A Design Model for Open Distributed Processing Systems

Marten *van Sinderen*[1], Luís *Ferreira Pires*[1], Chris A. *Vissers*[1, 2], Joost-Pieter *Katoen*[1]

(1) Tele-Informatics and Open Systems Group, University of Twente
(2) Telematics Research Centre
PO Box 217, 7500 AE Enschede, The Netherlands
e-mail: {sinderen, pires, vissers, katoen}@cs.utwente.nl

Abstract
This paper proposes design concepts that allow the conception, understanding and development of complex technical structures for open distributed systems. The proposed concepts are related to, and partially motivated by, the present work on Open Distributed Processing (ODP). As opposed to the current ODP approach, the concepts are aimed at supporting a design trajectory with several, related abstraction levels. Simple examples are used to illustrate the proposed concepts.

Keywords: ODP systems; Design methodologies; Design concepts; Entity domain; Behaviour domain; Structuring techniques.

1 Introduction

The growing interest in distributed system applications has motivated the standardization work on Open Distributed Processing (ODP) by ISO/IEC and CCITT ([3], [4]). The main purpose of this standardization work is to allow the support of applications on a heterogeneous collection of systems, permitting these systems to be arbitrarily distributed. Application end-users may have great benefits from standards for open distributed systems, including:

- *manufacturer and vendor independence*: distributed systems can be composed from products offered by different, usually competing, manufacturers. This is the usual interpretation of the term *openness*;
- *crossing organizational boundaries*: distributed systems may be spread across a number of autonomous management or control authorities. This enables the sharing and integration of resources and applications beyond the boundaries of the local organization; and
- *economy of scale*: standards development is a common effort in which many manufacturers may participate. The adoption of standards by many manufacturers may increase the production of systems derived from these standards and consequently may decrease their price.

The design and implementation of a distributed system is a complex undertaking, and so is the development of standards for open distributed systems. Even more because the potential benefit of manufacturer independence requires that standards are precise and unambiguous

prescriptions for implementations, formulated at a suitable abstraction level. They should be a reference for the implementation of conformant systems, while leaving maximum implementation freedom to the individual manufacturers.

In order to meet these requirements and to be able to produce standards fast enough to keep up with the end-users' needs and expectations, it is necessary to have an effective environment for producing, i.e. designing, standards. Such an environment can be called a design culture ([20]). An important example of such a design culture is the Open Systems Interconnection (OSI) environment, which led to the OSI Reference Model (OSI-RM) and associated service and protocol standards.

A design culture for producing standards must be commonly agreed upon, as opposed to design cultures adopted by industrial companies, whose competitiveness in part depends on the secrecy of their design culture. An important aspect of standardization work is therefore the establishment of a design culture which is appropriate to the application area at hand. Naturally, this work should precede as much as possible the production of the detailed technical standards. The OSI community, for example, initially focused on the development of the OSI-RM, which defines the key concepts necessary to define OSI services and protocols[1].

The first standard to be produced by the ODP community is a Basic Reference Model for ODP (ODP-RM) ([8], [19]). The purpose of this standard is to provide a coordinating framework for the elaboration of standards for ODP systems. We believe that such a framework should be defined as a common design environment, appropriate to the ODP application area, like the OSI-RM has been defined for the OSI application area. The ODP-RM should however be more comprehensive with respect to a design environment than the OSI-RM could be at the time of its conception. Indeed, it is possible to recognize the following distinctive elements of a design culture in the current version of the ODP-RM:

- framework of abstractions (Part 1 and Part 3 of [7]);
- design model (Part 2 and Part 3 of [7]); and
- architectural semantics (Part 4 of [7]).

This paper discusses a number of demands for an effective design culture, based on the OSI experience and on the experience with the development and use of Formal Description Techniques (FDTs) in the context of OSI. We conclude that the current version of the ODP-RM does not fully satisfy these demands, which implies that this reference model has to be improved. This paper also presents a design model consisting of a set of elementary design concepts that can be used as general purpose building bricks for the composition of designs and explicitly acknowledging the demands for an effective design culture. This design model is more general purpose than the one described in the ODP-RM and should be considered as complementary, rather than opposite to the ODP design model.

The presentation of our design model in this paper is carried out at a conceptual level, appealing to the designers' intuition. In this way we avoid any biasing with respect to specific FDTs. This paper also presents a possible basis for a formal semantics of the design model. Furthermore we evaluate the suitability of this design model to support a design methodology for ODP systems. An example is used to illustrate some aspects of this model.

The remaining of this paper is organized as follows: Section 2 presents some general demands for a design environment, and briefly discusses the support to these demands already provided by the ODP standardization, Section 3 presents the entity domain and the behaviour

1. Other elements of the OSI design culture were developed later, e.g. service conventions, formal description techniques, and architectural semantics.

domain, from which the elements of our design model are defined, Section 4 introduces five abstraction levels at which ODP systems should be considered, Section 5 presents a collection of concepts that allow for behaviour definitions, Section 6 introduces behaviour structuring mechanisms, Section 7 discusses the application of entity and behaviour domains in a framework for design and implementation, Section 8 illustrates our design model with an example of the design of a simplified system to support multi-media (audio and video) information exchange and Section 9 presents some concluding remarks.

2 Demands for a Design Environment

A number of demands for an effective design environment for ODP systems, primarily based on our OSI and FDT experience, are presented below in terms of rules. The extent to which these demands have been satisfied by the current version of the ODP-RM is indicated.

Rule 1: *Design complexity calls for the use of a design methodology*

There are many ways to arrive at the same design, and there are many alternative compositions of a design that reflect the same functionality. Yet, in both cases, one option is often preferred above the others. Therefore, especially if a design (process) is complex, designers should have a set of judgement criteria and procedures at their disposal which guides them in taking well-considered design decisions. These criteria and procedures may be based on subjective value judgements, rational techniques, consensus, or heuristics ([13]).

A set of related criteria and procedures, such that the design process as a whole is covered instead of some isolated parts of it, is called a *design methodology.* A design methodology enables designers to systematically deal with all concerns, requirements and constraints involved in the design of complex systems. It should allow to distinguish, order, and categorize concerns and handle categories of concerns in a step by step fashion. In each step only one category of concerns is dealt with according to some predefined design goal while preserving the design goals achieved in previous steps. By limiting design freedom, design methodologies may speed up the design process, control its quality, and ensure the consistency among designs.

The design gap to be bridged by ODP is very wide, ranging from enterprise requirements to engineering solutions. The involved complexity calls for the adoption of a design methodology. Nevertheless, the ODP work, so far, tries to be methodology-independent. The probable reason for this is that the ODP community comprises many different communities (among others tele-communication, software engineering, and data base communities), each with their own methodology, which they are reluctant to dispose of or to compromise.

Rule 2: *A design methodology is effectively supported by a set of properly related abstraction levels*

Abstractions in system design ignore those characteristics of a system which are irrelevant for a specific purpose. Hence, a set of abstractions can be effectively used as the basis of a design methodology, provided the abstractions are chosen in accordance to the design goals of the methodology. The ordering and the step by step handling of categories of design concerns calls for a set of *related* abstraction *levels*. The relationship between these levels should be such that at each next abstraction level the design goals achieved at previous abstraction levels are preserved. Abstraction levels are then hierarchically related.

The use of abstraction levels is attractive for various reasons. First, it supports a bootstrapping approach to design, i.e. it allows short design gaps between designs at adjacent abstraction levels, and thus enables easier validation of intermediate designs and leads to short repair cycles. Second, it supports an easier mapping of application requirements to technological requirements, since the former can be properly represented at higher abstraction levels and the latter at lower ones. Third, in case formal methods are used, the relation between adjacent abstraction levels can be formalized as an implementation relation, facilitating the development of (semi-)automatic design support tools (e.g. for validation and transformation).

The ODP-RM identifies a set of different views of the system, called *viewpoints*. However, we observe that the ODP viewpoints are not properly related. Although there are some relations and commonalities between the different viewpoints there seems to be no explicit consistency relation between them. These explicit relations are indispensable from a designer's point of view. For example, the requirements in the enterprise viewpoint are visible as environment constraints (quality of service, dependability, and so on) in the computational viewpoint, but it is unclear how the choices made at the computational viewpoint are influenced by the enterprise viewpoint. Another problem is how different viewpoint descriptions can be considered in combination, forming a single reference for implementation.

The establishment of hierarchical relationships between viewpoints is a possible solution to the problems above. Although in principle hierarchical relationships can only be established for certain aspects of viewpoints, according to the definition of viewpoint in the current ODP work, viewpoints can not be considered as related abstraction levels. However a tendency of considering viewpoints as abstraction levels can be observed, such as in recent versions of (the non-prescriptive) Part 1 of [7].

Rule 3: *Abstraction levels should address the common behaviour of a system and its environment, the role of the system in this common behaviour, and the decomposition of this role*

Future users of a system under design are first of all interested in the total behaviour that results from using the system. This behaviour allows the reflection of application requirements with respect to the system at the highest abstraction level.

A proper design concept that supports this abstraction is the concept of *service*. The service concept has a long history in the OSI-RM, although it has often been misinterpreted, and therefore sometimes used ineffectively. A service should correspond to the shared boundary of a system and its environment. The service behaviour defines the common behaviour of a system and its environment in terms of integrated interactions (*service primitives* in OSI terminology). An integrated interaction is defined independent from the possible individual responsibilities of the system and its environment; they should be considered as *actions*, rather than interactions.

Here lies the difference with the concept of *service provider*, another concept that is used in the OSI-RM. A service provider embodies the responsibility of the system in the common behaviour defined by the corresponding service. A service provider can be defined as a single entity of behaviour, in terms of its contributions to *interactions* with the environment. The service provider, therefore, can be considered as a component of the decomposition of the service. The service provider behaviour defines the role of the system in the common behaviour of a system and its environment. Another concept defined in the OSI-RM is that of *protocol*, which defines the internal structure, or decomposition, of a service provider in terms of a composition of protocol entities and a lower level service.

It follows that the concepts of service, service provider, and protocol, support the definition of a system at three subsequent abstraction levels. These levels can be used iteratively in a design methodology.

The way in which the concepts of service, service provider and protocol relate to viewpoints is not explicitly defined. For example in some cases a service-protocol relationship is considered between models of the computational and engineering viewpoints, but such a relationship is an intuitive interpretation, not supported by the ODP-RM.

Rule 4: *A design model must suit the purpose of its application area*

A design model consists of a set of elementary design (or architectural) concepts which can be used as general purpose building bricks for the composition of designs. Obviously, a design model should suit the purpose of the application area at hand. Despite its triviality, this requirement is often compromised by the unconditional adoption of modelling or specification techniques with preconceived limitations.

A common source of problems comes from considering a design and its specification as the same entities, whereas they should be considered as distinct entities. A design is an abstraction of a technical object, as conceived by a designer. A specification is only the representation of a design, albeit the only thing that allows others to look at the design. Hence, in case a specification language has severe limitations in its expressive power, design concepts supported by this specification language are merely approximations of the design concepts appropriate to the application area ([22]).

