Advances in Architectural Concepts to support Distributed Systems Design

Luís Ferreira Pires, Chris A. Vissers, Marten van Sinderen
Tele-Informatics and Open Systems Group
University of Twente
P.O. Box 217, 7500 AE Enschede, the Netherlands
e-mail: {pires,vissers,sinderen}@cs.utwente.nl

Abstract

This paper presents and discusses some architectural concepts for distributed systems design. These concepts are derived from an analysis of limitations of some currently available standard design languages. We conclude that language design should be based upon the careful consideration of architectural concepts. This paper aims at supporting designers by presenting a methodological design framework in which they can reason about the design and implementation of distributed systems. The paper is also meant for language developers and formalists by presenting a collection of architectural concepts which deserve consideration for formal support.

1 Introduction

Architectural concepts are abstractions (models) of frequently occurring aspects of technical objects. Such concepts are manipulated during the design and implementation process. Examples are service, service access point, service primitive, service primitive parameter, service data unit, protocol, protocol data unit, abstract interface, real interface, etc. Naturally such concepts should find a straightforward reflection in the design language that is used.

Once a design language is introduced, however, one easily runs the risk of only considering the characteristics of a technical object in the light of the design model imposed by this language. As a result, architectural concepts may be obscured by pre-conceived language limitations or even unintended design decisions may be forced.

Furthermore, attempts to enhance the expressive power of a design language that are purely based on the manipulation of the semantic model, though resulting in sound mathematical solutions, may be of no practical use. Enhancement in expressive power must also follow from careful consideration of the architectural concepts involved, in order to result in a coherent design model.

The relevance of the work reported in this paper resides in the fact that it concentrates on architectural concepts to provide engineering support for the design of distributed systems, whereas a design language is merely considered as a means to represent and manip-

ulate these concepts. Available standard design languages for distributed systems design show severe limitations in the representation of architectural concepts, possibly forcing designers to take improper design decisions. Sometimes these limitations are confused with abstraction. However, when a certain language model does not allow us to formally represent characteristics directly, it is not fair to simply state that we abstract from these characteristics. Rather one should acknowledge the language limitations and find ways to compensate for them.

When investigating the expressive power of LOTOS we noticed that more attention should be given to architectural design concepts. This paper gives account of some results of this research. It introduces and justifies a set of architectural concepts for designing distributed systems, and shows how these concepts can be used to make design choices explicit in the design trajectory. The paper addresses some design and structuring techniques and some refinement options that are useful implementation notions.

This paper is structured as follows: section 2 introduces the notion of design culture, as a framework in which the importance of architectural design concepts is acknowledged; section 3 discusses the architectural concepts which are supported by some currently available design languages, and identifies some of their limitations; section 4 introduces a collection of basic architectural concepts; section 5 discusses the representation of behaviours of interaction systems; section 6 addresses some structuring facilities for behaviour definition; section 7 indicates how the architectural concepts introduced in section 4 can be manipulated in realistic instances of design. Conclusions are drawn in section 8.

2 Framework: Design Culture

The purpose of the design process is to produce a technical object: a real system. At the beginning of the design process the system does not exist. Yet it must be conceived, analysed, manipulated and communicated among designers. This means that, at each stage in the design process, the technical object has to be represented by describing only those characteristics that are relevant at that stage in the design process and abstracting from details that are irrelevant at that stage.

Architectural (design) concepts, being abstractions of aspects of technical objects, can be used to design technical objects, by making a composition of such concepts. To conceive, manipulate, analyse and communicate designs, designers should be able to express them in a comprehensive, complete and unambiguous way. This implies that a design language, as a notation for *representing* designs, is necessary. Elements of a design language (e.g. syntax and semantics elements) must be derived from generic design concepts related to the technical area of concern, making the language general purpose in its application area. Formal models cater for the unambiguous interpretation of a design, since they are based on precise mathematical models.

This means that a distinction must be made between a design as an architectural notion and its specification, the latter being merely a representation of a design in a chosen design language.

A design language is only suitable for representing designs if there can be made a clear relationship between design concepts and compositions of language elements to represent them. We define the term *architectural semantics* as the relationship between architectural concepts and their possible representations in a design language. A design language

should allow a designer to concentrate on the design and the architectural concepts, and use the design language merely as a vehicle to represent design characteristics.

Design languages should also serve to support a design methodology. Effective design methodologies are based on the concept of separation of concerns and abstraction. While at a certain level of abstraction we recognize a relationship between a design and its representation, designs also relate to each other at different abstraction levels. This means that a design at a certain level of abstraction has to be elaborated according to specific design objectives, which are consistent with the design methodology. Examples of such design objectives are the incorporation of design decisions that manipulate some architectural concept, and the application of qualitative design principles.

Another important aspect 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 verifications, transformations, simulations, etc. of designs.

Designs are formulated by humans acting as designers, who have personal preferences or styles. General purpose specification styles can be derived from qualitative design principles and design objectives, being the representation of a design methodology in terms of the elements of a design language. Furthermore the design language should support the use of the specification styles dictated by the design methodology.

The collection of conventions, concepts, tools, methods, etc., determine the way systems are developed in a certain environment of designers, characterizing what we call the *design culture* of that environment ([17]). Figure 1 depicts the relationship between the elements of a design culture and how these relate to a design and its specification.

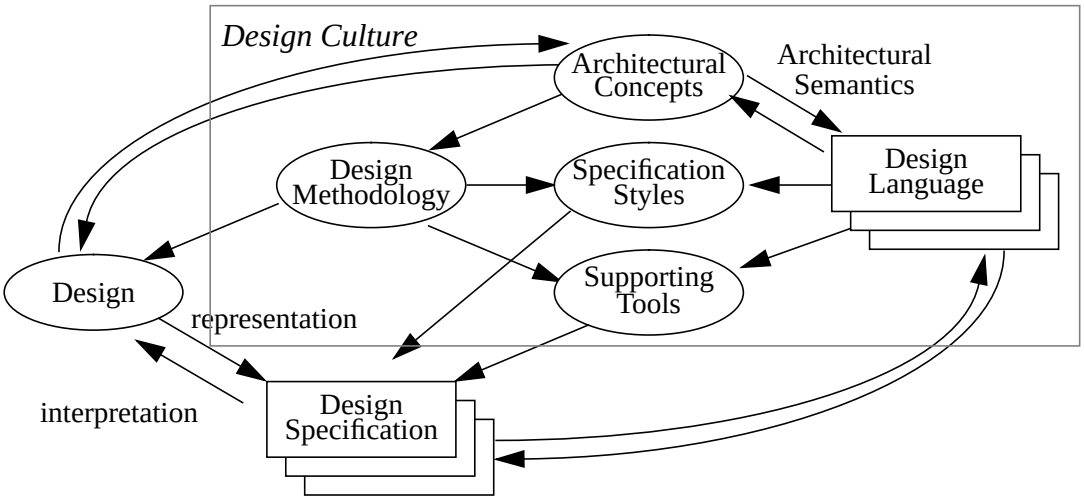


Figure 1: Application of the Elements of a Design Culture

In Figure 1 we notice the central position of architectural concepts: they influence the definition of the design language, and the design methodology, and constitute the objects to be manipulated in the elaboration of designs. Therefore architectural concepts form the foundation of a design culture.

3 Architectural Concepts in Current Design Languages

This section recalls some of the basic design concepts of current standard design languages and identifies a few of their limitations. Since these aspects have already been mentioned in a whole range of other publications (see [7] for example), we keep it short here.

We consider only two groups of formal design languages: (i) those based on finite state automata with asynchronous interactions via queues, such as SDL and Estelle, and (ii) those based on process algebra, i.e. labelled transition systems with synchronous interactions, such as LOTOS.

