

# Advanced Design Concepts for Open Distributed Systems Development

Luís Ferreira Pires<sup>b</sup>, Marten van Sinderen<sup>b</sup>, Chris A. Vissers<sup>a, b</sup>

(a) Telematics Research Centre, PO Box 217, 7500 AE Enschede, the Netherlands

(b) University of Twente, PO Box 217, 7500 AE Enschede, the Netherlands

## Abstract

*Experience with the engineering of large scale open distributed systems has shown that their design should be specified at several well defined levels of abstractions, where each level aims at satisfying specific user, architectural and implementation purposes. Therefore designers should dispose of a comprehensive design methodology, which allows them to conceive a specification at a certain abstraction level and transform this specification into a conforming specification at a lower abstraction level. The collection of these transformations should abridge the total design trajectory from initial user requirements to final implementation. This paper presents and discusses some advanced design concepts that provide a basis for such a design methodology.*

## 1 Introduction

In the recent past several projects have been launched in the framework of the European research programs ESPRIT, RACE and Telematics, aiming at developing large scale open distributed telematics and telecommunications systems. Experience with the development of such systems has shown that users, system architects and system implementors have quite specific interests in different behavioural definitions of these systems. In order to satisfy these interests, systems need to be defined at various specific abstraction levels along the design trajectory.

The following abstraction levels appear to have a particular relevance:

1. The definition of the common behaviour of the system and the user environment in which the system will be embedded. This design allows to express the user requirements at a high level of abstraction without being forced to decide how the responsibility for these requirements should be distributed over the system and the user(s). It forms the basis on which agreements between users and architects can be made on what new functions should be introduced in the user environment.

2. The definition of the system as a single functional entity, i.e. in terms of a *service provider*. This requires that the behaviour as defined in step 1 is distributed over the system and its user(s). It allows the architect to express the design of the service provider only in terms of what functions are provided by the system, without immediately being forced to distribute these functions over the parts that ultimately will constitute the service provider. This definition, together with the definition of the real interfaces as introduced in item 4 below, provides also the most suitable basis for producing user manuals.
3. The definition of the service provider in terms of a set of cooperating parts. This requires that the functions of the service provider are decomposed and distributed over the parts that constitute its internal structure. It allows the architect to define the functions of each individual part separately and without being forced to make decisions on how these functions are implemented. This definition forms a natural basis to identify and define generic (infra-)structural system parts.
4. The implementation of the abstract interfaces between the system parts by more concrete interfaces. This allows the implementor to select appropriate real interfaces as determined by engineering and user requirements.
5. The definition of the implementation structure that replaces the relationships between the real interfaces of each individual part. To achieve this definition, the design steps 3 and 4, but applied to an individual part, can be repeated iteratively and in various orders.

A comprehensive design methodology is needed to develop these designs at various abstraction levels. It should provide pragmatic guidance to the system designer to develop the system starting at a high level of abstraction with the user requirements and finishing at a low level of abstraction with the implementation specification ([4]). The basis for such a methodology should be formed by a *comprehensive design model*, consisting of *basic design concepts* and their *combination rules* ([3]), covering all relevant aspects of system design. This model should also provide

the basis for a design language in which designs can be expressed.

In practice it appears that basic design concepts and their combination rules are poorly understood. Moreover, there exists no common agreement on how a comprehensive design model should look like. Consequently most design models and design methodologies are too restrictive and cover only a limited part of the design trajectory.

These observations motivated our work in the development of a set of basic design concepts, some of which we present and illustrate below.

## 2 Design methodology

This section analyses the first two abstraction levels presented in section 1 in terms of their consequences for the definition of basic design concepts. These same basic design concepts can be used to develop designs at the subsequent abstraction levels.

The definition of the system and its environment *together* determine a *common behaviour*: what interactions in what order and with what value attributes can actually be 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 system. 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 while abstracting from the many different ways in which the system and its environment may contribute to these results. Therefore interaction systems are concerned with *integrated interactions*, which we henceforth call *actions*.

