrete behaviour by their corresponding concrete reference actions. For example, the

causality relation $a' \wedge b' \rightarrow c'$ in Figure 17 is preserved in the concrete behaviour by the combination of the causality relations $a \rightarrow e$, $b \rightarrow f$ and $e \wedge f \rightarrow c$;

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. Two alternatives for the preservation of attribute values may be considered:
   - *strong preservation*: all attribute values that are possible for an abstract reference action are also possible for its corresponding concrete reference action;
   - *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.

   For simplicity, we only consider strong preservation in this paper.

**Abstraction rules**

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 [2]. These rules are called *abstraction rules*.

The following two abstraction rules are examples of general rules that have been 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 enabling relations:

*Abstraction Rule 1*:
an inserted action that is an enabling condition for an action of the concrete behaviour can be replaced by the condition of the inserted action as defined in its causality relation.

*Abstraction Rule 1b*:
constraints on time attribute values of inserted actions being removed have to be considered in the computation of the time constraints on the remaining actions. This computation has to consider implicit time constraints of enabling relations.

Figure 18 illustrates the application of these abstraction rules. If we abstract from action $c$, the time constraint on action $d$ becomes $t_d < t_a+\delta_1+\delta_2$ if we only substitute the reference to $t_c$ by $t_a+\delta_1$ as defined in the time constraint $t_c=t_a+\delta_1$ on action $c$. In addition, we must consider the implicit time constraint $t_c < t_d$ of the enabling relation $c \rightarrow d$, which renders the time constraint $t_a+\delta_1 < t_d < t_a+\delta_1+\delta_2$ on action $d$. This constraint is considered in the computation of the resulting time constraint of action $b'$ when abstracting from $d$.

## 4.2 Action refinement

Action refinement consists of replacing an abstract reference action by an activity. An activity is a composition of actions, hence it is more concrete than its corresponding abstract

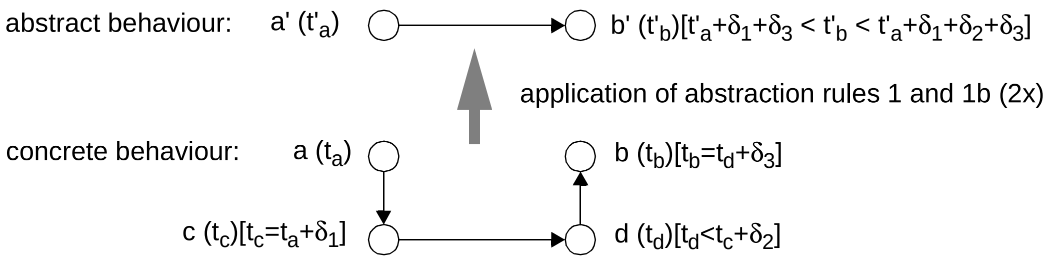



*Figure 18: Example of causality abstraction*

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.

In general the essence of action refinement is the decomposition of at least one of the action attributes of the abstract reference action in multiple action attributes of the concrete reference actions. Not only action values, but also location, time and probability of an abstract reference action may be distributed over actions of an activity. This is the essential difference between action refinement and causality refinement, since in the latter the attribute values of an abstract reference action and its corresponding concrete reference action must be the same.

### Correctness

Two correctness requirements are identified to determine if the activity that replaces an abstract reference action is a correct implementation of that action in its context:

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 rules for 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 characterize *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 characterize *context abstraction*.

Figure 19 depicts the relationships between attribute abstraction, context abstraction and the design choice to be taken in action refinement.

### Attribute abstraction

An action is a proper abstraction of an activity if it has attribute values that represent the attributes of the activity, namely the location, value, time and probability attributes. 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 [2] and [6].