Design concepts play a central role in a design culture: they determine how designs can be composed, understood and manipulated, and therefore should influence the development of modelling techniques, design methods and specification languages. However it is not easy to determine whether a set of design concepts is appropriate. Often heuristics are involved: design concepts should be appealing to the designer, allow him to conveniently address, at the correct level of abstraction, all design concerns relevant to the application area. In addition, they should observe qualitative architectural principles such as generality, orthogonality and parsimony.

The ODP-RM defines a large number of design concepts, including elementary design concepts (basic modelling concepts in ODP terminology). A subset of these design concepts stem from an object-oriented modelling approach and not primarily from needs of the application area. Consequently, the relationship between these concepts and the elementary design concepts is not always clear. This lack of clarity does not imply that the object-oriented paradigm is unsuitable for ODP; it might be useful to express these concepts. However, the concepts should be motivated from the designer point of view, tailored to the application area.

Rule 5: *A specification language should accommodate the design model*

A design culture should only adopt a specification language if this language allows a straightforward and intuitive representation of the design concepts, and compositions thereof, as defined by the design model. If a language is introduced, however, with little regard of this requirement, one easily runs the risk of only considering the characteristics of systems in the light of the design model imposed by the chosen language. As a result, design concepts become obscured by preconceived language limitations and designers may even be forced to take improper design decisions.

A defined unique mapping between the elementary design concepts of the design model and constructs of a specification language representing these concepts is called *architectural semantics*. The architectural semantics of a specification language allows the interpretation of

a specification in terms of the interpretation of constructs corresponding to elementary design concepts.

Available standard specification languages for open distributed systems show severe limitations in the representation of design concepts. It is then important to acknowledge such limitations and find ways to compensate for them. Enhancements should follow from careful consideration of the design concepts adopted, and not concentrate purely on manipulation of the semantic models of these languages.

A specification language should be applicable at each of the abstraction levels used in a design methodology. In particular, a specification language should permit *description* as well as *prescription*. Observable behaviour for example can be considered as a description, whereas behaviour that is defined in terms of an explicit internal structure can be considered as a prescription. The former is of interest to the future user, the latter to the implementor of the system. In both cases, behaviour can be considered as behaviour that can be interpreted by designers. This changes the concept of observability: designers prescribe the behaviour of a system, making it possible for implementors to use this prescription in order to construct the system. As a result, actions and interactions are to be considered in a single framework, from which designers can make implementation decisions explicit.

3 Entity Domain and Behaviour Domain

In most approaches towards the design of distributed systems one can recognize the existence of the following architectural concepts (see [5], for example):

- *(functional) entity*, which is a logical or physical component of the system
- *action*, which is an abstraction of an activity performed by an entity
- *interaction*, which is an action shared by two or more entities
- *action point*, which is the logical location for the execution of an action
- *interaction point*, which is the logical location for the interaction between entities.

Considering these architectural concepts we can identify two distinct domains for system description:

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

The entity domain considers aspects related to the structure of entities. These aspects involve the identification of the entities represented in the design, and their interconnection. An entity is delimited by interaction points and contains action points. Interaction points are shared by two or more entities, forming the common means of interaction of these entities. Each action point, however, can only belong to a single entity.

The behaviour domain considers aspects related to actions and interactions, and the relationships between them, which characterize behaviour. These relationships are called *causality relations*. Behaviours, especially complex ones, have to be structured in terms of behaviour compositions. We consider behaviours from a prescriptive point of view, i.e. they should be interpreted by the implementor as prescriptions of functional entities on how to build them.

There is a mapping from the behaviour domain onto the entity domain, such that behaviours are assigned to functional entities, and actions and interactions are assigned to action

points and interaction points. The composition of the behaviours assigned to entities has to be compatible with this combination of entities. This implies that interactions between entities can only occur at common interaction points, i.e. through which these entities are interconnected.

Figure 1 depicts the aspects considered by the entity and behaviour domains.

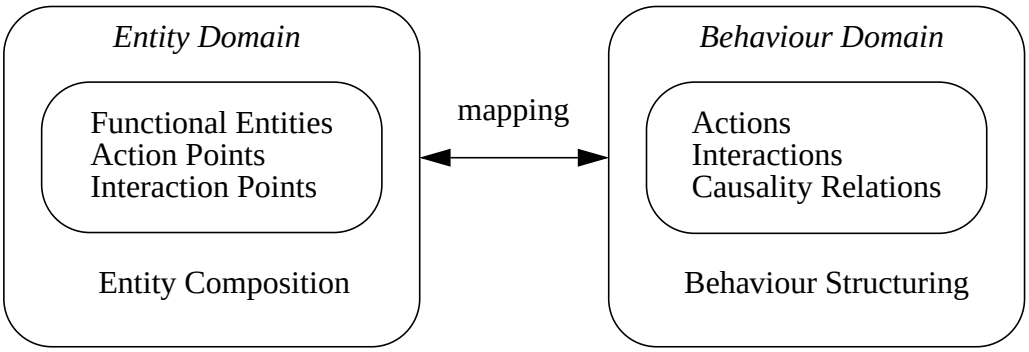


Figure 1: Entity and Behaviour Domains

Most design cultures lack the identification of entity and behaviour domains and concentrate on only one of these domains, while to our belief attention should be drawn to both. For example, in the elaboration of a design at a certain abstraction level one needs to define the entity structure as well as the behaviour assigned to each specific entity.

Figure 2 illustrates the entity domain and the behaviour domain and their relations for an arbitrary combination of entities F_1, F_2, F_3 and F_4. Behaviours B_1, B_2, B_3 and B_4 are assigned to F_1, F_2, F_3 and F_4, respectively. ip_1, ip_2, ip_3 and ip_4 are interaction points, and ap_1 and ap_2 are action points. The composition of behaviours B_1, B_2, B_3 and B_4 should represent the composition of entities F_1, F_2, F_3 and F_4. For example, interactions shared by B_1 and B_3 can only occur at the interaction point ip_2.

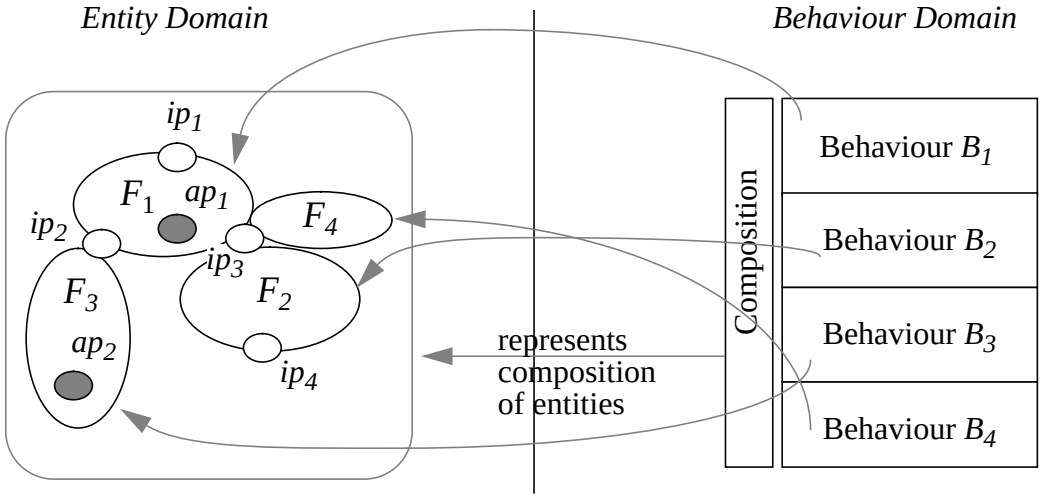


Figure 2: Example in Entity and Behaviour Domains

4 Five Related Abstraction Levels

Considering an arbitrary open distributed system, there are many possible alternative choices for the selection of abstraction levels. However, since we aim at applying these abstraction levels to system development, we have identified abstraction levels by defining their relative position in the total design trajectory and their global design goals. In an instance of a design process, where initially the system of interest does not exist and has to be built, these abstraction levels can be traversed from the higher abstraction levels to the lower, such that increasingly more details of the system are considered.

The identification of abstraction levels is most conveniently performed in the entity domain. The abstraction levels are therefore characterized by (compositions of) entities.

4.1 System Embedded in its Environment

Objective: definition of the application environment in which the system has to operate, in terms of entities of this environment and their cooperation. This abstraction level is useful to determine the activities of the (bounded) environment which should be supported by the distributed system and to determine which degree of support should be achieved.

This abstraction level is especially helpful when a system has to be designed from scratch, and has to be incorporated in an environment with the goal of supporting or enhancing some function of that environment. Effective applications of the system generally require that in such a case also the design of some activities of the environment has to be reconsidered. Examples can be found in the area of Computer Supported Cooperative Work (CSCW), where socio-technical systems are designed consisting of a computer system and a work organization in which the system is embedded ([1]).

It appears that little experience exists in the development of models at this abstraction level. Possible reasons for that are (i) the variety of applications, which makes it necessary to categorize them and to develop different models for different categories of applications, and (ii) the need for expertise on these different application areas for a proper modelling.

Since this abstraction level is used to explore the role of the system in support of some function of the environment, aspects of the environment, i.e. outside of the system, as well as aspects of the (role of the) distributed processing system may be considered.

4.2 Interaction System between System and Environment

Objective: definition of the shared boundary between the system and its environment. This abstraction level assigns the common behaviour performed by a system and its environment to a single functional entity, their *interaction system*, such that distributions of responsibilities and constraints between the system and the environment are not considered.

At this abstraction level many requirements of the common behaviour of the system and its environment can be defined, such as temporal ordering of actions, timing and reliability aspects, etc. Models at this abstraction level should be derived from the definition of the system embedded in its environment, by proper selection of functions.

Figure 3 illustrates the system embedded in its environment and the interaction system between system and environment.

The environment which embeds the system is shown in Figure 3 as consisting of three parts, reflecting some actual structure of the environment, with cooperation between the parts represented by double headed arrows. Although not considered at this level, the cooperation between parts may not be direct, but through intermediate entities. The purpose of the system design may then be to find a proper implementation of this cooperation.

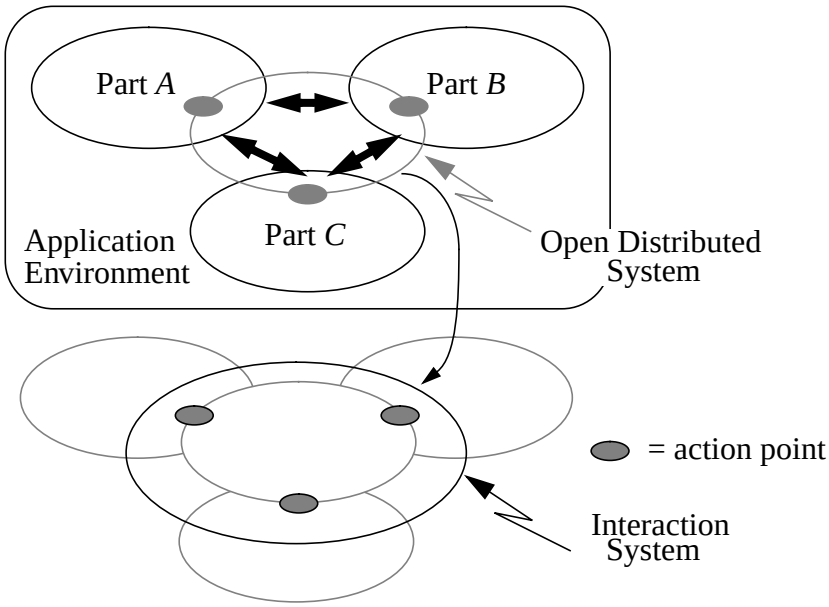


Figure 3: System Embedded in its Environment and Interaction System between System and Environment

4.3 Integrated Perspective of a System

Objective: definition of the behaviour of the system as it is observed by its environment. At this level of abstraction we consider the responsibilities and constraints that have to be assigned to the system and to its environment in performing interactions. Possible internal organizations of the system that result in the same observable behaviour are not considered at this abstraction level.