Our design methodology thus starts with the definition of an interaction system requiring as a basic design concept the notion of action.

Complex systems are better specified in a *structured* way in order to be conceived, understood, manipulated and maintained. Structuring behaviours as a composition of sub-behaviours, in which constraints on the original behaviour are separately defined, seems in particular to be a promising approach ([5]). Applying this technique to the definition of an interaction system behaviour requires that certain actions are decomposed into interactions that are assigned to constraints. This design structure has the benefit of actually *preparing* the decomposition of a functional entity (e.g. an interaction system) into parts (e.g. a system and its environment).

In the subsequent design step the interaction system is decomposed into a system and its environment, through the assignment of constraints to these parts. Interactions generated by defining constraints, are allocated to and distributed over these cooperating parts.

We illustrate these design structuring techniques with an example of an arbitrary interaction system in Figure 1. In this figure actions and interactions are indicated with circles and circle segments, respectively. This notation will be used consistently throughout the text.

Figure 1 shows three design structures in the development of this arbitrary interaction system, relating them to the first two abstraction levels discussed above. In this figure we illustrate the design choice of assigning actions and interactions and their relationships to the system and to its environment.

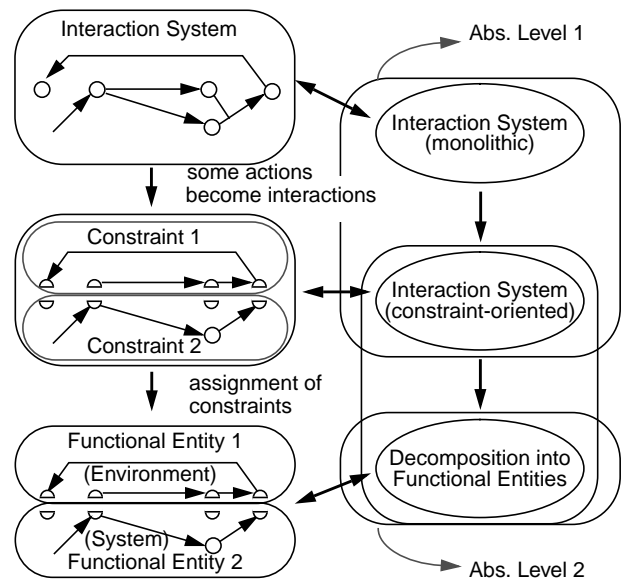


Figure 1: Three design structures

## 3 Basic architectural concepts

The concepts which allow the different forms of design structuring introduced in section 2 are elaborated in this section.

### 3.1 Monolithic behaviour definition

Actions, interactions and their relationships form the basic concepts that allow the definition of behaviour. For brevity reasons we use the term action to refer to both actions and interactions.

**Action attributes.** We identify, based on our design experience, the following attributes of actions:

- *Location attribute*: location at which the action occurs;
- *Time attribute*: moment or period of time when an action occurs or can occur;
- *Values of information*: values of information established in the action;

- *Functionality attribute*: set of values of information passed to this action by previous actions and that may be referred to by successive actions;
- *Probability attribute*: probability that an action occurs according to its definition, once this action is *enabled*. This attribute is not further discussed in this paper for brevity reasons.

We assume that we can always distinguish, and thus uniquely identify, all actions in behaviour definitions. In the examples below we use unique *action identifiers* to indicate their occurrence or non-occurrence.

**Behaviour definitions.** The behaviour of a functional entity (an interaction system, a system's environment, a system or a system part) contains the following elements:

- *Initial actions*: actions which occur independently of the occurrence of other actions of the defined behaviour, possibly with initial sets of attributes. These actions may occur spontaneously when the behaviour is instantiated, or they may be enabled through *entries* by other behaviours to which the behaviour is linked;
- *Causality context* (of an action): defines the role of an action in a behaviour;
- *Exit condition*: a behaviour is said to *exit* if it enables initial actions of another behaviour via an entry. This enabled behaviour is then allowed to start, allowing to make references to the attributes of the exit condition.
- *Termination condition*: an action is said to terminate a behaviour if no more actions or other behaviours are enabled by it.