3.1 SDL and Estelle

SDL and Estelle define a system in terms of a set of modules which are interconnected by channels. The behaviour of each module is defined by a finite state automaton which receives and sends messages via channels to other modules. Channels are modelled as infinite queues.

The benefits of this model are at the level of software implementation. Many commercially available operating systems, for example, present facilities for interprocess communication using channels and messages. Implementations of SDL and Estelle specifications under these operating systems can be rather straightforward.

The limitations of this model are that designers are forced to put queues between any two communicating modules. Even in situations where queues are not necessary nor desirable, for example when one module is decomposed in two modules, this model has already imposed this design choice. Certain behaviour patterns such as backpressure, negotiation of interaction values, disruption of behaviour, etc. cannot be represented, unless one uses complicated message exchanges via the queues, which obscure the interpretation of these behaviours. Complex behaviours are hard to structure in terms of finite state automata, which forces designers to take early implementation decisions that may be hard to justify at that level of abstraction.

We conclude that the design concepts used in SDL and Estelle are at a too low level of abstraction, are based on specific technological arguments, and targeted to specific implementation structures. Consequently SDL and Estelle have less expressive power and structuring facilities to support design at higher levels of abstraction. Designs in these languages have to be structured considering a lot of irrelevant internal details from the beginning of their elaboration.

3.2 LOTOS

LOTOS defines a system in terms of a process, or a composition of processes, such that each process interacts with its environment via synchronous events. More than two processes can participate in an event (multi-way synchronization). Events are not restricted to message passing, but they can also represent other forms of interaction such as value matching and value generation.

The interaction concept in LOTOS appears to be a powerful expressive element. Limitations in LOTOS are however in its too elementary basic semantics: the eventual execution of an event once it is enabled, non-deterministic choice, and interleaving. Expression of

explicit timing, real parallelism, probabilistic features, and modality are lacking. Some of these limitations will be illustrated in the sequel with small examples.

3.2.1 Real Parallelism versus Interleaving

Consider a bi-directional data buffer, with capacity one in each direction of communication. In order to simplify the discussion, we consider only one instance of communication per direction. We also consider two interaction points *a* and *b* where interactions *in* and *out* with data values occur, representing insertion and retrieval of data respectively. Figure 2 depicts this example.