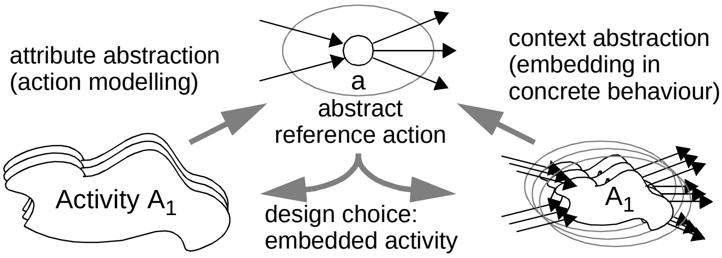



*Figure 19: Elements of action refinement*

Activities may make their value attributes available through the occurrence of one or more final actions. The following generic cases are distinguished:

- *single final action*: an activity has a single final action, such that this activity makes all its values available when this final action occurs;
- *conjunction of final actions*: an activity has multiple independent final actions, such that this activity makes all its values available when all these final actions occur;
- *disjunction of final actions*: an activity has multiple alternative final actions, such that this activity makes all its values available when one of these final actions occurs.

Not all information values of an activity have to be established in its final actions. Some information values may be established in other (non-final) actions, which are referred to by the final actions and are made available in their retained value attributes.

### Example: parallel interface

Figure 20 depicts a parallel interface activity and its corresponding abstract reference action *word*, which is an example of abstraction of a conjunction of final actions. The value attribute of the abstract reference action *word* consists of four different bytes that are established by four independent concrete reference actions $byte_i$. The moment of time at which action *word* occurs is equal to the moment of time at which the last byte becomes available. The location of the abstract reference action *word* contains the locations of the concrete reference actions $byte_i$, in a similar way as a certain country contains cities. This allows one to refer to the location of *word* as a single location that is actually implemented as a composition of (sub-)locations.

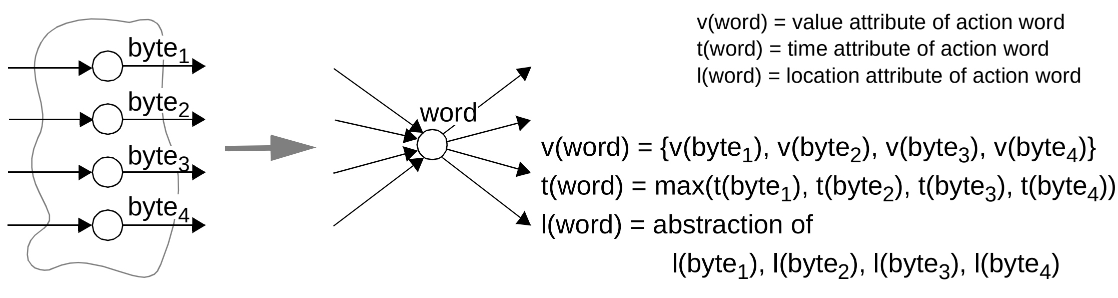



*Figure 20: Example of parallel interface*

### Context abstraction

Since an abstract reference action may be implemented by different alternative activities, we 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. A method for deducing the abstract behaviour of a concrete behaviour which is obtained through action refinement is given in [2].

This method starts with the definition of an abstract reference action for each activity and for each action that is not refined. The definition of an abstract reference action for an activity represents an abstraction of the final actions of this activity, and can be defined by considering the rules for attribute abstraction. Once the abstract reference actions have been defined, one should have rules to abstract from the remaining, i.e. the non-final actions of the activities, which are similar to the abstraction rules of causality refinement. Applying these rules one should obtain a behaviour which consists exclusively of abstract reference actions. This behaviour is the abstraction of the concrete behaviour.

## 5 Example: financial transaction

The design methodology presented in this paper is illustrated with the design of a system which supports a specific type of financial transaction.

### 5.1 Identification of money transfer system

Figure 21 depicts the application domain of a money transfer system. Suppose that a client, who may represent a business organisation or an ordinary household, wants to order a bank to carry out a specific transaction. Suppose this transaction consists of transferring money from two different accounts to a third account, and that these accounts are administered by three different banks.