The conditions for the occurrence of an action of a behaviour are defined in a *causality relation* between the other actions of this behaviour and this action. We say that a causality relation defines the conditions which enable and constrain the occurrence of an action. By using causality between actions to represent behaviour we avoid the drawbacks of arbitrary interleaving semantics often found in process algebra based formalisms ([2], [6]). Some theory on causality based behaviour definitions can be found in [1].

**Basic causality relations.** We define the causality relation  $a_1 \rightarrow a_2$  as: the occurrence of action  $a_1$  is a condition for the occurrence of action  $a_2$ ; attributes of  $a_2$  may refer to attributes of  $a_1$ .

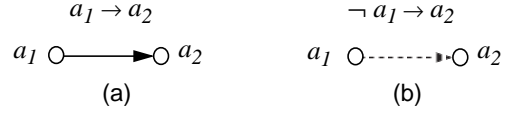
The causality relation above says nothing about the conditions for the occurrence of  $a_1$ . These are defined in the causality relation of  $a_1$ , which is not part of the causality relation of  $a_2$ . This implies that the possible occurrence of  $a_2$  can only be determined after evaluation of all causality relations that lead to the occurrence of  $a_1$ .

The occurrence of an action may also depend on the non-occurrence of another action, characterizing a type of causality, called *conflict*.

We define the causality relation  $\neg a_1 \rightarrow a_2$  as: the non-occurrence of action  $a_1$  is a condition for the occurrence of action  $a_2$ ; attributes of  $a_2$  can not refer to attributes of  $a_1$ .

Since this causality relation has to be evaluated at the moment the implementation decides whether  $a_2$  will take place or not, we consider that  $a_1$  does not happen before or at the time of  $a_2$ . If  $a_1$  happens before  $a_2$ , then  $a_2$  will never happen, but  $a_1$  may happen after  $a_2$ .

Figure 2 depicts these two basic causality relations.



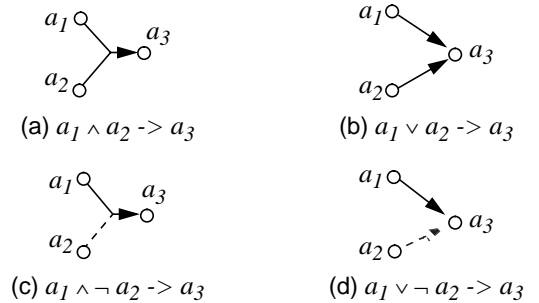
**Figure 2: Basic causality relations**

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

**Logical combinations of conditions.** Causality relations of arbitrary complexity are defined by logical combinations of occurrence and non-occurrence of actions using *and* ( $\wedge$ ) and *or* ( $\vee$ ) logical operators. We consider some examples below:

- *Conjunction of Occurrences*:  $a_1 \wedge a_2 \rightarrow a_3$ . Occurrences of both  $a_1$  and  $a_2$  are a condition for the occurrence of  $a_3$ .
- *Disjunction of Occurrences*:  $a_1 \vee a_2 \rightarrow a_3$ . Occurrence of  $a_1$  or  $a_2$  is a condition for the occurrence of  $a_3$ . Notice that  $a_1$  and  $a_2$  may both happen, but 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 causes  $a_3$ ; attributes of  $a_3$  may only refer to attributes of this causing action.
- *Conjunction of Occurrence and Non-occurrence*:  $a_1 \wedge \neg a_2 \rightarrow a_3$ . The occurrence of  $a_1$  and the non-occurrence of  $a_2$  are both conditions for the occurrence of  $a_3$ .
- *Disjunction of Occurrence and Non-occurrence*:  $a_1 \vee \neg a_2 \rightarrow a_3$ . The occurrence of  $a_1$  or the non-occurrence of  $a_2$  are conditions for the occurrence of  $a_3$ .