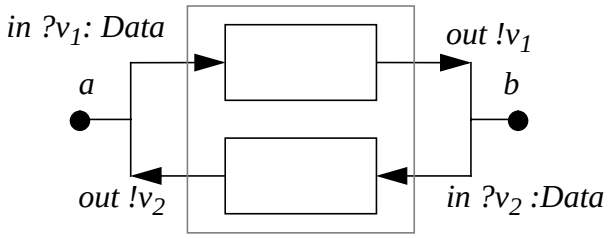


Figure 2: Bi-directional Buffer Example

Suppose the behaviour of the buffer is informally defined as follows: in case data is introduced at *a* (*b*) through an interaction *in*, it can be retrieved at *b* (*a*) through an interaction *out*; all interactions at *a* (*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 *)
```

By inspecting the behaviour tree of the example, which is depicted in Figure 3, we notice that the formal semantics of LOTOS[1] has introduced an extra property to the behaviour, namely that some interactions that were supposed to be independent are also interleaved. The consequence is that a designer cannot infer from the specification if the interactions must be interleaved, which is the case for the pairs <*a !in ?v , a !out ?v*> and <*b !in ?v , b !out ?v*>, or if the interactions are 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, our area of concern, the introduction of extra 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

---

1. In this paper we only consider the formal semantics as it has been published in the International Standard IS 8807. Extensions, such as the one proposed in [11], are not being considered here.

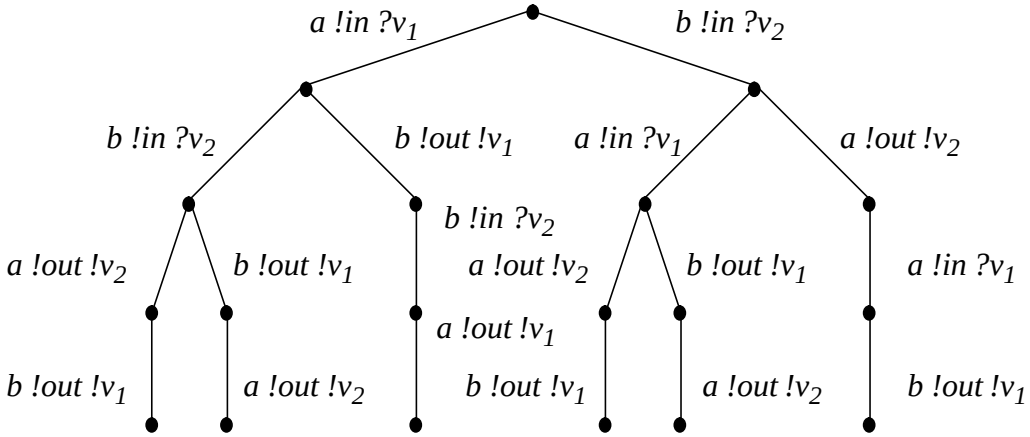
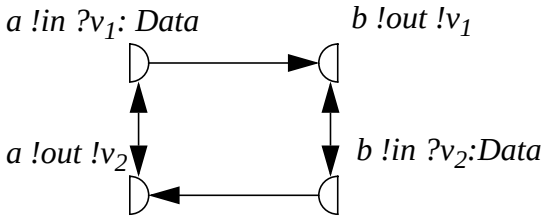


Figure 3: Behaviour Tree of the Bi-directional Buffer Example

objective of unambiguous interpretation of using a formal language, and in the latter it may generate low quality solutions (e.g. bad performance).

Figure 4 depicts in an ad hoc notation the desired behaviour, by representing only the essential relationships between the interactions.



(and) represent an interactions

↔ represents that the interactions are interleaved, i.e. they do not happen at the same time

→ represents that the interaction of the side without arrow causes the interaction at the side of the arrow

Figure 4: Example in True Concurrency Semantics

3.2.2 Timing

Now suppose we take the informal specification above and modify it to include some timing requirements, in the following way: in case data is inserted in *a* (*b*), it must be retrieved in *b* (*a*) within 10 seconds.

Suppose we would have some extension of LOTOS in which the time of interaction occurrences could be explicitly represented. In this case we could define the behaviour of the bi-directional buffer such that *out* in *b* (*a*) occurs within 10 seconds after *in* in *a (b)*, but this might not be enough. According to the model of LOTOS we also have to *suppose*,

for example, that the environment of the buffer is always ready to accept *out*'s, after their corresponding *in*'s have taken place. The odd thing here is rather subtle: we would describe *only* the system formally, and at the same time impose restrictions on the environment *informally*, corrupting in this way the objective of unambiguous interpretation of using a formal language.

We conclude that, although LOTOS supports the representation of interactions at a rather high level of abstraction and has elegant formal semantics and structuring mechanisms, it lacks expressive power to represent important aspects of realistic distributed systems.

A lot of work has been done to improve the expression power of LOTOS along the lines mentioned in this section (see for example [1, 3, 11]). Most of this work, however, consists of manipulations of the LOTOS formal semantics and syntax. This inspired us to rather concentrate our research on the necessary architectural design concepts.

4 Basic Architectural Concepts

This section introduces some basic architectural concepts for the design and implementation of distributed systems. The novelty of our approach is to consider both *actions* and *interactions* in a single framework for achieving specific design objectives. Our language approach, though, is kept quite elementary. We represent behaviours of functional entities in terms of causality relations between its actions and interactions. Problems such as the ones caused by interleaving semantics are in this way avoided.

4.1 Actions and Interactions

Usual interpretations of a (behavioural) system specification can be found in [9]. A system specification either (i) constrains the behaviour of its environment or (ii) it does not state what happens in the unspecified cases. In other words a system specification defines what the system does under pre-defined conditions which should be known and respected by the system's environment, and says nothing about the unknown pieces of behaviour.

In LOTOS a system is defined as a component *separately* from its environment. This is done in terms of possible orderings of *interactions*, value establishment in these interactions, and the constraints on these values, *as imposed by the system*. The system's environment can be defined in the same way. The definitions of the system and its environment *together* determine a *common behaviour*: *what* interactions in *what* order and with *what* value attributes are actually established. This common behaviour is what the user is really interested in, and therefore it should be specified first and be used later to derive the behaviour of the system. For the latter there are many possible combinations of system and environment definitions that result in the same common behaviour, leading to design freedom for choosing these definitions.

We call this common behaviour the *interaction system* between the system and its environment. In the definition of the interaction system only the result of each interaction is defined but not the different ways in which the system and its environment may contribute to the establishment of these results. The interaction system defines at a high level of abstraction *what* happens in terms of established interactions, not *how* it happens.

Taking the example mentioned before (in case data is established in *a* (*b*), it must be retrieved in *b* (*a*) within 10 seconds) it could be accomplished in amongst others the following ways:

1. the buffer has a certain delay shorter than 10 seconds and the environment has no constraints for the acceptance of data (interaction *out*) in *a* and *b*;

2. the buffer has no measurable delay and the environment reads the data within 10 seconds after it is stored.

This means that restrictions can be placed in the participants of an interaction quite freely, under the condition that the desired integrated behaviour is implemented. The required implementation notions in this case are expected to be rather different than the ones defined so far for process algebras.

An *integrated interaction* is an interaction viewed in such a way that the distribution of individual responsibilities and constraints amongst the involved functional entities (system, environment or system parts) is ignored. An *action* is an occurrence inside a functional entity. Since we have abstracted from the individual responsibilities of involved functional entities when defining an integrated interaction, an integrated interaction of an interaction system can be considered as an action of an interaction system.

From now on we use the term action to denote integrated interactions. It is interesting to remark that although all integrated interactions are actions, some actions cannot be called integrated interactions, since one may have actions that in the course of the design process are not distributed over system parts.

The possibility of having both actions and interactions explicitly represented in a design model seems to be one of the innovative elements in our approach.

We use the neutral term *event* to refer to an action or an interaction in the sequel.

4.2 Observability: Description versus Prescription

Behaviour is inherently related to observability. We can only define behaviour in terms of what we can observe. The aspects being observed, how observation is performed, and the purpose of the behaviour definitions are what actually make the difference.

In the design model of LOTOS, observability exists by means of interaction, thus an environment observes the system by interacting with it. The purpose of observability in this case has been to *describe* the behaviour of the system.

Alternatively, we can consider observability from the point of view of designers, such that actions and interactions are observed at a meta-level with respect to the system and its environment. The purpose of observability in this case is to *prescribe* the behaviour of the system, which can be used to construct the system.

4.3 Event Attributes

We expect events to have the following attributes: location, time, value(s) of information, functionality, and probability. These attributes are introduced in order to allow events and their relationships to be defined. Each of these attributes is briefly discussed below:

- *location attribute*: this attribute can be used as a reference for the system parts that perform an event. It may be used in lower abstraction levels to delimit these system parts. This attribute can be also called *(inter)action point*;

- *time attribute*: this attribute indicates the moment or period of time when an event occurs or can occur. It also determines the moment of time when the value(s) of information established in the event can be referred to by other events. Time attributes have been understood and represented in many different ways in different design models (absolute or relative time, continuous or discrete time, global time, local time, etc.). In our design model we consider that time is always relative, and that events may refer directly to time attributes of other events if necessary;

- *values of information*: the purpose of defining events is to use them as means to establish values of information. Values of information established in an event can be of arbitrary complexity (a single value, a set of values, a list of values, etc.), defining complex information or data structures. In order to be manipulated effectively, only those characteristics of these values of information which are relevant at the level of abstraction 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 supports abstract representation and structuring of data is the *Abstract Data Type* (ADT) theory;

- *functionality attribute*: this attribute defines the set of values of information passed to this event by previous events and that may be referred to by successive events. This set of values of information may refer to other attributes of other events that have taken place before, becoming the representation of the *relevant history* for an event. Roughly comparing, this attribute is 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;

- *probability attribute*: this attribute defines the probability that an event occurs according to its definition, once this event is *enabled*. We say that an event is enabled if all conditions for this event to occur are satisfied.

Event attributes are important for the definition of behaviours. Fundamental design steps of a design process can be defined in terms of the manipulation of events and their attributes. Implementation notions can also be defined in terms of relationships between these attributes at consecutive levels of abstraction.

5 Interaction System Behaviour

We define the behaviour of an interaction system by a set of relationships amongst actions that altogether determine the possible ways an interaction system can function. Similarly, the behaviours of a system, its environment and its parts can be defined in terms of relationships and dependencies between events. Section 6.2 addresses these behaviour definitions.

We assume that we always can distinguish between actions in behaviours. The identification of such distinct actions is guided by the technical needs to distinguish them. Distinct

actions are supposed to be unique, even if two or more distinct actions have identical attributes. This implies that each distinct action of a behaviour can be identified by a unique *action identifier* for the purpose of this discussion. An action identifier is used here to represent that an action happens, while action attributes precisely define the characteristics of this action.

5.1 Behaviour Elements

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

- *initial actions*: are those actions which are allowed to occur independently of the occurrence of other actions of that behaviour. These actions may refer to an initial set of attribute values (time reference, initial values for functionality variables, etc.), which have been established before the behaviour is activated. They may occur spontaneously as the behaviour is instantiated (*start* actions), or may be enabled by other behaviours (*entry points*);

- *causality contexts*: the causality context of an action defines the role of this action in a behaviour. Possible time orderings of actions are implicitly defined by their causality contexts;

- *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, values of information and functionality. An action is said to terminate a behaviour if no more actions or other behaviours are enabled by it, characterizing *deadlock.*

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 defines the conditions which enable and constrain the occurrence of an action. The reason for using causality as the basis for defining behaviour is based on the fact that a distributed behaviour should not have a global state, but rather a collection of ‘sub-states’ depending on the relationships between actions and action occurrences. These ‘sub-states’ are defined in the causality relations.

5.2 Basic Causality Relations

The occurrence of an action may depend on the occurrence of a previous action, characterizing *causality*. For example, suppose an action a_2 is only allowed to happen if another action a_1 has happened before.

We define the causality relation a_1 -> a_2 as:

> *the occurrence of an action a_1 is a condition for the occurrence of action a_2. If a_1 occurs then a_2 is allowed to occur.*

The causality relation above says nothing about the conditions for the occurrence of a_1. These are defined in the causality relation of a_1, which is not part of the causality relation of a_2.

The occurrence of an action may depend on the non-occurrence of another action, characterizing a special type of causality, which is often called *conflict*. For example, suppose an action a_2 is only allowed to happen if another action a_1 does not happen. Since this condition has to be evaluated at the moment we decide whether a_2 takes place or not, we consider that a_1 does not happen before nor at the time a_2 happens.

We define the causality relation $\neg\, a_1 \text{ -> } a_2$ as:

> *the non-occurrence of an action a_1 (before or at the same time a_2 happens) is a condition for the occurrence of a_2.*

These two basic causality relations are used as building blocks to build arbitrarily complex causality relations.

5.3 Generic Causality Relations

Causality relations define the patterns of behaviour that enable an action to occur. These patterns of behaviour can be represented by logical combinations of occurrence or non-occurrence of actions, or both, using *and* ($\wedge$) and *or* ($\vee$) logical operators. This allows one to compose causality relations of arbitrary complexity.

We consider some typical examples below.

5.3.1 Conjunction of Occurrences

$$a_1 \wedge a_2 \text{ -> } a_3$$

This causality relation states that the occurrences of both a_1 and a_2 is a condition for the occurrence of a_3.

> Considering that a_1 and a_2 have no relationship with each other, the behaviour above could be expressed in the following way in LOTOS:
>
> ```
> ( a1; exit ||| a2; exit ) >> a3; stop
> ```
>
> This comparison only holds modulo interleaving semantics, and disregards the fact that in LOTOS we actually describe interactions. It has been included here only in order to illustrate our ideas to the reader familiar with LOTOS. These remarks also apply to the next comparisons.

5.3.2 Disjunction of Occurrences

$$a_1 \vee a_2 \text{ -> } a_3$$

This causality relation states that the occurrence of a_1 or a_2 is a condition for the occurrence of a_3. Notice that in this case a_1 and a_2 may even both happen, but the occurrence of one of them is sufficient for the occurrence of a_3. In case both a_1 and a_2 happen 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 either a_1 or a_2, this choice determines which attributes are used (from a_1 or from a_2, but not from both).

> Considering that a_1 and a_2 have no relationship with each other, the behaviour above could be expressed in the following way in LOTOS:

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

Notice that a LOTOS definition of this behaviour in a rather structured way requires the introduction of some form of internal communication through internal gates. Recalling that internal gates characterize a resource-oriented style (see [18]), which should be used to represent internal structure, this internal gate only obscures the interpretation of the specification. Another possibility would be to explicitly include all the traces which result in the desired condition. In this case the structure of the specification would have been completely flattened.

Disjunction of occurrences is useful in the definition of some kinds of error handling behaviours, where many errors may happen, but only one of them in a non-deterministic way causes some reaction. Being a relevant behaviour pattern, it should be better supported by the design model.

5.3.3 Conjunction of Occurrence and Non-occurrence

$$a_1 \wedge \neg\, a_2 \text{ -> } a_3$$

This causality relation states that occurrence of a_1 and the non-occurrence of a_2 are both conditions for the occurrence of a_3.

Considering that a_1 and a_2 have no relationship with each other, the behaviour above could be expressed in the following way in LOTOS:

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

5.3.4 Disjunction of Occurrence and Non-occurrence

$$a_1 \vee \neg\, a_2 \text{ -> } a_3$$

This causality relation states that occurrence of a_1 or the non-occurrence of a_2 are conditions for the occurrence of a_3.

Considering that a_1 and a_2 have no relationship with each other, the behaviour above could be expressed in terms of the possible traces in the following way:

```
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

Figure 5 depicts these causality relations. The graphical notation used in this figure will be used consistently throughout this text.

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 our 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 with actions A_B has the form $F(A)$ -> a_j, $A \subseteq A_B - \{a_j\}$, and where F is a formula using $\vee$, $\wedge$ and conditions of the form a_k and $\neg\, a_k$, representing the occur-

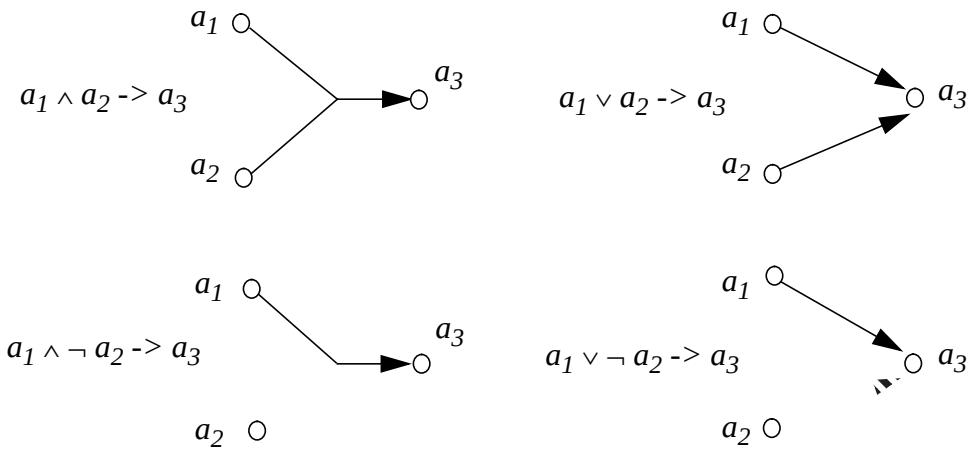


Figure 5: Some Causality Relations and their Graphical Notation

rence and non-occurrence of $a_k \in A$, respectively. The formula *F(A)* has to be evaluated to true at the moment a_j occurs.

5.4 Conditions and Constraints on Attributes

Action attributes can play two distinct roles, depending whether an action is a condition or a resulting action in a causality relation:

1. *attribute conditions*: premises (boolean functions) involving attributes of actions in a condition of a causality relation can be used to define additional conditions for the resulting action to occur;

2. *attribute constraints*: attributes of a resulting action can be constrained by premises (boolean functions). These premises possibly involve attributes of the actions which are present in the conditions for the resulting action, thus determining a set of allowed attribute values for this action.

This section further illustrates, by means of examples, some possible ways in which action attributes can be used to define conditions and constraints. The discussion here is far from complete. Due to size limitations, probability and functionality aspects are not discussed.

5.4.1 Conditions on Values of Information

The occurrence of an action may depend on the occurrence of another action with specific values of information. Suppose we have the following causality relation:

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

This causality relation states that only in case a_1 happens with value v_1 smaller than 10, a_2 is allowed to happen. In case a_1 does not happen or happens with v_1 greater or equal to 10, the condition for a_2 is false, and a_2 is not allowed to happen.

We may need to express that the occurrence of an action depends on the non-occurrence of another action with specific attribute values. Suppose that an action a_2 can only hap-

pen if an action a_1 with value $v_1 < 10$ has not happened. In order to make references only attributes of occurrences of actions, we indicate this in the following way:

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

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

5.4.2 Constraints on Value and Time Attributes

Attributes of an action can be also constrained in its causality relation, defining in this way a set of allowed values. Suppose an action a_1 is a condition for another action a_2, but that the possible attribute values of a_2 are constrained. These constraints may be defined in terms of references to attributes of a_1.

Some examples are:

$$a_1\ (v_1\text{: Nat}, t_1\text{: Time}) \text{ ->}$$
$$\quad a_2\ (v_2\text{: Nat}, t_2\text{: Time})\ [v_2 = v_1 + 2 \wedge t_2 < t_1 + 10]$$

Action a_2 can only happen for v_2 equal to $v_1 + 2$, and for t_2 smaller than $t_1 + 10$. For values that do not comply to these conditions, a_2 does not happen.

$$a_1\ (v_1\text{: Nat}, t_1\text{: Time}) \text{ ->}$$
$$\quad a_2\ (v_2\text{: Nat}, t_2\text{: Time})\ [v_1 < v_2 < 100 \wedge t_2 = t_1 + 1]$$

Action a_2 can only happen for v_2 greater than v_1 and smaller than 100, and for t_2 equal to $t_1 + 1$. For values that do not comply to these conditions, a_2 does not happen.