Models at this abstraction level should be derived from the definition of the interation system between system and environment, by proper selection of responsibilities in the establishment of interactions between system and environment.

Figure 4 illustrates the interaction system between system and environment, the integrated perspective of a system, and their possible relationship.

4.4 Partitioned Perspective of a System

Objective: definition of the application support functions, without considering the communication infra-structure (distribution). This abstraction level identifies logical functions that support the functional requirements of the integrated perspective of the system, such that their combination conforms to the integrated perspective of the system. It should be derived from the definition of the integrated perspective of the system, by identifying (logically) orthogonal functions and distributing them onto application support components.

Figure 5 illustrates the integrated and the partitioned system perspectives and their possible relationship. The partitioned perspective should conform to the integrated perspective, such that the behaviour of these two perspectives cannot be distinguished from the environment point of view.

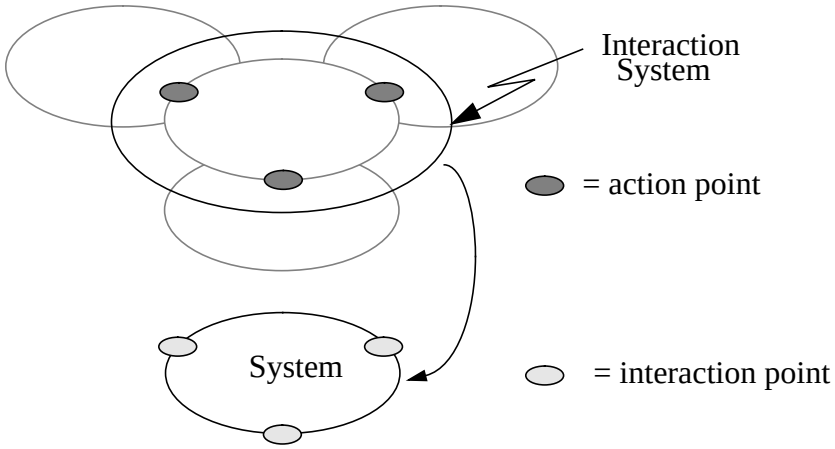


Figure 4: Interaction System between System and Environment and Integrated System Perspective

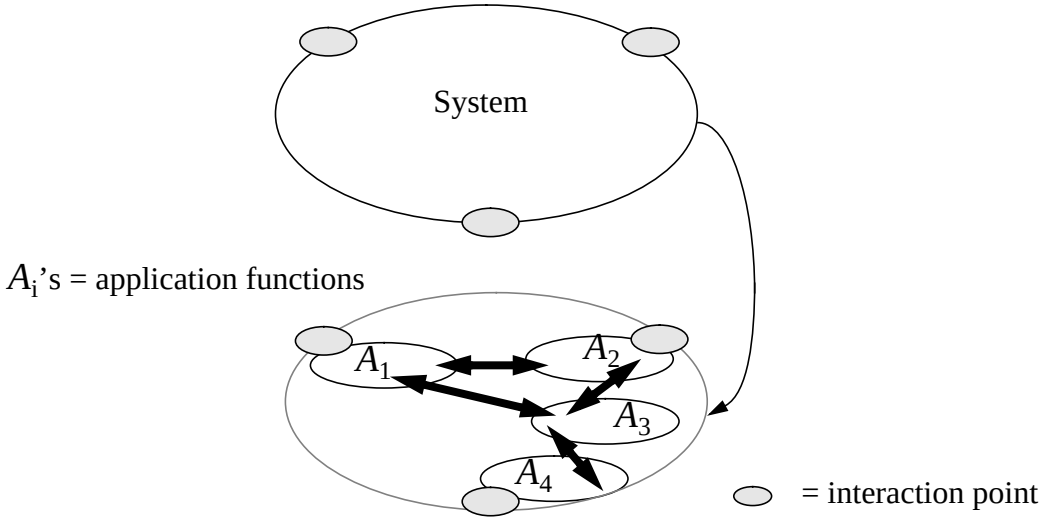


Figure 5: Integrated and Partitioned System Perspectives

4.5 Distributed Perspective of a System

Objective: definition of the functional requirements for inter-working at the application and communication levels.

The distributed system perspective should be derived from the partitioned system perspective, by considering the communication infra-structure that supports the communication between application functions, taking into account how the application functions use the communication infra-structure in order to operate. Cooperation between application components defined in the partitioned system perspective are supported by the communication infra-structure in the distributed system perspective.

Figure 6 illustrates the partitioned and the distributed system perspectives, and their possible relationships.

Heterogeneity between various implementation environments, such as hardware and operating systems issues, makes it unnecessary and even undesirable to consider standardization further than the distributed system perspective, except for concrete interfacing.

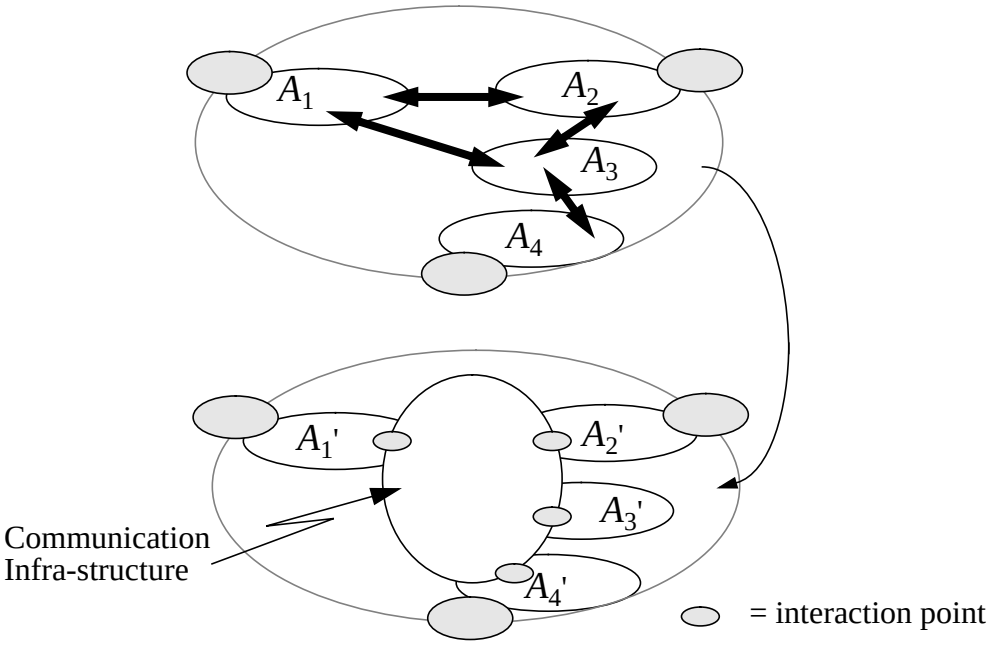


Figure 6: Partitioned and Distributed System Perspectives

4.6 Abstraction Levels and ODP Viewpoints

According to the current definition of viewpoints in the ODP-RM, viewpoints can not be directly mapped onto a set of related abstraction level. However we indicate in the sequel possible relationships between aspects of the ODP viewpoints and the abstraction levels introduced before.

The ODP enterprise viewpoint could be related to the abstraction level of the system embedded in its environment. In particular the behavioural roles and activities performed by ODP systems are addressed by this abstraction level.

The ODP information viewpoint could be related to the information established in (inter)actions at each of the abstraction levels defined above.

The ODP computational viewpoint could be related to the abstraction level of system's partitioned perspective. Distribution transparent objects, activities and interactions defined in this viewpoint could correspond to entities, behaviour and interactions of a system's partitioned perspective.

The ODP engineering viewpoint could be related to the abstraction level of system's distributed perspective. In particular the organization of an abstract infra-structure to support interworking corresponds to the communication infra-structure depicted in Figure 6.

The ODP technology viewpoint is not considered by our abstraction levels.

5 Elementary Behaviour Concepts

The behaviour of an entity is defined in terms of relationships between the actions and interactions of this entity. These relationships result in a specific ordering between these

actions and interactions. This section presents a collection of concepts that allow for the definition of behaviours.

5.1 Actions and Interactions

We suppose that there is an activity in the real world that we want to model from which all details are known. A possible approach is to select the most essential elements of this activity at a certain abstraction level and model them as actions, allowing us to reason about these activities without the burden of their details. Therefore we introduce the concept of *action* which is a unit of activity that is assigned to a functional entity at a specific abstraction level.

Since a designer generally wants to be able to refer to individual occurrences of actions, we assume that each action can be distinguished from the others. Actions are distinguished according to specific modelling and design goals. This means that we can, for convenience, assign a unique *identifier* to each action, allowing to refer to each individual action.

An action can be characterized by one or more *attributes*. The following attributes are considered in this text:

- *location*: defines where an action is allowed to occur;
- *time*: defines when an action is allowed to occur;
- *information* (local results): defines the possible results of an action, in terms of values of information established by its occurrence;
- *functionality* (passed results): defines values of information that are passed to an action by previous actions, so extending the local results of the action.

We should be able to define what values are possible for an action's attributes. Restrictions on the values of an attribute are called *constraints*. An attribute may have zero, one or more constraints associated with it.

The following examples illustrate some possible actions:

a	action with identifier *a*, no attributes considered
b (v: Nat)	action *b*, with unconstrained information attribute of type *Nat*
c (v: Nat) [2 < v < 10]	action *c*, with information attribute of type *Nat*, constrained between the values *2* and *10*

An *interaction* is a unit of activity that is common to two or more functional entities, and is defined such that the contributions of each functional entity to the interaction can be distinguished. An interaction can therefore be considered as a decomposition of an action in the sense that its occurrence is visible to the involved functional entities, and its attribute values are determined by the conjunction of individual constraints imposed by all the participating functional entities. A contribution of a functional entity to an interaction can be characterized using the same attributes as the ones we have considered for an action.

We illustrate some interactions, with contributions from two functional entities, that correspond to the actions above.

interaction contribution 1	interaction contribution 2	corresponding action
a	*a*	*a*
b (v:Nat)	*b (v:Nat)*	*b (v:Nat)*

c (v:Nat) [v < 10] *c (v:Nat) [v > 2]* *c (v: Nat) [2 < v < 10]*

In the following, we use the term *action* to refer to actions or interactions, unless it is felt necessary to be specific about one of them.