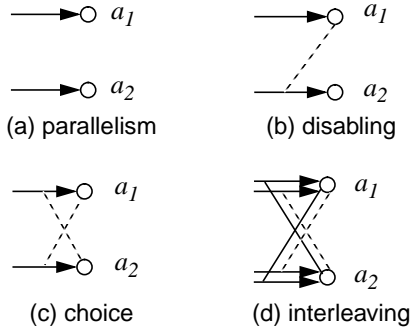
Figure 3 depicts these causality relations.



**Figure 3: Some causality relations**

Consistently with our interpretation of the basic causality relations, all the prescribed conditions have to be fulfilled at the moment when a resulting action is scheduled to occur.

Figure 2(a) depicts sequential composition of actions. Real parallelism, disabling, choice and arbitrary interleaving, which are behaviour patterns that appear very often in distributed systems, can also be represented in our notation, as it is shown in Figure 4.



**Figure 4: Some behaviour patterns**

These behaviour compositions can be used to represent behaviours of arbitrary complexity, yielding a quite powerful structuring facility.

**Attributes in causality relations.** Action attributes play distinct roles in causality relations, depending on whether an action is a condition or a resulting action:

- *Attributes in conditions:* can be used to define prerequisites for attribute values of actions in a condition, with which the resulting action will be enabled;
- *Attributes in resulting actions:* can be used to define the allowed attribute values of the resulting action.

We illustrate these with the following example:

$$a_1 (v_1: \text{Nat}) [v_1 > 10] \rightarrow a_2 (v_2: \text{Nat}) [v_2 < v_1 + 3]$$

This causality relation states that only in case  $a_1$  happens with a value  $v_1$  greater than 10,  $a_2$  is allowed to happen. If  $a_2$  happens its value has to be smaller than  $v_1 + 3$ . As long as  $a_1$  does not happen or if it happens with  $v_1 \leq 10$ , the condition for  $a_2$  is false, and  $a_2$  is not allowed to happen.

Similar discussions also apply to the other action attributes, which are not elaborated in this paper.

**Finite monolithic behaviour.** A finite behaviour can be represented by a set of causality relations, one relation per action of this behaviour. Consider the following example:

$$B := \{ \text{start} \rightarrow a_0, \text{start} \wedge \neg a_0 \rightarrow a_1, \\ a_1 \vee a_0 \rightarrow a_2, a_1 \wedge \neg a_2 \rightarrow a_3, a_2 \rightarrow a_4 \}$$

$\text{start} \rightarrow a_0$  states that  $a_0$  is enabled from the beginning of the behaviour.  $\text{start} \wedge \neg a_0 \rightarrow a_1$  implies that from the beginning of the behaviour and while  $a_0$  does not happen  $a_1$  is allowed to happen. Hence  $a_0$  and  $a_1$  are initial actions of  $B$ .  $B$  determines the causality context of all its actions. For example,  $a_2$  is the resulting action in  $a_1 \rightarrow a_2$ , and appears as

condition in  $a_1 \wedge \neg a_2 \rightarrow a_3$ , and  $a_2 \rightarrow a_4$ . Together they completely define the role of  $a_2$  in  $B$ .

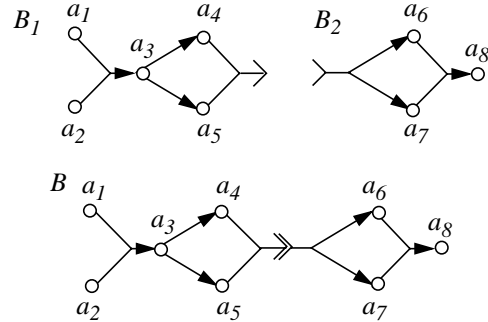
### 3.2 Structured behaviour definition

This section presents some mechanisms to represent repetitive and infinite behaviour and to structure designs.

**Causality-oriented behaviour composition.** In this type of behaviour composition, conditions inside an instance of behaviour enable actions of another instance of behaviour, in a similar way as conditions on actions enable resulting actions in causality relations. Entries and exits are used in our design model as (syntactical) mechanisms to represent this type of behaviour composition.