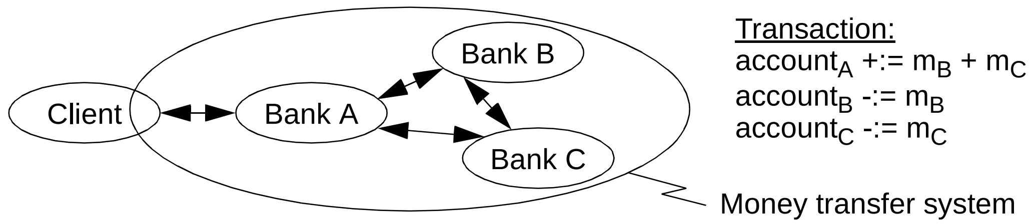



*Figure 21: Application domain*

### 5.2 Money transfer service

The common behaviour of the client and the money transfer system can be modelled by a single action at a high abstraction level. This action, called *Trans* in Figure 22 (a), defines the outcome of the financial transaction.

In reality, the transaction must be initiated by the client. This is modelled at a lower abstraction level by a separate action, called $Trans_{req}$ in Figure 22 (b), which establishes the information that is needed to perform the transaction, i.e. the accounts ($s_A$, $s_B$ and $s_C$) and the amounts of money to be transferred ($m_B$ and $m_C$).

The relation between the more concrete behaviour of Figure 22 (b) and the abstract behaviour of Figure 22 (a) is defined by the abstraction rules of the causality refinement design operation. In this case, actions *Trans* and $Trans_{cnf}$ are reference actions and action $Trans_{req}$ is an inserted action.

The more concrete behaviour of Figure 22 (b) allows one to prescribe the time constraint that applies to the transaction, which must be performed within a maximal time interval $\Delta$t.

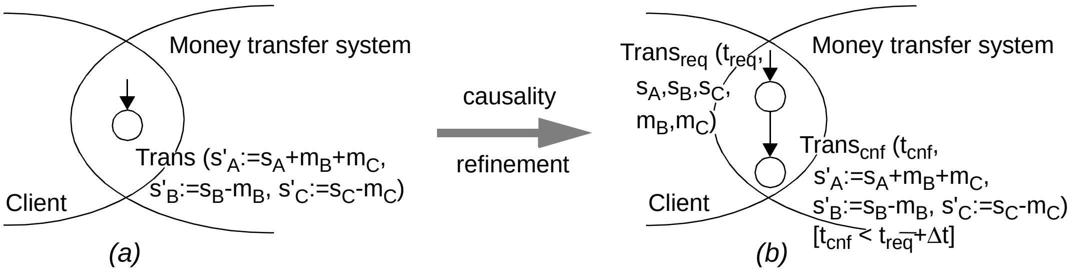



*Figure 22: Money transfer service*

## 5.3 Money transfer service provider

Figure 23 depicts the distribution of the conditions and constraints on the occurrences of actions *Trans*$_{req}$ and *Trans*$_{cnf}$ between the sub-behaviours of the functional entities *Client* and *Money transfer system*. The constraint of the interaction contribution *<u>Trans</u>*$_{req}$ in the *Client* sub-behaviour defines that the client is responsible for providing the necessary account information and the amounts of money to be transferred. After the interaction has happened these values are also known by the money transfer system.

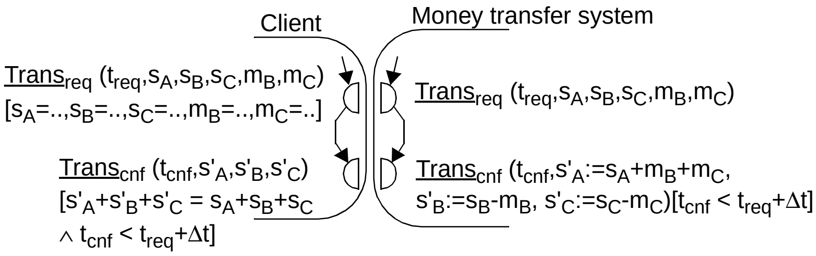



*Figure 23: Money transfer service provider*

The responsibility for performing the financial transaction correctly and in time is assigned to the money transfer system. This responsibility is represented by the operations and constraints on the attributes of the interaction contribution *<u>Trans</u>*$_{cnf}$ in the corresponding sub-behaviour. Following the principle of parsimony, it would be sufficient to assign this responsibility to only the *Money transfer system* entity and inform the *Client* entity about the transaction outcome by means of value passing.