5.2 Causality Relations

The role of an action in a behaviour is determined by its relationships with other actions of this behaviour. These relationships are defined by means of causality relations. A *causality relation* states the conditions which enable and constrain the occurrence of an action. These conditions are called the *enabling condition* for the action, and the action itself is called the *result* action. An enabling condition specifies the occurrence and non-occurrence of actions that are required for the result action to occur. The result action only refers to the actions in the enabling condition, which can be seen as the minimal state information necessary for the result action. Causality relations therefore form an appropriate basis for the definition of the behaviour of open *distributed* systems, which do not have a global state, but rather a collection of 'sub-states' (multiple threads of control).

Consider the situation in which an action a_2 is allowed to occur only if another action a_1 has occurred. We represent this by the causality relation a_1 -> a_2, where a_1 is an enabling condition and a_2 is the result action. Although no explicit reference to time attributes are included so far, we assume that a_1 must have occurred before a_2. This time condition is always implicitly present in case of causality with the occurrence of an action in an enabling condition.

Consider now the situation in which an event a_2 is allowed to occur only if another event a_1 has not occurred (before nor at the same time). We represent this by the causality relation $\neg a_1$ -> a_2. The implicit time condition related to this relation is that if both a_1 and a_2 occur in a certain system run, a_1 should occur after a_2.

Arbitrary complex enabling conditions can be constructed by combining occurrence and non-occurrence of actions using the logical operators $\wedge$ and $\vee$. Some elementary examples are:

$a_1 \wedge a_2$ -> a_3	(conjunction of occurrences)
$a_1 \wedge \neg a_2$ -> a_3	(conjunction of occurrence and non-occurrence)
$a_1 \vee a_2$ -> a_3	(disjunction of occurrences)
$a_1 \vee \neg a_2$ -> a_3	(disjunction of occurrence and non-occurrence)

Enabling conditions may be defined in terms of specific attribute values of the enabling action occurrences. Constraints on the attributes of a result action can make reference to attribute values of the actions in the enabling conditions. Some examples are:

a_1 *(v_1:Nat) [5 < v_1 <10]* -> a_2 (the enabling condition of a_2 is that a_1 happens with value v_1 between 5 and 10)

a_1 *(v_1:Nat)* -> a_2 *(v_2:Nat) [v_2 = v_1 + 10]* (the value v_2 established in a_2 is constrained by a reference to the value v_1 established in action a_1 of the enabling condition)

a_1 *(v_1:Nat) [5 < v_1 <10]* -> a_2 *(v_2:Nat) [v_2 = v_1 + 10]* (combination of attribute value conditions and constraints)

5.3 Behaviour Definition

The behaviour of a functional entity can be characterized by the following elements: initial actions, relationships between actions, and termination conditions of the behaviour. We represent behaviour as a set of causality relations between actions, one causality relation for each action, which describes the conditions and constraints of this action. Initial actions are enabled by a special *start* condition, which means that these actions do not depend on other actions. Examples of behaviours are:

$B_1 := \{start \rightarrow a_1, a_1 \rightarrow a_2\}$ defines the sequential ordering of a_1 and a_2.

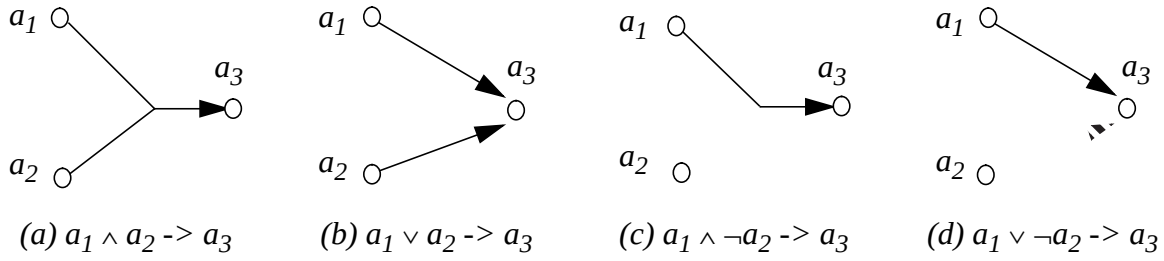
$B_2 := \{start \rightarrow a_1, a_1 \wedge \neg a_2 \rightarrow a_3, a_1 \wedge \neg a_3 \rightarrow a_2\}$ defines the sequential ordering of a_1 and a non-deterministic choice between a_2 and a_3.

$B_3 := \{start \rightarrow a_1, start \rightarrow a_2\}$ defines the independence of a_1 and a_2.

5.4 Graphical Notation

A graphical notation for causality relations is used as an alternative representation to the textual form presented above. This graphical notation is expected to be useful for helping understanding and analyzing causality relations. In this notation actions are indicated as circles, and causality relations as arrows. Interactions are represented as circle segments, each segment representing a contribution to the interaction, such that the composition of segments corresponds to the original common action (see Figure 11 for example).

Figure 7 depicts some examples of causality relations between actions and their representation.



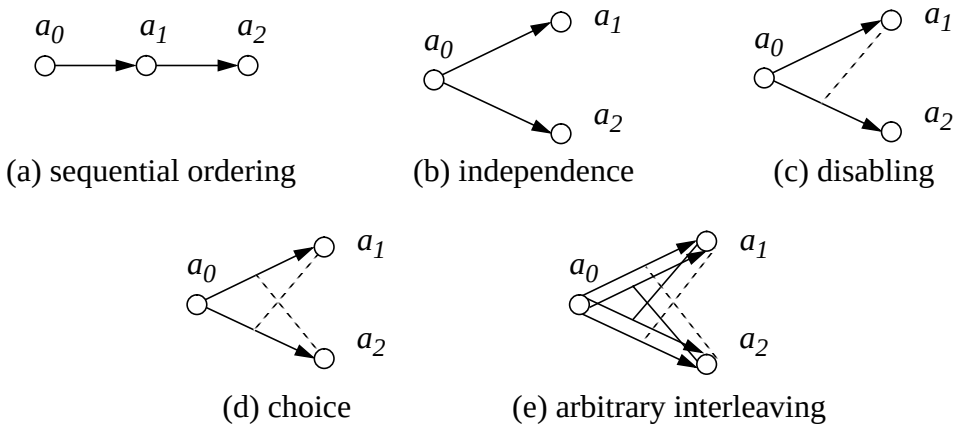
(a) $a_1 \wedge a_2 \rightarrow a_3$ (b) $a_1 \vee a_2 \rightarrow a_3$ (c) $a_1 \wedge \neg a_2 \rightarrow a_3$ (d) $a_1 \vee \neg a_2 \rightarrow a_3$

Figure 7: Graphical Representation of Causality Relations

Figure 8 depicts some well-known behaviour patterns using the graphical notation. In this figure the conditions $a_0 \wedge (a_1 \vee \neg a_1)$ and $a_0 \wedge (a_2 \vee \neg a_2)$ are represented in terms of their equivalent forms $(a_0 \wedge a_1) \vee (a_0 \wedge \neg a_1)$ and $(a_0 \wedge a_2) \vee (a_0 \wedge \neg a_2)$, respectively.

5.5 Mapping onto Petri Nets

This section presents the mapping of causality relations onto place/transition Petri nets ([14]). We take Petri nets for this since they support the representation of true parallelism, and moreover, are based on causal relationships between transitions. Thus we may expect a straightforward and intuitive mapping of concepts like actions and causality relations onto nets. Furthermore, nets have a nice and intuitive graphical representation, and a lot of extensions of nets are known, such as timed, stochastic and colored nets, that seem to be useful when decorating actions with attributes like time, probabilities, and values.

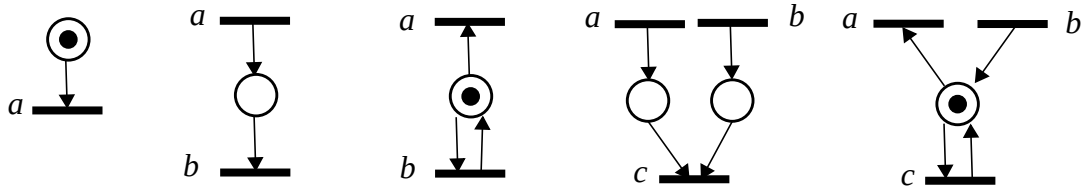


(a) sequential ordering (b) independence (c) disabling

(d) choice (e) arbitrary interleaving

Figure 8: Some Behaviour Patterns

A net consists of a set of places (represented by circles), a set of transitions (fat bars), a flow relation connecting states (transitions) to transitions (states), depicted as arrows, and a marking assigning one or more tokens (black dots) to some states. A transition, which is the semantic counterpart of an action, is able to execute when all its incoming states contain at least one token, that is, when all conditions in the corresponding causality relation are fulfilled. On execution it consumes a token from all its incoming states, and produces (instantaneously) a token in all its outgoing states. Transitions that are able to execute may execute in parallel; there is no need for interleaving of transitions. The nets that correspond to some elementary causality relations are depicted in Figure 9. We assume that each transition may fire at most once.



start -> a *a -> b* $\neg a$ *-> b* $a \wedge b$ *-> c* $\neg a \vee b$ *-> c*

Figure 9: Net semantics of Some Elementary Causality Relations.

We conjecture that the net semantics of all behaviours constructed from causality relations can be generated from the three left-most nets of Figure 9 in a *compositional* way.

As an example we consider the net semantics of $a \vee \neg a$ *-> b* (see Figure 10), denoting that *b* should interleave with *a*. We also consider that actions may refer to attributes of enabling actions. To model this aspect nets are extended with open tokens, which represent attributes that may be referred to, and dotted arrows, i.e. transitions along which only open tokens are transferred. Closed tokens are transferred along solid arrows.

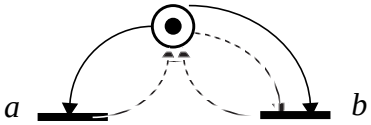


Figure 10: Net Semantics of $a \vee \neg a$ -> b.

It should be noticed that it suffices to have either a full or an open token (but not both) to be present in the common state in order for transition *b* to fire. When only considering the

ordinary solid transitions, the net is fully symmetric in *a* and *b*: if *b* is interleaved with *a*, *a* is interleaved with *b* also. However, when incorporating the way in which actions (transitions) may refer to each others' attributes, interleaving is *not* symmetric as *a* is not able to refer to attributes of *b*, whereas the reverse does hold (when *b* is caused by *a*).

The work on establishing a (formal) semantics is currently ongoing. We stress that a net semantics is only *one* of several possibilities to assign a semantics to the presented model. Other possible approaches could be, for instance, extensions of event automata ([12]), linear-time temporal logic ([10]), families of posets ([15]), or causal automata ([6]). A net semantics of finite behaviours is available. Extensions of this work towards a full semantics will include, amongst others, the formalization of implementation relation(s), decorating attributes and constraints on attributes to causality relations, definition of appropriate equivalence relations and considering recursive behaviours. This is subject of further research.

6 Behaviour Composition

Behaviour definitions in terms of monolithic causality relations, as presented in Section 5, are limited to the representation of simple finite behaviours. Recursion and complexity makes it necessary to introduce extra structuring mechanisms. Causality-oriented and constraint-oriented behaviour composition are the structuring mechanisms presented in this section.