Suppose  $B_1$  and  $B_2$  are behaviours and that  $B_1$  has one exit and  $B_2$  has one entry. A causality-oriented composition of  $B_1$  and  $B_2$  can be defined by combining the exit of  $B_1$  and the entry of  $B_2$ , such that the conditions of the exit of  $B_1$  become conditions of the actions related with the entry of  $B_2$ . The resulting behaviour can be obtained by short-circuiting the exit of  $B_1$  and the entry of  $B_2$ .

Figure 5 depicts an example, in which the exit conditions of  $B_1$  have turned into conditions of actions  $a_6$  and  $a_7$  of the entry of  $B_2$ .

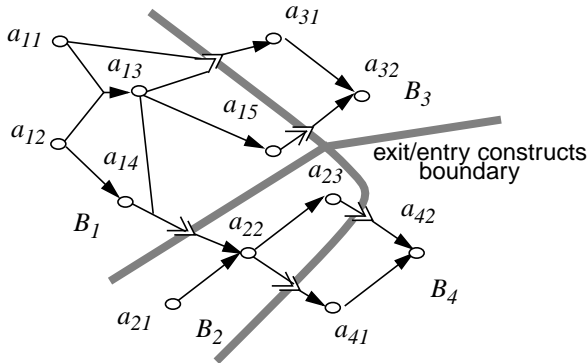


**Figure 5: Exit/entry construct**

This type of composition can be generalized to more than one exit condition or entry points or both per behaviour. The consequence is that exits and entries must be identified, in order to allow their unambiguous combinations.

Figure 6 illustrates the effect of this composition with arbitrary 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 causality relations. This mechanism allows us to structure a monolithic behaviour in sub-behaviours, such that compositions of behaviour definitions can be created.

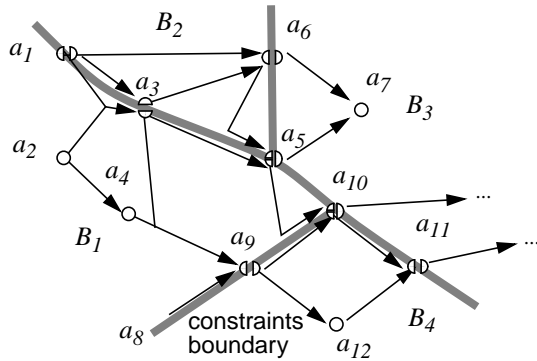
**Constraint-oriented behaviour composition.** In this type of behaviour composition, a behaviour is represented as a conjunction of constraints on actions, which are described in separate (sub-)behaviours.



**Figure 6: Causality-oriented composition**

The consideration of this approach in our design model forces us to represent some actions in a distributed form, i.e. as interactions. The 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.

Figure 7 illustrates the effect of composition of constraints with four arbitrary behaviours  $B_1$ ,  $B_2$ ,  $B_3$  and  $B_4$ .



**Figure 7: Constraint-oriented composition**

## 4 Design steps

This section shows the application of our design model to accomplish design objectives in design steps.

### 4.1 Refinement types

We consider the following refinement types with respect to an arbitrary behaviour:

- *Behaviour refinement*: the objective of this refinement type is to refine the behaviour structure, such that sub-behaviours can be identified and assigned to component functional entities of the interaction system. Behaviour refinement consists of *introducing actions* in the behaviour, and modifying some of the original causality relations. The implementation notion must be such that the original causality relations are still valid in the new

behaviour, if we abstract from the actions introduced in the design step.

- *Interface refinement*: the objective of this refinement type is to refine some action structure at a location, such that the abstract interface associated to this location can be replaced by a more concrete interface. Interface refinement consists of replacing actions of the behaviour by compositions of actions. The implementation notion must be such that the causality contexts of the composition of actions correspond to the causality contexts of the original action structure.