However, in case entities do not trust each other completely, the same responsibility may be assigned to multiple entities. In this case the client has been assigned the responsibility to check the outcome of the transaction. The interaction contribution *<u>Trans</u>*$_{cnf}$ of the *Client* sub-behaviour defines that the transaction must be performed in time and that the sum of the accounts credits before and after the transaction should be the same.

## 5.4 Money transfer protocol

Figure 24 represents the behaviour definition of the money transfer protocol. The protocol behaviour is a refinement of the causality relation between the interaction contributions *<u>Trans</u>*$_{req}$ and *<u>Trans</u>*$_{cnf}$. We assume this behaviour is distributed between three functional entities, which represent the three different banks involved.

The transaction is performed in two stages, which are coordinated by bank A. The purpose of the first stage is to make a reservation on the three accounts in order to avoid interference with other transactions. Action *A-RR* models the reservation of the account at bank A and

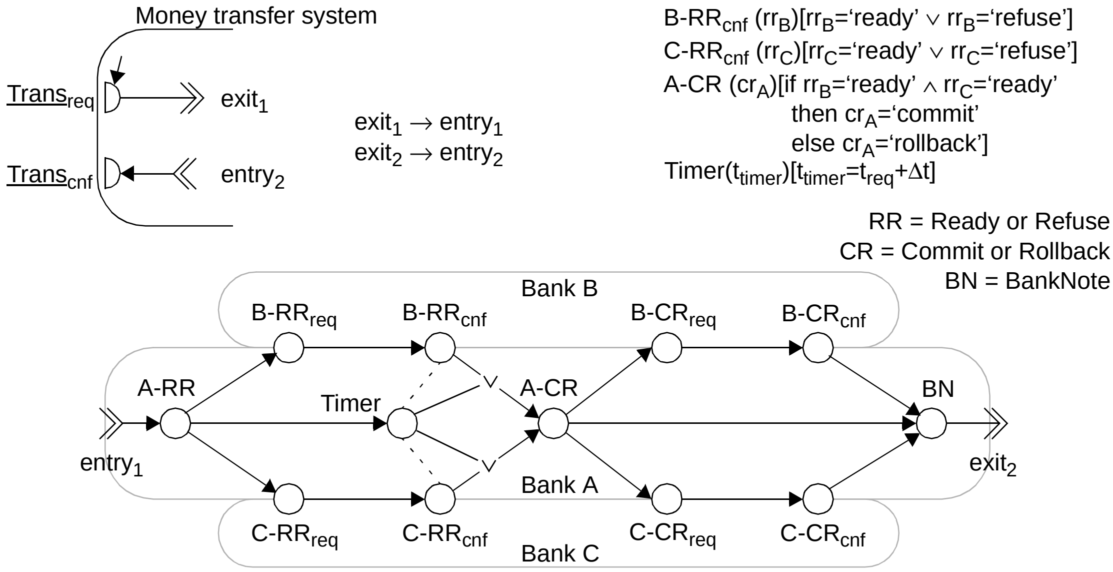



*Figure 24: Money transfer protocol*

enables the reservation of the accounts at banks B and C. The reservations at banks B and C are modelled by a reservation request (action $X\text{-}RR_{req}$) which is followed by a reservation confirm (action $X\text{-}RR_{cnf}$).

An $X\text{-}RR_{cnf}$ contains an answer with two possible values: the value *'ready'*, which represents that the reservation is made, or the value *'refuse'*, which represents that the reservation could not be made (e.g. because of a low balance). Figure 24 depicts the attributes of some actions. Since the attributes of other actions are rather straightforward they are not represented for the sake of brevity.

In this example we assume that the first stage is critical with respect to time, because e.g. resources have to be reserved at banks B and C. The $X\text{-}RR_{cnf}$'s from both banks should occur within a certain time limit, which is modelled by action *Timer*, otherwise the transaction should not happen according to the constraints that were defined in Figure 23.