6.1 Causality-oriented Behaviour Composition

Behaviours can be structured in terms of causality relations between behaviours, rather than between actions. A structuring mechanism, to be interpreted as a shorthand notation, has been introduced for this purpose. This structuring mechanism consists of:

- *entry points*: points in a behaviour that can be used to allow other behaviours to enable actions of this behaviour;
- *exits*: conditions in a behaviour that can be used to enable actions of other behaviours.

Behaviours can be composed by relating their exit and entry points. Entry points and exits are denoted with the keywords *entry* and *exit*, respectively. An example is:

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

In this example the entry of B_1 is enabled by a start, and the entry of B_2 is combined to the exit of B_1, such that a_2 becomes a condition for a_3.

Similarly to causality relations between actions, causality relations between (entries and exits of) behaviours allow the reference to attribute values. An example is:

B := {start -> B_1(entry), B_1(exit(v_2: Nat)) -> B_2(entry(v_2))}
where
B_1 := {entry -> a_1, a_1 -> a_2(v_2: Nat)[v_2 = 5], a_2(v_2:Nat) -> exit(v_2)}
B_2 := {entry(v_2: Nat) -> a_3(v_3:Nat)[v_3 < v_2]}

In this example, a_2 establishes value *5*, which is forwarded to a_3 by the *entry/exit* construct. After that a_3 establishes a value which is smaller than *5*.

We generalize the causality relations above in order to allow behaviours and actions to enable each other. An example is:

B := {start -> B_1(entry), B_1(exit) -> a_3}
where ...

Structuring behaviours in terms of these generalized causality relations allows reusability of components and the definition of hierarchies of behaviours in terms of sub-behaviours. This structuring technique is called *causality-oriented* behaviour composition.The notion of entry and exit points in a behaviour can be generalized in a natural way to allowing multiple entry and exit points. In this way, behaviours can be composed in a flexible way.

6.2 Constraint-oriented Behaviour Composition

Another structuring approach is based on the conjunction of constraints on actions, which are defined in separate behaviours. This structuring technique is called the *constraint-oriented* behaviour composition. It forces us to represent actions in a distributed form, since each action to be distributed is defined by a collection of causality relations in different behaviours that represent the constraints on the action. The combination of these causality relations completely determines the (constraints on) the action.

There are some options for the decomposition of a behaviour in sub-behaviours that represent constraints. Each individual action can be assigned to just one sub-behaviour or it can be shared by more than one sub-behaviour. Furthermore, in case an action is shared by sub-behaviours, its conditions and constraints, which were defined in a single causality relation in the monolithic behaviour definition, can be again distributed over these sub-behaviours in many different ways. These choices determine the design freedom designers have, and should be selected according to specific design objectives and quality principles.

Figure 11 shows an example of two design choices for the decomposition of a behaviour in constraints.

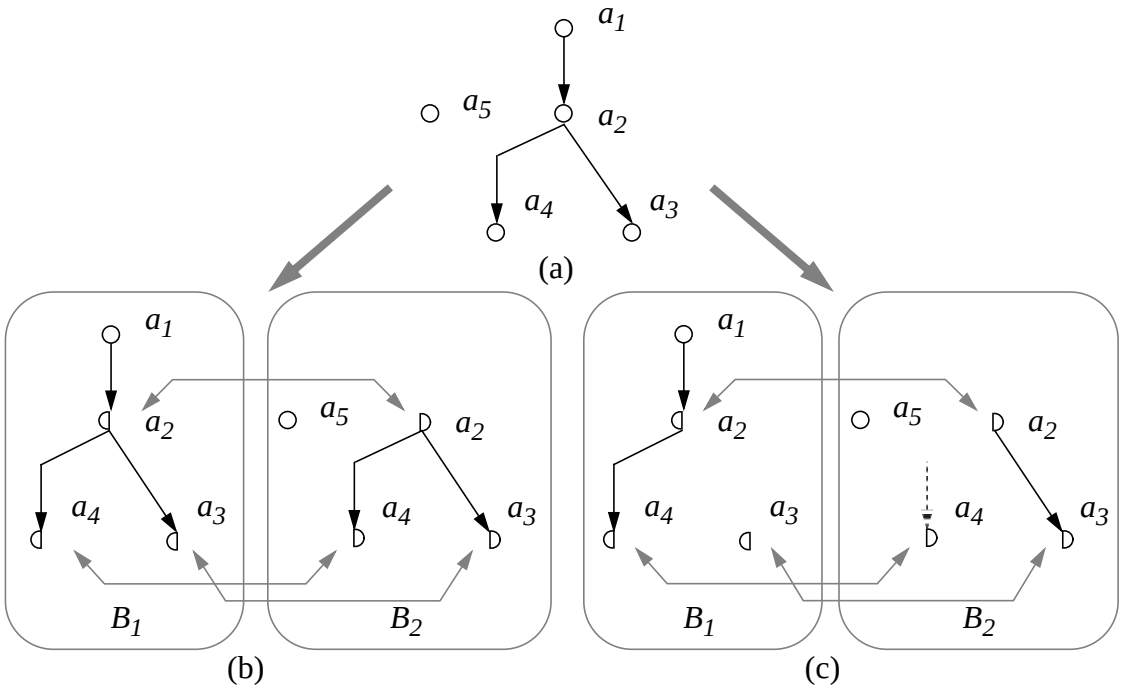


Figure 11: Example of Behaviour Decompositions in Constraints (a) Monolithic Behaviour, (b) Decomposition 1, (c) Decomposition 2

Figure 11(a) considers a monolithic behaviour which we want to structure such that actions a_2, a_3 and a_4 are distributed over sub-behaviours B_1 and B_2. These sub-behaviours represent specific constraints on actions a_2, a_3 and a_4. Actions a_1 and a_5 are not distributed, being assigned to B_1 and B_2 respectively. We refrain from discussing the specific design objectives that motivated these choices and the decomposition choices that follow.

Figure 11 (b) and (c) show two possible constraint-oriented decompositions of the monolithic behaviour of Figure 11(a). Non-decomposed actions are shown as circles, and decomposed actions are shown as semicircles. This graphical notation is used throughout this work.

Conditions can be duplicated in the sub-behaviours. Figure 11 (b) shows, for example, that condition a_2 enables a_3, can be placed in both B_1 and B_2. Conditions can also be distributed over sub-behaviours, since the composition of sub-behaviours implies that conditions and constraints of both B_1 and B_2 apply. Figure 11 (c) shows, for example, that the conditions for a_4, namely the occurrence of a_2 and the non-occurrence of a_5 can be distributed, such that occurrence of a_2 is guaranteed by B_1 and the non-occurrence of a_5 is guaranteed by B_2.

In some circumstances designers may have no choice of assignment of constraints on actions to behaviours. In Figure 11, the condition a_1 enables a_2, considering the distribution of actions to behaviours given in the example, can only be placed in B_1, which can be seen in Figure 11 (b) and (c).

The constraint-oriented behaviour composition supports the development of a design structure that distinguishes between functional entities with interactions between them.

6.3 Causality versus Constraint-oriented Composition

Causality-oriented behaviour composition using the *entry/exit* mechanism corresponds to decomposing causality relation(s) such that the conditions and result(s) are put in separate behaviours. The graphical representation of causality-oriented composition is that the conditions of causality relations are disconnected from the result actions. Figure 12 compares the graphical interpretation of causality-oriented and constraint-oriented behaviour compositions Observing Figure 12 we can notice that the causality-oriented behaviour composition looks as if someone has cut the causality relations with a knife, defining sub-behaviours in this way. In the constraint-oriented behaviour composition the knife goes through the actions, generating sub-behaviours that share these actions.

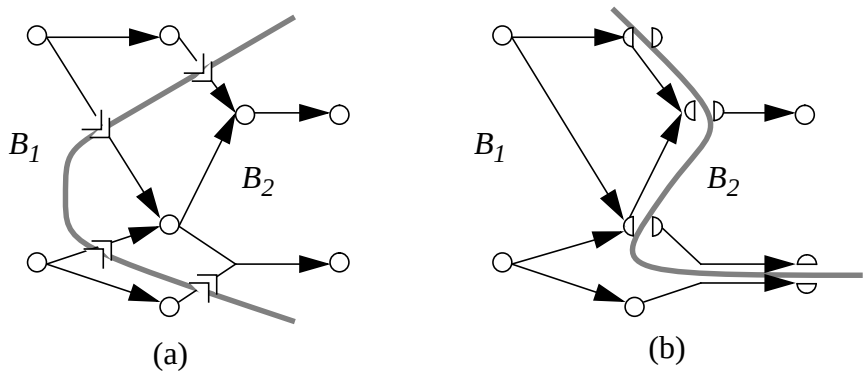


Figure 12: Graphical Representation of (a) Causality-oriented Behaviour Composition and (b) Constraint-oriented Behaviour Composition

7 Framework for Design and Implementation

This section discusses the application of our design model in a framework for the design and implementation of distributed systems.

7.1 Entity Composition and Behaviour Structuring

There are two main purposes for applying structuring in design: (i) understandability, which aims at getting overview of a complex design, and (ii) prescription for implementation, which aims at defining compositions of parts that should reflect the system implementation.

Entity structuring relates to prescription for implementation, i.e. a composition of entities is interpreted as the structure to be found in the actual implementation of the system.

Behaviour structuring can relate to both understandability and prescription for implementation. Understandability is supported when a certain structured behaviour (causality-oriented, constraint-oriented or a combination of both) is mapped onto a single functional entity. Prescription for implementation is supported when representing a composition of entities by the composition of their corresponding behaviours. In both cases behaviours are assigned to functional entities, and the combination of these behaviours has to comply to the *consistency conditions* derived from the combination of functional entities:

- interactions common to two or more behaviours assigned to functional entities happen at interaction points which are shared by all these functional entities;
- actions of a behaviour assigned to a functional entity happen at action points which are part of this functional entity.

These rules are the only consistency restrictions to the designer's freedom to define entity structures, behaviour structures and their relationship. Practical restrictions arise, for example, from technological limitations and the availability of specific system components.

Figure 13 depicts the design freedom for entity and behaviour domains. This figure shows that behaviours may be structured as monolithic, causality-oriented, constraint-oriented, or mixed causality-constraint-oriented structures. It also shows that the introduction of interactions (interaction points in the entity domain) forces some sort of constraint-oriented composition, either explicit, which means that interactions between functional entities are explicitly defined, or implicit, which means that only a system in terms of its interactions with an environment is defined.

Specification styles and their application to formulate LOTOS specifications ([21]) can now be placed in the perspective of these two domains. Since LOTOS does not support a separate definition of entity structuring and behaviour structuring, this could only be done now that we have obtained a more comprehensive framework.

The monolithic specification style corresponds to an unstructured representation of the behaviour of a functional entity. The constraint-oriented specification style also concentrates on the behaviour specification of a single functional entity, structuring this behaviour in terms of a composition of constraints, and corresponds to the constraint-oriented behaviour composition. The resource-oriented specification style concentrates on the representation of entity compositions by means of the composition of their behaviours, and considers a specific assignment of behaviours to functional entities, which can be generated from constraint-oriented behaviour compositions. The state-oriented specification style corresponds to the representation of the behaviour of a specific functional entity based on some assumptions on how this functional entity should be implemented in terms of a finite state machine.