The causality context of an action structure defines its role in a behaviour. This can be represented by the causality relations between actions inside and outside the action structure.

### 4.2 Example

We illustrate the refinement types of section 4.1 using the example of a word transfer service. The behaviour of this service is such that an action in which a 16-bit word of data is established (*req*), followed by a corresponding action for the acceptance of this data (*ind*). This is specified as follows:

$$\begin{aligned}
 &WTS \text{ (* Word Transfer Service *)} := \\
 &\{ \text{start} \rightarrow \text{req}(w_1:\text{word}), \\
 &\quad \text{req}(w_1:\text{word}) \rightarrow \text{ind}(w_2:\text{word})[w_2=w_1] \}
 \end{aligned}$$

Figure 8 depicts the above behaviour.



**Figure 8: Word transfer service**

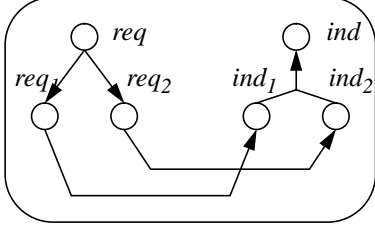
We investigate the implementation of this service given the availability of a (lower level) octet transfer service. Two alternatives for the octet transfer service will be considered, viz. (i) two independent channels for octet transfer, and (ii) a single channel for octet transfer which preserves the order of the octets.

**Use of independent channels.** In our implementation using alternative (i), *req* causes two actions (*req*<sub>1</sub> and *req*<sub>2</sub>) independent of each other, each one establishing an octet. *req*<sub>1</sub> (*req*<sub>2</sub>) causes a corresponding action *ind*<sub>1</sub> (*ind*<sub>2</sub>) for the acceptance of the same octet. Both *ind*<sub>1</sub> and *ind*<sub>2</sub> must occur in order to cause *ind*. We assume that we have assigned the transfer of the first octet to a specific channel and the transfer of the second octet to other one, allowing the word to be assembled by *ind*. The following specification represents this behaviour:

$$\begin{aligned}
 &WTS' := \\
 &\{ \text{start} \rightarrow \text{req}(w_1:\text{word}), \\
 &\quad \text{req}(w_1:\text{word}) \rightarrow \text{req}_1(o_1:\text{octet})[o_1=\text{first}(w_1)], \\
 &\quad \text{req}(w_1:\text{word}) \rightarrow
 \end{aligned}$$

$$\begin{aligned}
& req_2(o_1:octet)[o_2=second(w_1)], \\
& req_1(o_1:octet) \rightarrow ind_1(o_3:octet)[o_3=o_1], \\
& req_2(o_2:octet) \rightarrow ind_2(o_4:octet)[o_4=o_2], \\
& ind_1(o_3:octet) \wedge ind_2(o_4:octet) \rightarrow \\
& \quad ind(w_2:word)[w_2=conc(o_3,o_4)] \}
\end{aligned}$$

Figure 9 depicts this behaviour.



**Figure 9: Use of two independent channels (1)**

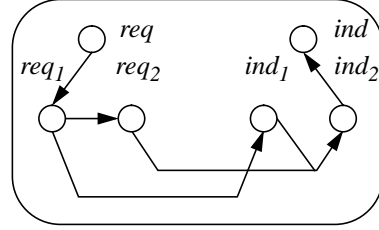
The original causality relation between *req* and *ind* is still valid, since *ind* can only happen if *req* has happened. Furthermore, *ind* does not depend on any other action besides *req* and the actions introduced in this design step. We assume that the attributes of the original *req* and *ind* and their values are preserved, which should follow from the definition of data type operations *first*, *second* and *conc*, i.e. the requirement  $conc(first(w), second(w)) = w$  should be satisfied. These are intuitive indications that the given specification is a correct implementation of the original one.