It is also possible to have conditions involving attributes of different actions. We illustrate this with the following example: suppose that we have 4 actions, a_1, a_2, a_3 and a_4, and that a_3 is only allowed to happen if a_1 and a_2 happen in this order, and that a_4 is only allowed to happen if a_2 and a_1 happen in this order. This can be specified in the following way:

$$(a_1\ (t_1\text{: Time}) \wedge a_2\ (t_2\text{: Time}))\ [t_2 > t_1] \text{ -> } a_3$$
$$(a_1\ (t_1\text{: Time}) \wedge a_2\ (t_2\text{: Time}))\ [t_2 < t_1] \text{ -> } a_4$$

Notice that in this example the conditions for the occurrence of a_3 and a_4 have been described using boolean functions involving time attributes of both a_1 and a_2.

In causality relations of the form $a_1 \vee\ a_2 \text{ -> } a_3$ we may need to refer to different values and in different ways in case the cause is a_1 or a_2. Suppose a_1 has value v_1, and a_2 has value v_2, and that in case a_1 is the cause of a_3, v_3 is a function f of v_1, and in case a_2 is the cause of a_3, v_3 is a function g of v_2. We could then define it in the following way:

$$a_1\ (v_1\text{: Data}) \vee\ a_2\ (v_2\text{: Data}) \text{ ->}$$
$$\quad a_3\ (v_3\text{: Data})\ [\quad \text{if } a_1 \text{ then } v_3 = f(v_1)$$
$$\qquad\qquad\qquad\quad \text{if } a_2 \text{ then } v_3 = g(v_2)\ ]$$

5.5 Generic Finite Behaviour

Finite behaviour can be represented by a set of causality relations, one relation per action of the behaviour. Some actions may be enabled from the beginning of the behaviour (initial actions), while some others depend on occurrences of other actions in order to be enabled.

Consider the following example:

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

start -> a_0 states that a_0 is enabled from the beginning of the behaviour, i.e. it does not have to wait for other actions. $\neg\, a_0$ -> 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*. The behaviour definition *B* also determines the causality context of all its actions. For example, a_2 is the resulting action in the causality relation a_1 -> a_2 and appears in the causality relations {$a_1 \wedge \neg\, a_2$ -> a_3, a_2 -> a_4}, which completely defines the role of a_2 in *B*.

Figure 6 depicts this example using our graphical notation.

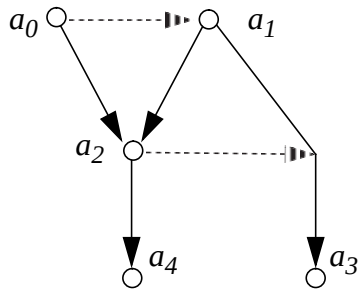


Figure 6: Graphical Representation of the Example

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 the *monolithic specification style* of [18], in which behaviours are described as monolithic wholes, without any structuring.

6 Structured Interaction System Behaviour

Experience has shown that complex systems cannot always be represented in a monolithic fashion. We anticipate that the description of an interaction system between a system and its environment must be structured, in the case of complex systems, in order to be developed, understood, manipulated and maintained.

Causality relations, as presented so far, are a compact and parsimonious notation to define relationships between actions, but lack structuring. Furthermore this notation is restricted to finite behaviours. Therefore this section builds on the previous section, presenting some mechanisms to structure designs and to represent repetitive and infinite behaviour.

6.1 Sequential Composition

Sequential compositions of behaviours are characterized by the fact that conditions inside an instance of behaviour enable actions of another instance of behaviour, in a similar way as conditions on actions enable result actions in causality relations.

6.1.1 Entries and Exits

Entries and exits are introduced in our design model as mechanisms to represent sequential composition of behaviours. Suppose B_1 and B_2 are behaviours, defined as sets 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. This can be generalized to more than one entry, more than one exit, or both.

We illustrate this with the following example:

$B_1 := \{$ *start* -> a_1, *start* -> a_2,
$a_1 \wedge a_2$ -> a_3,
a_3 -> a_4,
a_3 -> a_5,
$a_4 \wedge a_5$ -> *exit* $\}$

$B_2 := \{$ *entry* -> a_6, *entry* -> a_7,
$a_6 \wedge a_7$ -> a_8 $\}$

The statements *entry* -> a_6, *entry* -> a_7 mean that a_6 and a_7 can be enabled by coupling a condition to this *entry*. $a_4 \wedge a_5$ -> *exit* means that if a_4 and a_5 happen, the exit condition is true. We can define a sequential composition between B_1 and B_2 in the following way:

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

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. The resulting behaviour can be obtained by the ‘short-circuit’ of the *exit* of B_1 and the actions of the *entry* of B_2.

Figure 6 depicts this example, showing that the conditions of the exit of B_1 have turned into conditions for the occurrence of a_6 and for the occurrence of a_7.

Exits can also be used to refer to negation of conditions. Consider the following example:

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

$B_2 := \{$ *entry* -> a_3 $\}$

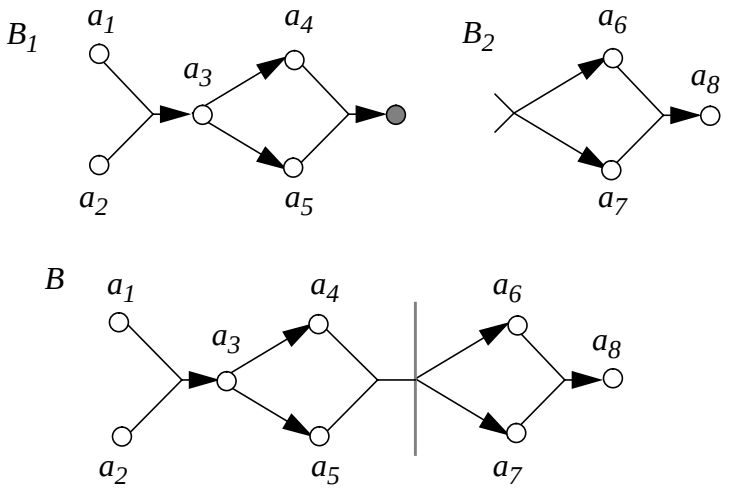


Figure 7: Example of Sequential Composition of Behaviours

The behaviour definition ¬ B_1 *(exit) -> B_2 (entry)* makes the negation of the exit of B_1, i.e the non-occurrence of both actions a_1 and a_2, a condition for the occurrence of a_3.

6.1.2 Attributes in Entries and Exits

Entry/exit constructs can be also used to pass information from the *exit*ing behaviour to the resulting behaviour. Taking for example B_1 and B_2 above, we can use this construct to relate attributes of actions of the exit of B_1 to the attributes of actions of the entry of B_2. A requirement for the exit/entry coupling between two behaviours is that there must be a matching between the values defined in the exit conditions and the values expected in the corresponding entry condition.

For example, we could have:

$$B_1 := \{ ..., a_4 (v_1 : T_1) \wedge a_5 (v_2 : T_2) \text{ -> } exit (v_1, v_2), ...\}$$

$$B_2 := \{ entry (a: T_1, b: T_2) \text{ -> } a_6, entry (a: T_1, b: T_2) \text{ -> } a_7, ... \}$$

In this case the behaviour $B := \{ B_1 (exit) \text{ -> } B_2 (entry) \}$ would have been sound, because 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.

Summarizing, the entry/exit constructs can be used in the following way:

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

2. an *exit* can be used to define sequential composition of behaviours in order to indicate that *C(B(exit))* is valid. This can be done by explicitly referring to the exit in the condition of a causality relation (such as in *B(exit)*);

3. an *exit* can be used to define sequential composition of behaviours to indicate that *C(B(exit))* is not valid (¬ *C(B(exit))*). This can be done by explicitly referring to the negation of the exit in the condition of a causality relation (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 sequential composition of behaviours to indicate that the conditions of a causality relation apply to each action of *A(B(entry))*. This is done by referring to the entry in the result of a causality relation (as in *B(entry)*);

Considering again the example depicted in Figure 6, notice that $A(B_2(entry)) = \{a_6, a_7\}$ and $C(B_1(exit)) = a_4 \wedge a_5$. The statement B_1 *(exit) ->* B_2 *(entry)* means that for each action a_i of $A(B_2(entry))$, there will be a causality relation of the type $C(B_1(exit))$ -> a_i.