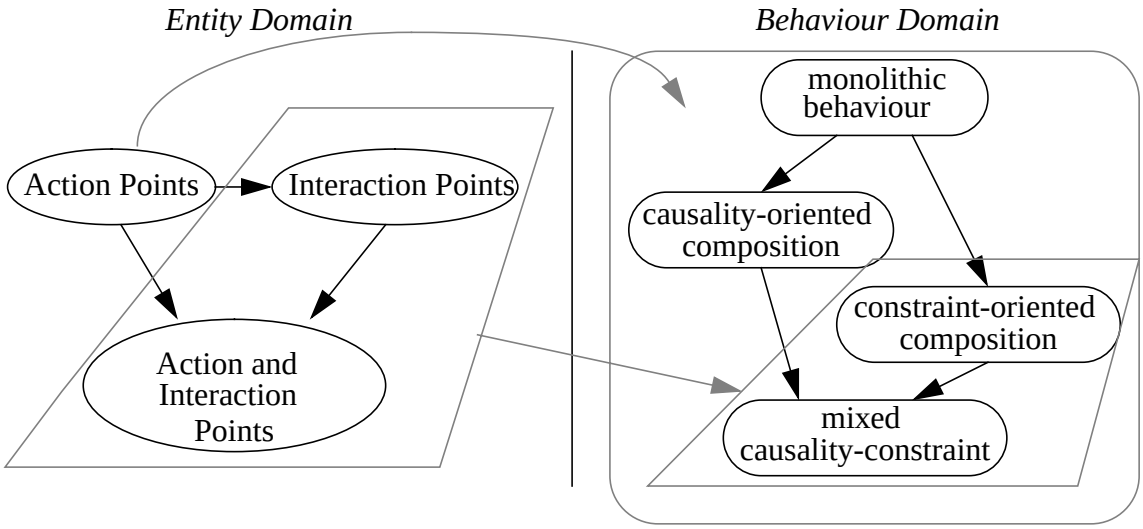


Figure 13: Design Freedom for Relating Entity and Behaviour Compositions

7.2 Design Steps

One of the benefits of identifying and working out separate but related models for the entity and behaviour domains is that this distinction enables a clear separation of design concerns for the characterization of design steps.

The design objectives of some important design steps are defined from the entity domain, in terms of manipulations of elements in this domain. Examples of such steps are functionality (entity) decomposition and interaction point refinement ([16]). Although the objectives of these design steps originate from the entity domain, they are also characterized by conditions that apply to the behaviour domain, thus defining correctness criteria for them.

Other design steps may have their objectives originated from the behaviour domain, and are defined in terms of manipulations solely in this domain, while the entity composition with its actions and interaction points remains intact. Examples are reduction of non-determinism and behaviour reduction or extension ([16]).

In the sequel we define two design steps, entity decomposition and introduction of action points, and discuss how these steps can be applied to perform design steps such as the ones determined by the abstraction levels presented before.

Entity Decomposition

Entity decomposition is a design step in which an entity of a certain abstraction level is replaced by more, possibly cooperating, entities at the next lower abstraction level. We restrict entity decomposition to a specific treatment of interaction and action points and specific behaviour conditions. We suppose that in the entity domain the following conditions hold:

1. original interaction points of the functional entity are also found in the decomposition, and
2. original action points are either maintained or transformed into interaction points in the decomposition.

Figure 14 depicts an example of entity decomposition.

In the behaviour domain we define the conformance between the composition of the behaviours of the decomposed functional entity and the original behaviour of the functional entity.

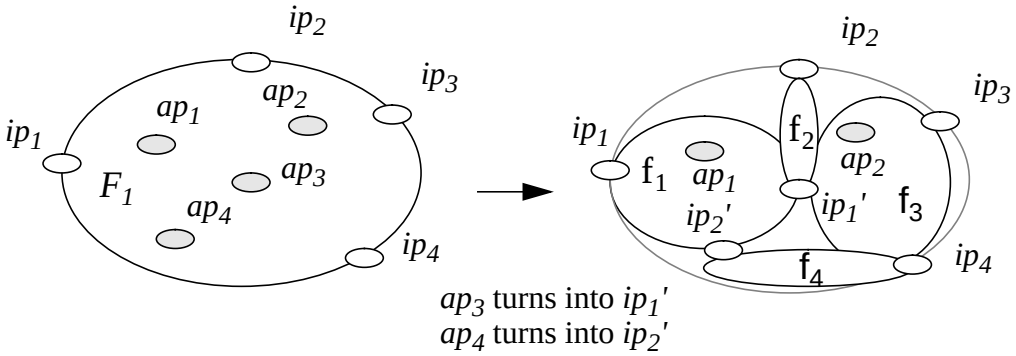


Figure 14: Entity Domain of Entity Decomposition Example

We expect that some form of behaviour isomorphism applies, in which some internal behaviour and relationships between the original interactions and actions are preserved by the decomposed functional entity. In this area we think that most research is yet to be done.

Introduction of Action Points

Introduction of action points is a design step in which action points are introduced in a functional entity.

We suppose that in the entity domain the following conditions hold:

1. the original interaction and action points of the functional entity are still found in the decomposition, and
2. some action points are introduced in the decomposed functional entity.

We may consider these new action points to be placed between existing actions or interaction points, such that in the behaviour domain we are forced to decompose the relationships between actions or interactions according to the placement of action and interaction points.

In the behaviour domain the condition seems to be a partial behaviour isomorphism, in which the relationships between the original actions and interactions is preserved in the decomposition. Again some research on this topic is necessary.

A specific form of partial behaviour isomorphism is the observable behaviour condition, in which behaviour as observed by the environment of a functional entity is supposed to be preserved by the decomposed functional entity. Several variations of this condition are available in the literature for process algebras, such as weak bisimulation equivalence, testing equivalence, etc. ([18]).

Figure 15 depicts an example of introduction of action points. This example shows that the introduction of action points may contain some more conditions concerning the relative position of the action points. In the example, action points ap_3 and ap_4 are placed between ap_1 and ap_2, such that all direct relationships between actions in ap_1 and ap_2 should now be made through actions at ap_3 and ap_4. In this simple example, the original condition that a_1 is a condition for a_2 is indirectly preserved by actions a_3 and a_4 in the decomposed behaviour.

7.3 Recursion

The design steps introduced in this section can be used recursively, defining a design methodology for the design and implementation of distributed systems. For example functional entities at some abstraction level can be decomposed, and the resulting decomposition yields again functional entities that can be decomposed and so on.

Figure 16 illustrates an example of recursive application of design steps.

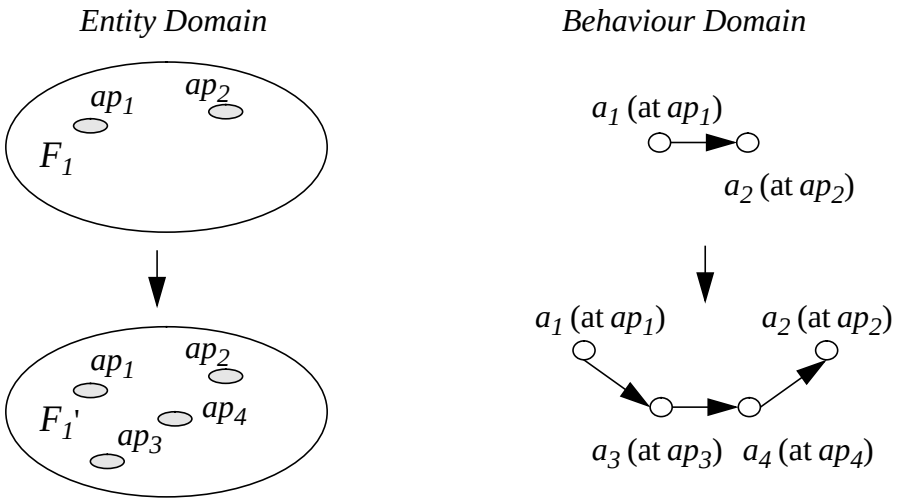


Figure 15: Example of Introduction of Action Points in Entity and Behaviour Domains

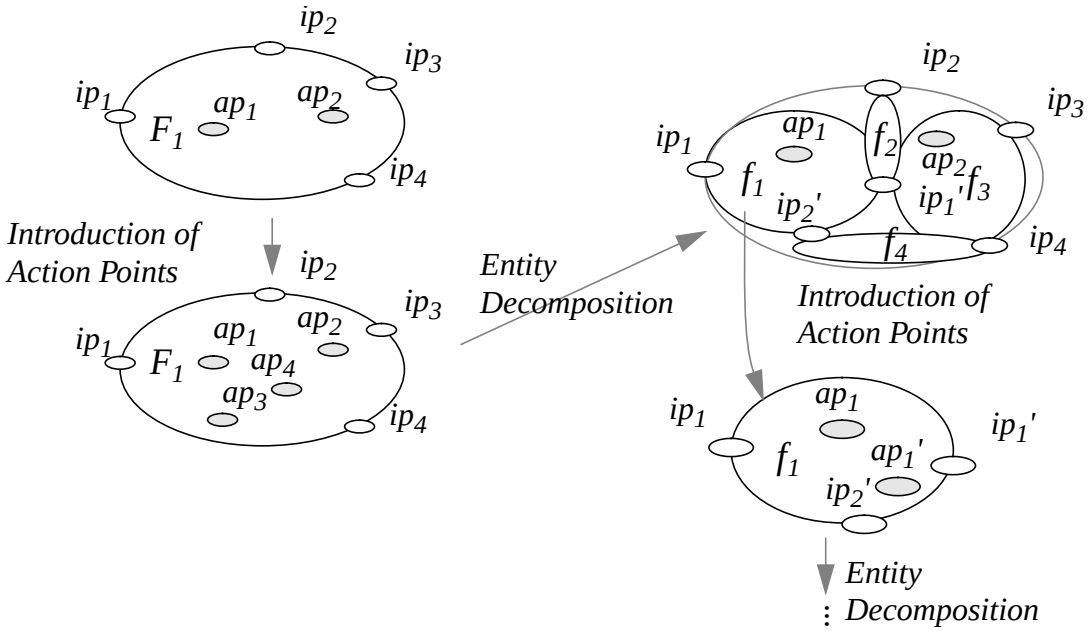


Figure 16: Recursive Application of Design Steps in Entity Domain

The design steps between the abstraction levels presented in Section 4 relate to the design steps presented before. The table below presents this relationship.

Embedded System -> Interaction System	Introduction of action points
Interaction System -> Integrated System	Entity decomposition
Integrated System -> Partitioned System	Introduction of action points and Entity decomposition
Partitioned System -> Distributed System	Recursion

8 Example

The use of our design model is illustrated by the design of a system which supports a Multimedia Information Exchange Service (MIES). This example shows how a design process can be carried out in a stepwise fashion according to the methodology presented before. Furthermore, it shows how real concurrency and timing conditions can be dealt with, which are notorious problems with many existing specification languages. It is clearly not our intention to focus on technological solutions to the problem of multimedia information exchange. Our example, therefore, simplifies the problems encountered in practical settings.