**Use of one FIFO channel.** In our implementation using alternative (ii), the original *req* causes an action *req<sub>1</sub>* in which the first octet of the word is established, keeping the original word in the functionality attribute. *req<sub>1</sub>* causes an action *req<sub>2</sub>*, in which the second octet is established. *req<sub>1</sub>* (*req<sub>2</sub>*) causes a corresponding action *ind<sub>1</sub>* (*ind<sub>2</sub>*) for the acceptance of the first (second) octet. In addition, *ind<sub>1</sub>* is a condition for *ind<sub>2</sub>*, and the first octet is kept in the functionality of *ind<sub>2</sub>*. *ind* is finally caused by *ind<sub>2</sub>*. The following specification represents this behaviour:

$$\begin{aligned}
WTS'' := & \\
\{ & start \rightarrow req(w_1:word), \\
& req(w_1:word) \rightarrow \\
& \quad req_1(o_1:octet, w_1:word)[o_1=first(w_1)], \\
& req_1(o_1:octet, w_1:word) \rightarrow \\
& \quad req_2(o_2:octet) [o_2=second(w_1)], \\
& req_1(o_1:octet) \rightarrow ind_1(o_3:octet)[o_3=o_1], \\
& req_2(o_2:octet) \wedge ind_1(o_3:octet) \rightarrow \\
& \quad ind_2(o_4:octet, o_3:octet) \rightarrow \\
& \quad \quad ind(w_2:word)[w_2=conc(o_3,o_4)] \}
\end{aligned}$$

Figure 10 depicts this behaviour.

Similarly to alternative (i), the original causality relation between *req* and *ind* is still valid after this design step. Again, the attributes of the original *req* and *ind* and their values are preserved, giving an intuitive indication that this specification is a correct implementation of the original one.

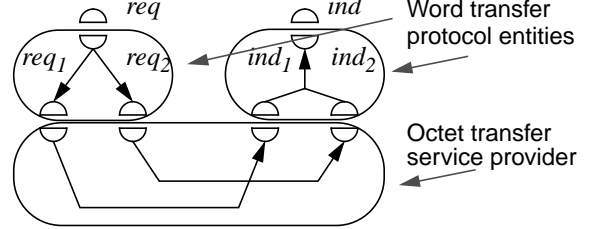


**Figure 10: Use of one FIFO channel (1)**

**Assignment to functional entities.** Actions can be decomposed into interactions, which are then assigned to functional entities.

Considering the behaviour of Figure 9, we assign the causality relations between *req<sub>1</sub>* (*req<sub>2</sub>*) and *ind<sub>1</sub>* (*ind<sub>2</sub>*) to the octet transfer service provider, while the other relationships are assigned to the word transfer protocol entities.

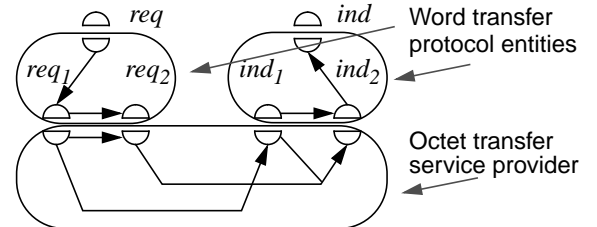
Figure 11 depicts these assignments.



**Figure 11: Use of two independent channels (2)**

Considering the behaviour of Figure 10, we assign the causality relations between *req<sub>1</sub>* (*req<sub>2</sub>*) and *ind<sub>1</sub>* (*ind<sub>2</sub>*) to the octet transfer service provider. The ordering relation between *req<sub>1</sub>* and *req<sub>2</sub>* is distributed over the service provider and the sending protocol entity, since this protocol entity imposes the establishment of the first (second) octet in *req<sub>1</sub>* (*req<sub>2</sub>*). Furthermore, the octet transfer service provider can handle one octet at the time, imposing that *req<sub>1</sub>* and *req<sub>2</sub>* should happen in sequence. This order is preserved by the service provider during transfer, which is represented by the causality relation between *ind<sub>1</sub>* and *ind<sub>2</sub>* in the service provider. In the receiving protocol entity a causality relation between *ind<sub>1</sub>* and *ind<sub>2</sub>* is used to pass the first octet to *ind*.