The purpose of the second stage is to decide and guarantee that the transaction either happens completely or not at all. The transaction should happen if both actions $B\text{-}RR_{cnf}$ and $C\text{-}RR_{cnf}$ occur and establish the attribute value 'ready', otherwise the state of the accounts before the transaction request should be restored. Action *A-CR* models this decision which is represented by the attribute values '*commit*' and '*rollback*', respectively.

Furthermore, action *A-CR* enables the process of updating or restoring the accounts at banks B and C and the release of the reserved resources. Once both actions $B\text{-}CR_{cnf}$ and $C\text{-}CR_{cnf}$ have happened, a bank note can be made for the client, which is modelled by action *BN*. The updating of the account at bank A can be modelled by either action *BN* or action *A-CR*. This decision may depend on reliability criteria and the amount of interest involved.

## 5.5 Money transfer communication infra-structure

In the money transfer protocol design so far we have assumed that bank A is directly connected to the client and to the other banks. The objective of this design milestone is to allow these entities to be geographically distributed by the use of a common communication infra-structure.

This implies that the interactions between different entities should be mapped onto interaction or communication patterns that are supported by the communication infra-structure. We illustrate how this mapping can be achieved for part of the behaviour of the interaction system between banks A and B: the actions $B\text{-}RR_{req}$ and $B\text{-}RR_{cnf}$ and their causal relationship, which corresponds to the account reservation service behaviour.

Figure 25 depicts three design steps in which the account reservation service is refined and decomposed into two protocol entities and a data transfer service provider.

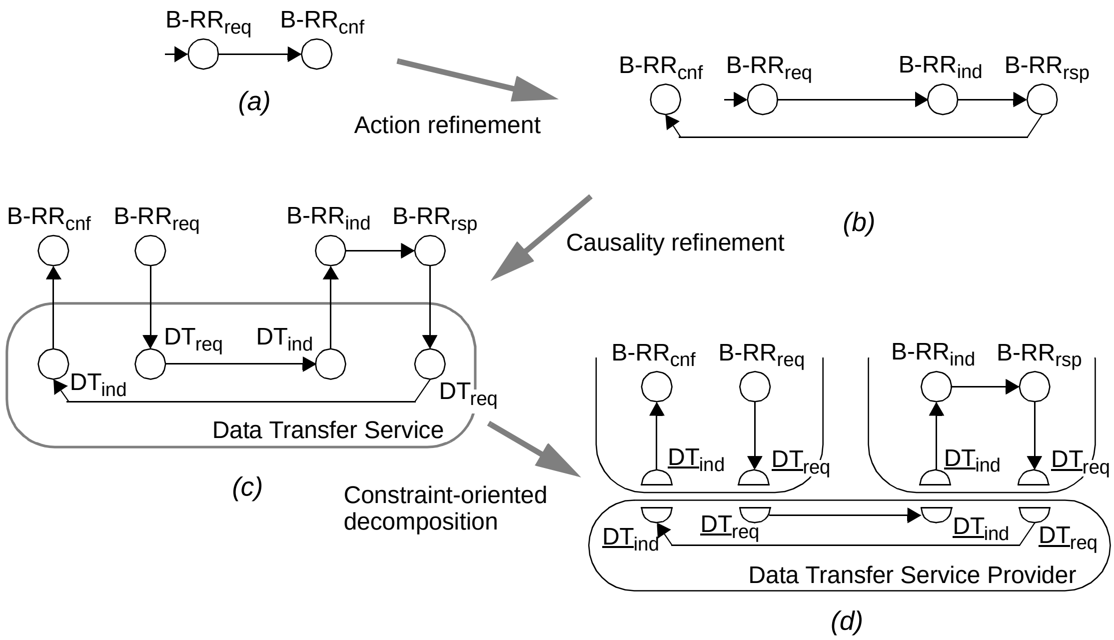



*Figure 25: Communication infrastructure*

The objective of the first design step in Figure 25 is to design the account reservation service taking into account that banks A and B are geographically distributed, i.e. they have to interact via a third party. This is achieved by refining the account reservation service into a user confirmed service, according to the rules of action refinement. Figure 25 (b) depicts the refined account reservation service. Actions $B\text{-}RR_{req}$ and $B\text{-}RR_{cnf}$ are performed at bank A and actions $B\text{-}RR_{ind}$ and $B\text{-}RR_{rsp}$ are performed at bank B.