6.1.3 Multiple Entries and Exits

We generalize the exit/entry constructs, by considering that a behaviour may have more than one exit conditions or entry points or both. The consequence is that, in order to allow a proper combination of exit/entry pairs, exits and entries must be identified.

We illustrate this with an example:

B_1 := { *start* -> a_4, *start* -> a_5, *start* -> a_7,
$a_4 \wedge a_5$ -> $exit_1$
a_7 -> $exit_2$ }

B_2 := { $entry_1$ -> a_8,
$entry_2$ -> a_9 }

The behaviour definition $\{B_1 (exit_1)$ -> $B_2(entry_1)$, B_1 $(exit_2)$ -> B_2 $(entry_2)\}$ means 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. It is important to remark that B_1 $(exit_1)$, B_1 $(exit_2)$ and $B_2(entry_1)$, B_2 $(entry_2)$ refer to the same instance of B_1 and B_2, respectively.

Figure 8 illustrates the effect of this combination mechanism with generic behaviours B_1, B_2, B_3 and B_4. Notice that the exit/entry constructs define a line that delimits the behaviours by decomposing the causality relations. This mechanism allows us to structure a monolithic behaviour in sub-behaviours, in such a way that compositions of behaviour definitions can be created.

6.2 Composition of Constraints

An important structuring technique identified in [18] for the representation of behaviours is the *constraint-oriented style*. According to this style, a behaviour is represented as a conjunction of constraints on events, which are described in separate processes.

The consideration of this approach in our design model forces us to represent some actions in a distributed form. This happens since the global causality relation of each action to be distributed among multiple constraints is defined as a collection of causality relations in the different behaviours that represent these constraints. The combination of this collection of causality relations determine the desired global causality relation for each distributed action.

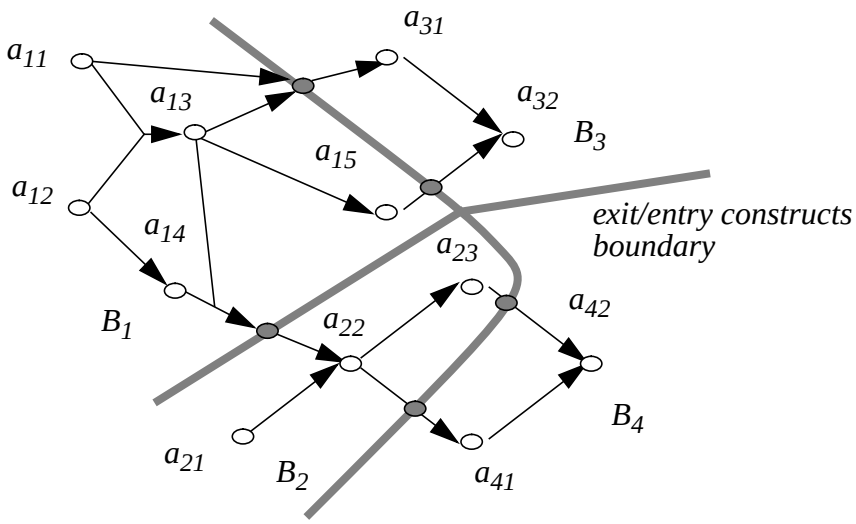


Figure 8: Generalized Exit/Entry Constructs

After the global causality relation of the distributed actions has been defined in separate behaviours, these behaviours can be assigned to the system and to its environment. Actually only at this point an action is transformed to an interaction.

Figure 9 depicts these two design steps, relating them to the two levels of abstraction of system structuring (interaction system, and system and environment) identified before.

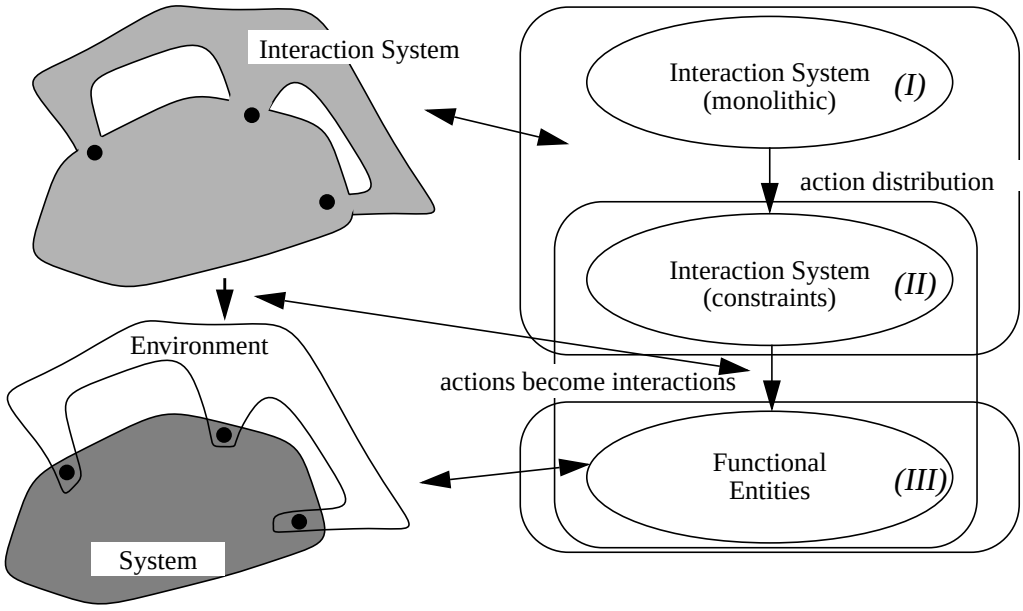


Figure 9: Transforming Actions into Interactions

We consider as an example a behaviour *B*, which contains a causality relation **a** ∧ **b** -> **c**. We also suppose that **a**, **b** and **c** may be or may be not distributed over behaviours B_1 and B_2. Actions which are not distributed are represented in this example with **bold**, while actions which are distributed are represented in *italics* when they turn into interactions. This analysis concentrates exclusively on the causality relation of **c** without loss of generality.

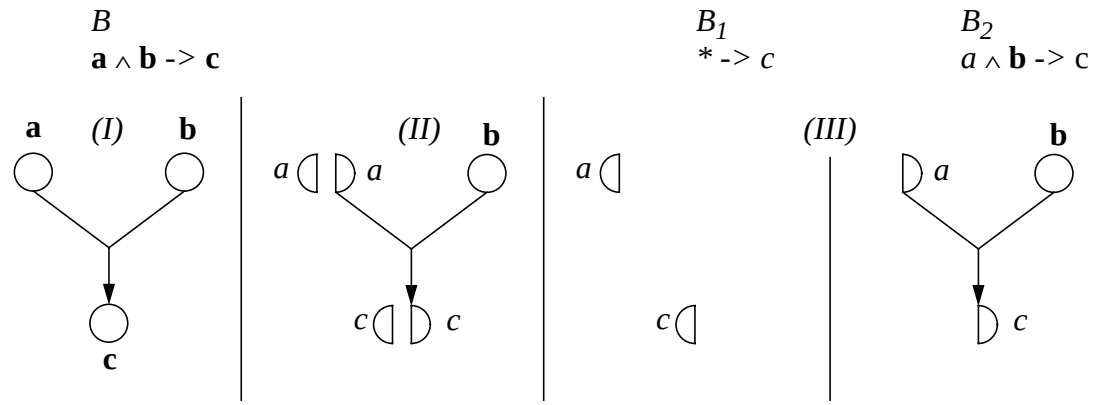
We spare the reader from the boring exercise of considering all possibilities for the decomposition of **a**, **b** and **c**, but rather present one of them below:

Distribute condition a (or b) and the result c

The correct options for the decomposition of the causality relation **a** $\wedge$ **b** -> **c** by distributing condition **a** and result **c** over B_1 and B_2 should preserve the original causality relation.