8.1 Informal Description of the Design Problem

In the context of multimedia systems, a medium denotes a type of information such as data, voice, audio, and video ([17]). A multimedia system supports multimedia applications that handle several media in an integrated fashion. In the case of a distributed multimedia system, this induces specific requirements on the subsystem that is concerned with the exchange of multimedia information between remote application processes. One significant requirement is that of *synchronization*, which assures a temporal relationship between information elements in accordance to the application.

In this example we consider the exchange of live audio and video between a source and a single destination. Audio and video are stream media, i.e. media that may be expressed as a function of real-time. Together they form a multiple stream, with sound-track synchronization[1] between the audio and video component. Since audio and video have different characteristics (e.g. in terms of sensitivity to delay variations and loss of samples), it is often desirable to treat these streams independently during transmission over a network ([9],[11]). For this reason we decompose the multiple stream into two single ones, viz. an audio and a video stream. Figure 17 depicts the example.

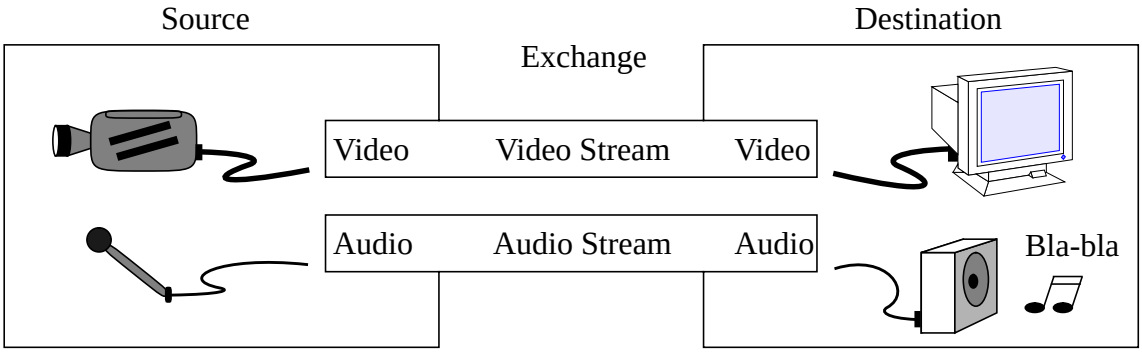


Figure 17: Exchange of Audio and Video

The synchronization requirements are formulated as follows:

1. the function of time of both audio and video streams should be preserved[2]. This means that the rate at which audio and video samples are produced at the source should be equal to the rate at which they arrive at the destination;

1. The term sound-track synchronization comes from motion pictures celluloid where the sound is recorded in a track along the picture frames. This kind of synchronization for voice and video is often called lip synchronization.
2. Within predefined limits, determined by human auditorial and visual perception, deviations can be tolerated. This is not further discussed here.

2. the sound-track synchronization between audio and video should be preserved. This means that the temporal relationship between audio samples and video samples at the source should also apply at the destination.

In the following, we assume that the stream components are *isochronous* in nature, i.e. the audio and video samples are generated at fixed time intervals. Furthermore, we do not consider the possibility of loss or corruption of samples.

The design problem is formulated independent of a specific application environment (e.g. video-conferencing), but rather in terms of general purpose multi-media support. Therefore we start the design at the abstraction level of the interaction system between the system and its environment, rather than at the abstraction level of the system embedded in its environment.

8.2 MIES Definition

The MIES design is concerned with the target application of the system, i.e. the exchange of audio and video such that the synchronization requirements are fulfilled. This enables the playback of audio and video at the destination as generated at the source. The MIES design does not address the possible limitations of available network technology, the synchronization anomalies that may occur as a consequence of these limitations, and the corrections of these anomalies such that the original goal (as presented by the MIES design) is attained. These concerns are deferred until next design steps. During later design steps it may turn out that the requirements set by the original goal cannot be satisfied. In such a case, a new, less ambitious goal should be formulated, leading to changes in the MIES design.

In the entity domain, we consider the MIES as an entity containing four action points. There are two action points co-located with the source, *a_in* and *v_in*, associated with the audio and video stream respectively; and there are two action points co-located with the destination, *a_out* and *v_out*, respectively associated with the audio and video stream.

In the behaviour domain, the submission of a sample and the arrival of a sample are represented as distinct actions. The submission of an audio sample is identified as *a_req*, and of a video sample as *v_req*; the arrival of an audio sample is identified as *a_ind*, and of a video sample as *v_ind*. The isochronous nature of the streams is represented by the constraint that audio samples are separated in time by the sampling delay Δ_a, and video samples by the sampling delay Δ_v. The synchronization requirements are represented by the constraint that the delay between submission and arrival of samples is constant (requirement 1) and the constraint that this delay should be identical for both streams (requirement 2). This delay, the latency or transmission delay, is denoted by δ.

Figure 18 shows the representation in both domains.

A textual representation of the MIES behaviour is given below.

MIES_behaviour := { start -> a_req,
a_req -> a_ind [$t_{a\_ind} = t_{a\_req} + \delta$],
a_req -> SAudio (entry (t_{a_req})),
start -> v_req,
v_req -> v_ind [$t_{v\_ind} = t_{v\_req} + \delta$],
v_req -> SVideo (entry (t_{v_req}))
where
SAudio := {entry (t_0)-> a_req [$t_{a\_req} = t_0 + \Delta_a$],
a_req -> a_ind[$t_{a\_ind} = t_{a\_req} + \delta$],
a_req -> SAudio (entry (t_{a_req})) }
SVideo := {entry (t_0)-> v_req [$t_{v\_req} = t_0 + \Delta_v$],

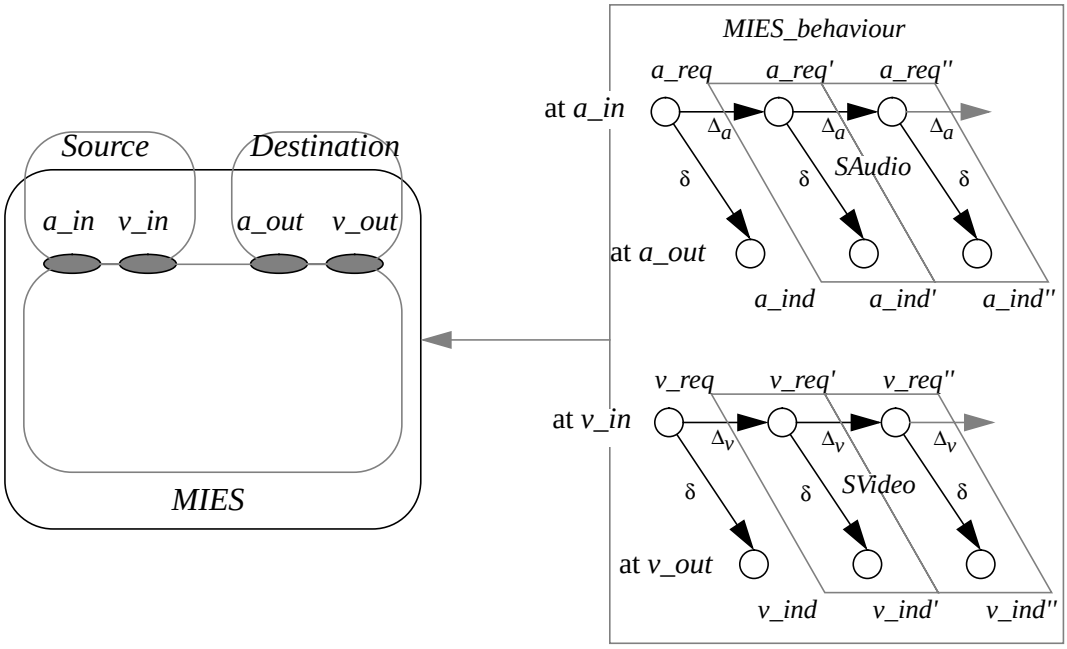


Figure 18: MIES Representation in Entity and Behaviour Domain

v_req -> v_ind [$t_{v\_ind} = t_{v\_req} + \delta$],
v_req -> SVideo (entry (t_{v_req})) } }

8.3 MIES Provider Definition

The next step is an application of entity decomposition where we determine the assignment of responsibility in the MIES actions to the system supporting the service, i.e. the MIES provider (MIESP). We assume that the MIESP is able to support the exchange of audio and video streams with constant as well as variable sample rates up to a certain maximum, depending on the sample sizes. This maximum concurs with the maximum throughput (in terms of samples/sec) supported by the MIESP. Assuming a fixed size for audio and video samples, we denote the maximum throughput supported for audio by $1/\lambda_a$, and the maximum throughput supported for video by $1/\lambda_v$.

The constraint on the submission of audio samples, from the MIESP point of view, is then that audio samples are separated in time by at least a time period λ_a. Similarly, video samples should be separated by at least a time period λ_v. In order for the MIESP to support the MIES, two conditions apply: $\lambda_a < \Delta_a$ and $\lambda_v < \Delta_v$. Furthermore, the MIESP is responsible for assuring that the transmission delay is equal to δ, as specified in the MIES definition.

Figure 19 represents the MIESP.

A textual representation of the MIESP behaviour is given below.

MIESP_behaviour := { start -> a_req,
a_req -> a_ind [$t_{a\_ind} = t_{a\_req} + \delta$],
a_req -> SPAudio (entry (t_{a_req})),
start -> v_req,
v_req -> v_ind [$t_{v\_ind} = t_{v\_req} + \delta$],
v_req -> SPVideo (entry (t_{v_req}))
where

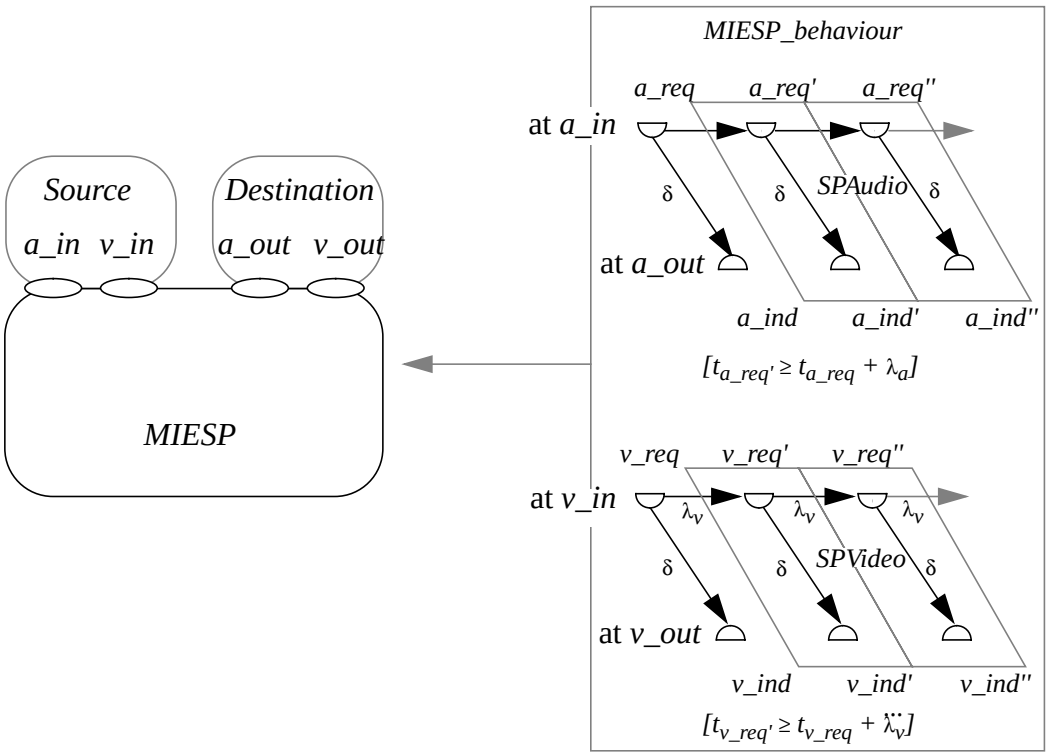


Figure 19: MIESP Representation in Entity and Behaviour Domain

$$SPAudio := \{entry\ (t_0) \text{-}> a\_req\ [t_{a\_req} \geq t_0 + \lambda_a],$$
$$a\_req \text{ -}> a\_ind[t_{a\_ind} = t_{a\_req} + \delta],$$
$$a\_req \text{ -}> SPAudio\ (entry\ (t_{a\_req}))\ \}$$
$$SPVideo := \{entry\ (t_0) \text{-}> v\_req\ [t_{v\_req} \geq t_0 + \lambda_v],$$
$$v\_req \text{ -}> v\_ind\ [t_{v\_ind} = t_{v\_req} + \delta],$$
$$v\_req \text{ -}> SPVideo\ (entry\ (t_{v\_req}))\ \}\ \}$$