Figure 12 depicts these assignments.



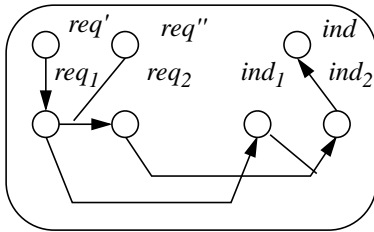
**Figure 12: Use of one FIFO channel (2)**

In both cases above  $req$  and  $ind$  are distributed over the protocol entities and the word transfer service users.

The correctness of these decompositions can be intuitively assessed by comparing the resulting causality relations with the original ones.

**Interface refinement.** We consider the refinement of the abstract interface associated to  $req$  in the behaviour of Figure 10. Two alternatives are investigated: replacement of  $req$  by two parallel actions and by two sequential actions.

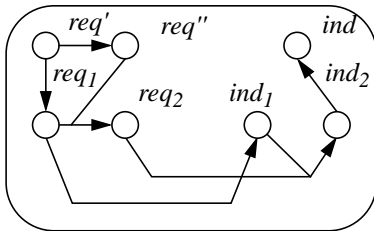
Figure 13 depicts a possible replacement of  $req$  by two parallel actions.



**Figure 13: Two parallel actions**

The replacement of  $req$  by  $req'$  and  $req''$  depicted in Figure 13 does not conform to our original implementation notion for interface refinement. However we have manipulated the causality relations such that the occurrence of  $req_2$  depends directly on the occurrence of both  $req''$  and  $req_1$ , and indirectly of  $req'$ , which corresponds to the original condition ( $req_2$  occurs after  $req$  and  $req_1$ ). This has been possible since  $req_1$  does not need to be dependent of  $req''$ .

Figure 14 depicts the replacement of  $req$  by two sequential actions.



**Figure 14: Two sequential actions**

Although  $req'$  and  $req''$  are already in the desired order (first and second octet), this order has to be maintained by  $req_1$  and  $req_2$ . Similarly to the former example, we could have made  $req''$  a condition for  $req_1$ , but this would be unnecessarily restrictive since  $req_1$  does not need to be dependent of  $req''$ .

The examples show that interface refinement at a certain location, in this case the location where  $req$  occurs, does not influence actions at other locations. Therefore the designer is free to choose specific interface refinements, without running the risk of jeopardizing the whole design.

## 5 Conclusions

Our design methodology for open distributed systems development shows five major abstraction levels:

1. common behaviour of a system and its environment;
2. the system as a functional entity;
3. the system as a composition of abstract functional parts;
4. the introduction of real interfaces between parts;
5. the implementation of parts.

These abstraction levels, and the transformations to move from one level to the other, are supported by repeated application of the following design concepts:

- behaviour defined as a set of causality relations;
- behaviour defined as a causality-oriented composition of sub-behaviours;
- behaviour structured as a set of constraints;
- the introduction of actions within causality relations;
- the assignment of behaviours to functional entities;
- the refinement of an action into multiple actions.

These concepts allow the representation of sequential composition, parallelism, arbitrary interleaving, choice, disabling, enabling the representation of behaviours of arbitrary complexity.

An example that illustrates the application of these concepts to some design steps has been presented in this paper.

We believe that the design methodology presented in this paper, together with its design model, are generally applicable to most applications in the area of distributed systems design, such as Open Systems Interconnection, Open Distributed Processing, and various telematics and telecommunications systems.

## References

- [1] J. Gunawardena. Causal automata. *Theoretical Computer Science*, 101:265–288, 1992.
- [2] V. Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.
- [3] E. Reichtin. The art of systems architecting. *IEEE Spectrum*, pages 66–69, October 1992.
- [4] 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.
- [5] 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.
- [6] C. A. Vissers, M. van Sinderen, and L. Ferreira Pires. What makes industries believe in formal methods. In *Protocol Specification Testing and Verification, XIII*, 1993. to appear.