The objective of the second design step is to refine the account reservation service of Figure 25 (b) into an account reservation protocol in which the remote causality relations between actions $B\text{-}RR_{req}$ and $B\text{-}RR_{ind}$ and actions $B\text{-}RR_{rsp}$ and $B\text{-}RR_{cnf}$ are implemented by a generic reliable data transfer service. The causality refinement design operation can be applied to obtain the account reservation protocol, which is depicted in Figure 25 (c).

The objective of the third design step in Figure 25 is to decompose the account reservation protocol into protocol entities and a lower level service provider. Figure 25 (d) depicts this decomposition. The protocol entities define part of the behaviour of banks A and B. The data transfer service provider defines part of the behaviour of the communication infrastructure.

## 6 Conclusions and further work

This paper discusses some basic design concepts for distributed system design. Basic design concepts help the designer to conceive, structure and refine the characteristics of a system. We believe that the development of an effective design methodology should be based upon a careful choice, correct understanding and precise definition of its basic design concepts.

The behaviour structuring techniques presented in section 3.3 follow from the concepts of action, interaction and causality relation. Constraint-oriented behaviour structuring is based upon the implementation relationship between actions and interactions. Causality-oriented behaviour structuring is merely a syntactic operation which allows one to distribute the result action and its enabling and disabling conditions over separate sub-behaviours.

The causality and action refinement design operations presented in section 4 follow from the concepts of action and causality relation. Both types of behaviour refinement are defined in terms of manipulations of actions and action attributes, and manipulations of their causality relations.

The definition of generic design milestones in section 3.5 is enabled by the introduction of the action concept. Causality- and constraint-oriented behaviour structuring, and causality and action refinement are used to perform design steps between successive design milestones.

Our design concepts provide a sound basis for further work on other elements of our design methodology. In the near future we will concentrate on the definition of a concise, easy to use and effective design language. Further, we will work on the formalization of this design language and the presented design operations, aiming at the development of supporting tools.

## Acknowledgements

We would like to thank Mark de Weger for interesting discussions on some of the topics of this paper. We also would like to thank the anonymous reviewers for their useful comments and suggestions for improvement.

This work is partly funded by the Dutch Ministry of Economic Affairs in the Platinum project.

## References

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

[2] L. Ferreira Pires. *Architectural notes: a framework for distributed systems development*. PhD thesis, University of Twente, Enschede, the Netherlands, 1994.

[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] ISO. Information Processing Systems – Open Systems Interconnection – Basic Reference Model, 1984. IS 7498.

[5] ISO/IEC JTC1/SC21/WG7. Basic Reference Model of Open Distributed Processing. ISO 10746, 1993. Part 1 to 4.

[6] D. A. C. Quartel, L. Ferreira Pires, H. M. Franken, and C. A. Vissers. An engineering approach towards action refinement. In *Proceedings of the Fifth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, pages 266–273. IEEE

Computer Society Press, 1995.

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

[8] K. J. Turner, editor. *Using Formal Description Techniques. An Introduction to Estelle, LOTOS and SDL*. John Wiley & Sons, 1993.

[9] M. van Sinderen. *On the design of application protocols*. PhD thesis, University of Twente, Enschede, the Netherlands, 1995.

[10] M. van Sinderen, L. Ferreira Pires, C. A. Vissers, and J.-P. Katoen. A design model for open distributed processing systems. *Computer Networks and ISDN Systems*, (27):1263–1285, 1995.

[11] 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.

[12] 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.

[13] 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 Specification, Testing, and Verification, XIII*, pages 3–26, The Netherlands, 1993. Elsevier Science Publishers B.V. (North-Holland).