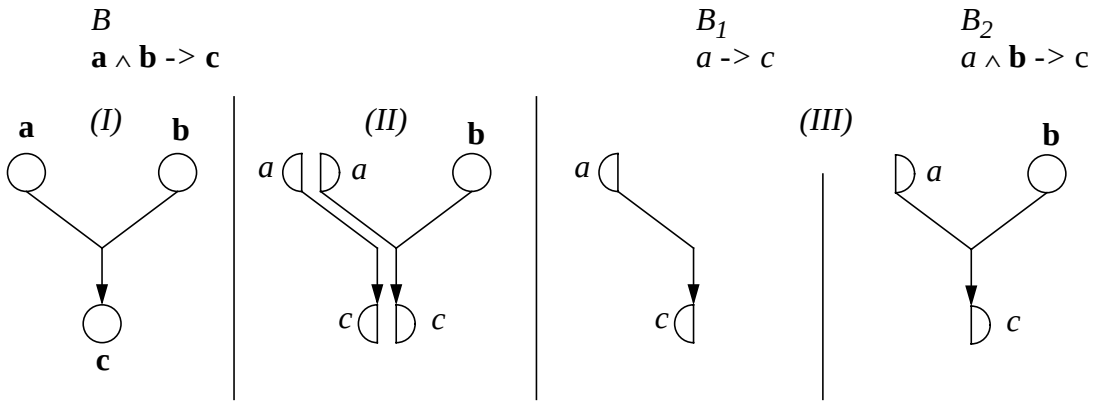
These options are presented and illustrated below. The illustrations also show the relationships between the distribution of causality relations and the design forms of Figure 9.

1. the whole causality relation is placed in one of the behaviours. In the illustration below we place the original causality relation in B_2.



   In the illustration above * -> *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 an initial action, or by any other conditions which are also necessary conditions of *a*.

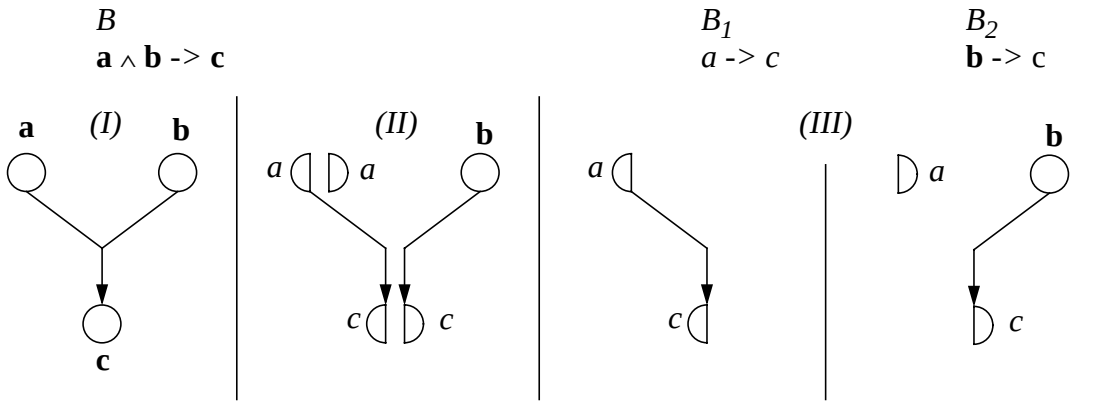
2. the whole causality relation is placed in one of the behaviours, but part of the causality relation is duplicated in the other behaviour. In the illustration that follows we have placed the original causality relation in B_2 and the causality between *a* and *c* is duplicated in B_1.



   Although the duplication of constraints in B_1 and B_2 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 in **a** two values of information v_1 and v_2 are established, and that both of them are used for the determination of the values of **c**. In this case we can use B_1 to constrain the dependencies with respect to v_1 and B_2 to constrain the dependencies with respect to v_2. If v_1 and v_2 characterize two distinct aspects of the functions of behaviour *B*, this structuring in constraints shall be preserved in later phases of design and may be found back in the implementation of the system.

3. the original causality relation is distributed over both behaviours. In the illustration below the causality between *a* and *c* is placed in B_1 and the causality between **b** and *c* is placed in B_2. The combination of these constraint yield the original causality relation.



The mirror images in which B_1 and B_2 are exchanged, and the distribution of **b** instead of **a** are also considered here.

Similar reasoning can be applied to other distributions of **a**, **b** and **c**, and to other forms of causality relations. It is left to the reader to find out that the distribution of disjunctions has less correct possibilities than the distribution of conjunctions.

Figure 8 illustrates the effect of composition of constraints with four generic behaviours B_1, B_2, B_3 and B_4.

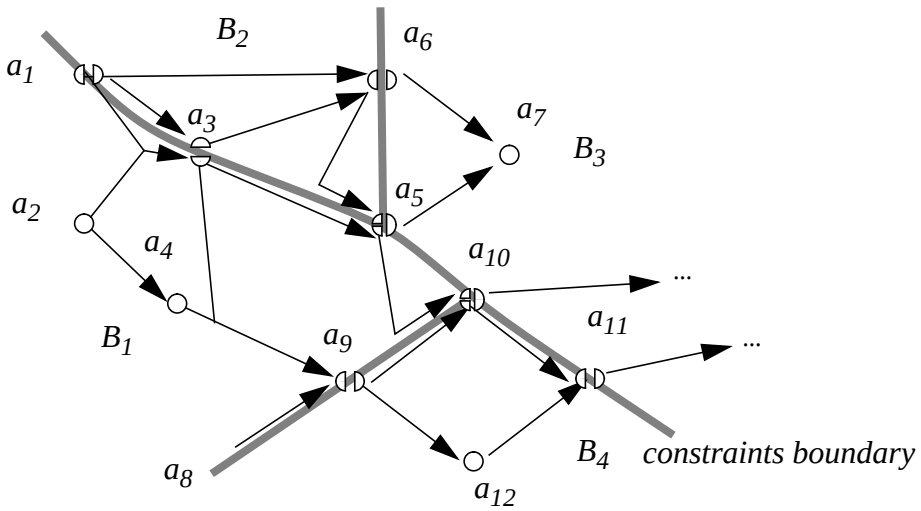


Figure 10: Generalized Composition of Constraints

6.3 Formal Semantics

The definition of a formal semantics for this notation is outside the scope of this paper. However we indicate here some directions of research and related work. *Causality automata*, which are comparable to our causality relations, have been discussed in [4, 5]. Another causality model, in which occurrences (positive conditions) and non-occurrence (negative conditions) are grouped in separate conditions that have to be true and false respectively in order to enable events, is presented in [6]. None of these paper however treat event attributes formally. In [12] a lot of useful ideas on how to formally represent (real) concurrent behaviour are presented, and some preliminary discussions on the interpretation and representation of event attributes are found.

7 Design Applications

This section briefly discusses the application of our design model on the design and implementation of distributed systems. A forthcoming paper ([15]) will address the application of these concepts in the scope of ODP (Open Distributed Systems).

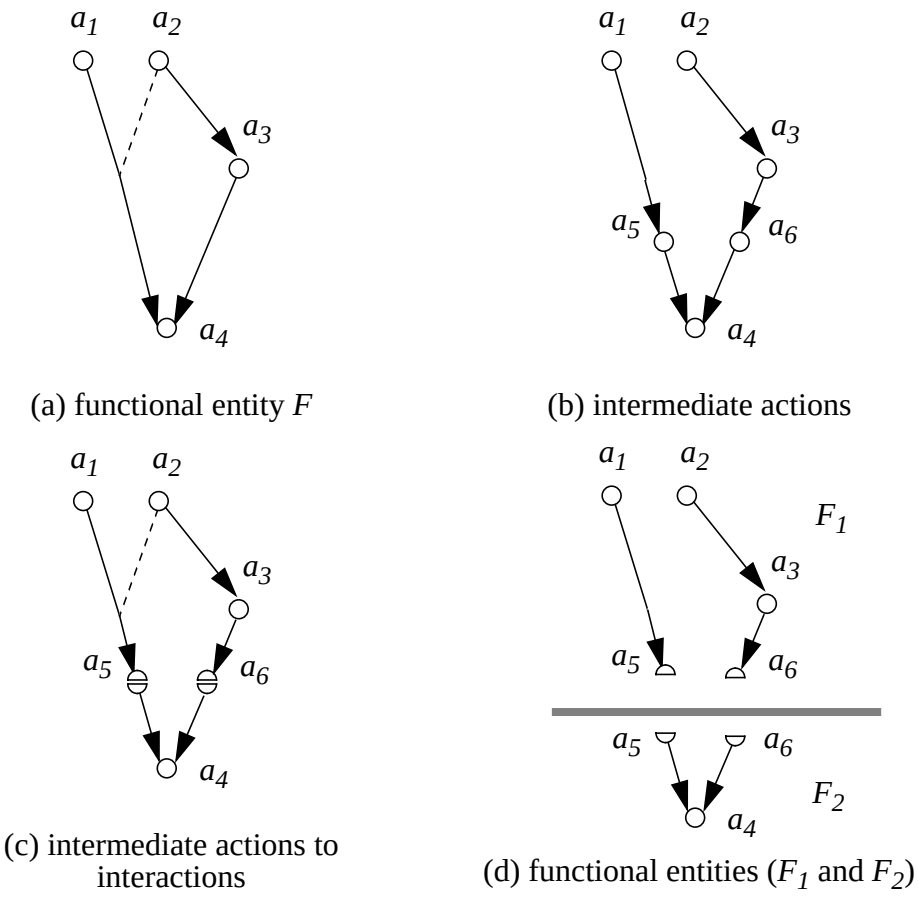
We identify and illustrate below some possible design transformations, based on the manipulation of aspects of the architectural concepts discussed in the previous sections.

- *actions to interactions*: this design transformation consists of transforming actions into interactions, aiming at defining the behaviour of interacting functional entities. Conditions and constraints of the original actions are distributed over the different functional entities. This design transformation can be a step in the decomposition of a functional entity into multiple functional entities.

- *event refinement*: events are replaced by compositions of events, while the functionality, established values and causality relations of the original events are preserved. This criterion shall be used to define implementation notions. Event refinement can be performed in order to (i) decompose location attributes into sub-locations, (ii) decompose the representation of values of information, e.g. to distribute the decomposed values over different events, or (iii) introduce intermediate events for the purpose of decomposition of functional entities.

- *making states explicit*: the behaviour of some functional entity, probably containing independent events, can be explicitly made interleaved. This can be useful in later phases of design, when functional entities which operate in a single computer system or processor are identified. The behaviour of these functional entities can be at this point represented as finite state automata, e.g. for the purpose of software implementation.

Figure 11 illustrates the use of some design transformations mentioned above to decompose a functional entity into two functional entities. The example in Figure 11 shows that once we have a (monolithic) behaviour (a), we can introduce intermediate actions that do not disturb the original causality relations (b), transform these newly introduced actions into interactions (c), and assign the decomposed causality relations to the newly generated functional entities (d).



(a) functional entity F

(b) intermediate actions

(c) intermediate actions to interactions

(d) functional entities (F_1 and F_2)

Figure 11: Example of Functional Entity Decomposition

8 Conclusions

This paper identified some architectural concepts for distributed systems design which deserve formal support. We have shown the importance of architectural concepts in the establishment of a design culture, and we have concluded that language development should follow a careful choice and consideration of the architectural concepts.

In the light of the architectural concepts we have introduced (actions, interactions, causality relations and behaviours), we have given some indications on how designs can be structured and manipulated during the design process, so that design decisions can be incorporated.

Some novel concepts of this paper seem to be the disjunction of conditions, the combination of occurrence and non-occurrence of events in a single causality relation statement, the explicit representation of actions and interactions in a single design model and the entry/exit structuring mechanisms. This framework also allows the combination of timing and performance modeling with (functional) behaviour definitions. The relationships between traditional performance modeling and our design model should be further investigated.

We have refrained in this paper from discussing the specific ways behaviours may share interactions. Our view is that behaviours should be able to explicitly define the characteristics of the interactions they wish to participate in, for example based on the location attributes of these interactions, or based on specific values of information. This would avoid the need for void processes just for synchronization when using constraint-oriented style to specify complex behaviours in LOTOS, which has been reported in [8]. This will be subject of further work.

Further work should be done on the development of a (formal) language to support these design concepts, and on the development of implementation notions to support design transformations.

Acknowledgements

We would like to thank Dick Quartel for his useful comments on this paper. Some of the ideas presented here have been improved during discussions within the TIOS Architecture Group.

References

[1] A. Azcorra. *Formal Modeling of Synchronous Systems*. PhD thesis, Universidad Politecnica de Madrid, Spain, November 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] J-P. Courtiat and R. J. Coelho da Costa. A LOTOS based Calculus with True Concurrency Semantics. In K. Parker and G. Rose, editors, *Formal Description Techniques, IV.* North-Holland, 1992.

[4] J. Gunawardena. Causal Automata. *Theoretical Computer Science*, 101:265-288, 1992.

[5] J. Gunawardena. Causal Automata I : Confluence AND,OR Casuality. In M.Z. Kwiatkowska, M.W. Shields, and R.M. Thomas, editors, *Semantics for Concurrency, Leicester 1990*, pages 137-156, Great-Britain, 1990. Springer-Verlag.

[6] J. Gunawardena. Gemeotric Logic, Causality and Event Structures. In J.C.M. Baeten and J.F. 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.

[7] P. W. King. Formalization of Protocol Engineering Concepts. *IEEE Transactions on Computers*, 40(4):387-403, April 1991.

[8] H. Kremer, J. van de Lagemaat, A. Rennoch and G. Scollo. Protocol design using LOTOS: A critical synthesis of a standardization experience. In M. Diaz and R. Groz, editors, *Formal Description Techniques, IV.* North-Holland, 1993.

[9] L. Lamport. A simple approach to specifying concurrent systems. *Communications*

of the ACM, 32(1):32-45, January 1989.

[10] R. Langerak. Even structures for design and transformation using LOTOS. In K. Parker and G. Rose, editors, *Formal Description Techniques, IV.* North-Holland, 1992.

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

[12] V. Pratt. Modeling Concurrency with Partial Orders. *International Journal of Parallel Programming*, 15(1):33-71, 1986.

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

[14] M. van Sinderen, L. Ferreira Pires and C. A. Vissers. Protocol Design and Implementation using Formal Methods. *The Computer Journal*, October 1992.

[15] M. van Sinderen, C. A. Vissers and L. Ferreira Pires. Design Concepts for Open Distributed Systems. *To appear in: International Conference on Open Distributed Processing, ICODP'93*.

[16] C. A. Vissers. FDTs for Open Distributed Systems, a Retrospective and a Prospective View. In *Protocol Specification, Testing and Verification X*, 1990.

[17] 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.

[18] 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.

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