8.4 MIEP Definition

According to our methodology, the logical next step in the design process (that is, the partitioned perspective) would be the consideration of a decomposition of the MIESP in terms of interacting application support functions. However, since the MIESP is only concerned with transfer and not with processing of multimedia information, we cannot ignore the distribution of functions, and should therefore immediately consider a distributed perspective of the system. This distributed perspective is called the Multimedia Information Exchange Protocol (MIEP).

The MIEP design considers the fulfillment of the synchronization requirements in the light of the characteristics of general purpose end-to-end data communication systems. General purpose communication systems are characterized by the supported data transfer quality of service (QoS), including properties such as data unit size, throughput, transfer delay, and reliability. In this example, we only consider throughput (in terms of data units/sec) and transfer delay in relation to two categories of data units: those that carry an audio sample (audio data units), and those that carry a video sample (video data units).

This design step can be seen as an application of the introduction of action points (Section 7). In the entity domain we represent the MIEP by four interaction points and four action

points. The interaction points are the same as those of the MIESP. The four action points are associated with the transfer of data units, one pair for the transfer of audio data units and one for the transfer of video data units. Although data transfer is assumed to be transparent, i.e. independent of the data contents, the two data transfer paths can be used to support different QoS properties in order to suit the different characteristics of audio and video streams. The action points where data units are sent are named *da_in* and *dv_in*; the corresponding action points where data units are received are respectively named *da_out* and *dv_out*. Notice that the four action points can be viewed as being the action points contained by a data transfer service (DTS). In the following, we will only further discuss the transfer of audio data units; by symmetry, a similar reasoning can be applied to the transfer of video data units.

An audio sample should be presented to the destination after latency δ. This means that the actual transfer delay of an audio sample, received in an audio data unit, should be known to the receiving side in order to provide for the required additional delay. In this example, we assume that the sending and receiving side have synchronized clocks. The sending side attaches a time stamp to the sample, and they are together transferred in an audio data unit. At the receiving side, the actual delay is determined by comparison of the time stamp with the local clock, and the sample is buffered for the time that remains until the control time expires.

We denote the properties that characterize the transfer of audio data units as follows:

- the transfer delay varies in between δa^- and δa^+(delay jitter); and
- the maximum throughput is $1/\lambda_{da}$ data units/sec.

Furthermore, the total protocol processing time per data unit at the sending and receiving side is denoted by δp. Given the requirements expressed in the MIES behaviour, the following conditions apply: $\delta a^+ + \delta p \leq \delta$, and $\lambda_{da} \leq \lambda_a$.

In the MIEP behaviour, the following constraints should be expressed with respect to a send action, identified as *da_req*, and a receive action, identified as *da_ind*:

- the elapsed time between *a_req* and the corresponding *da_req* is at most $\delta - \delta a^+$. Moreover, assuming equal processing times at the sender and receiver, the maximal elapsed time between *a_req* and *da_req* is $(\delta - \delta a^+)/2$;
- the minimal time between two subsequent *da_req*'s is λ_{da};
- the time period between *da_req* and the related *da_ind* is between δa^- and δa^+;
- the time period between a *da_ind* and the related *a_ind* is equal to the local protocol processing time (at most $(\delta - \delta a^+)/2$) plus the buffer time. This time period is calculated by the receiver entity on basis of the current time (concurring with the time at which the *a_ind* occurred) and the time stamp (indicating the time at which the related *a_req* occurred) as follows: δ - (current time - time stamp).

Figure 20 shows the representation of the MIEP so far (the constraints on *da_req* and *da_ind* are not indicated).

Another, more refined, representation of the MIEP can be obtained by application of entity decomposition, where the action points are transformed into interaction points of protocol entities and a DTS provider. Figure 21 depicts this decomposition, introducing separate protocol entities for the support of audio and video streams (only those in support of the audio stream are shown).

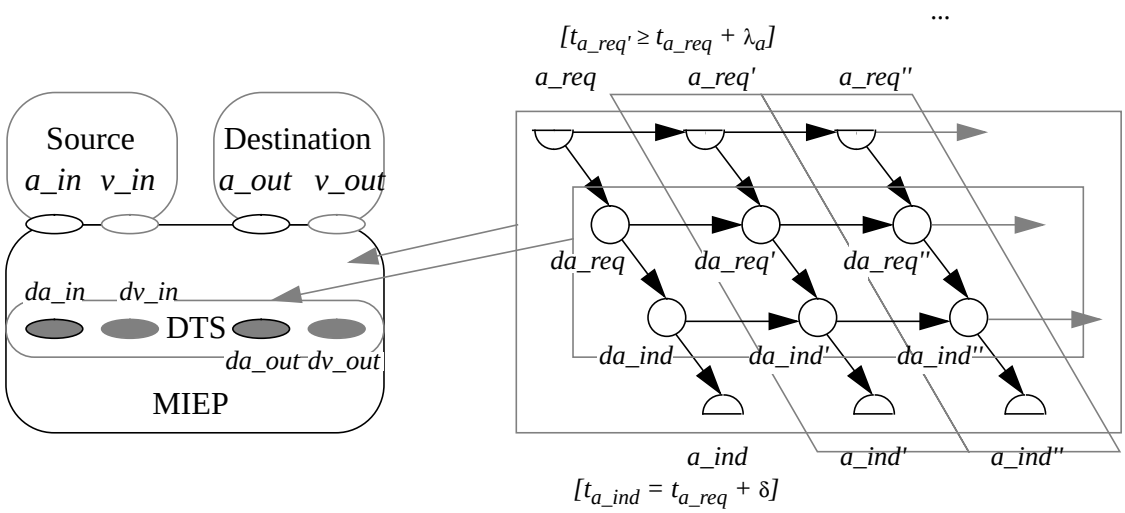


Figure 20: MIEP Representation in Entity Domain (without Entity Decomposition) and Behaviour Domain (Support of Audio Stream only)

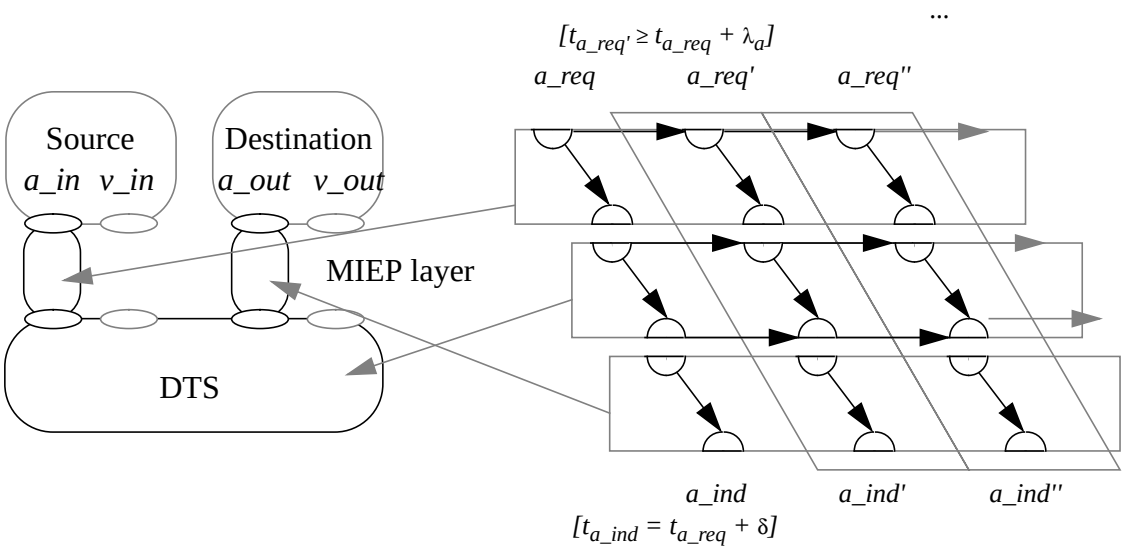


Figure 21: Decomposed MIEP Representation in Entity Domain and Behaviour Domain (Support of Audio Stream only)

9 Concluding Remarks

This paper discusses demands of design methodologies for ODP systems, and concludes that the current status of ODP-RM standard does not fully support these demands. This motivated the development of a design model. The purpose of this model is to support the design of open distributed systems in a systematic stepwise fashion. We argued that the concepts of this design model are useful in the context of ODP, namely for supporting the development of consistent and parsimonious standards in that area.

The basic design concepts for distributed systems have been grouped in two domains, viz. the entity domain and the behaviour domain. These two domains and their relationships form

a framework for design and implementation, in which design steps can be precisely formulated and performed. We have furthermore identified a number of related abstraction levels that can be used as milestones in a stepwise design process and we have related these abstraction levels through design steps, providing some guidance to designers.

The novelties of our approach are the explicit use of entity and behaviour domains, the use of actions and interactions in a single behaviour model, and the representation of behaviours through causality relations between actions and interactions. It is our belief that our model is complete with respect to functional requirements, such as timing, real parallelism and data. Aspects related to action probability are also being studied in the same framework. Further work should be done on the precise definition of implementation notions, especially those related to behaviour isomorphisms.

The choice of the design concepts has been based on architectural requirements that emerged from abstraction levels. No assumptions were made with respect to a (formal) semantics to support these concepts. Our notation is still rather ad-hoc and cannot be supported by any form of automated tool. However since we did not use any specific design language, we have been able to focus on the concepts to be manipulated in design, without considering the possible limitations design languages may bring. We admit, however, that language support is necessary in order to effectively apply the concepts in the design of practical systems. A fully developed design language to support the concepts presented in this paper constitutes therefore another important area in which further work is needed.

Although we have intentionally avoided to mention object-orientation in the paper, we believe that entities can be considered as objects, and the object-oriented paradigm applies to their behaviours. In this way encapsulation, re-usability, etc. would be incorporated in our design model. Behaviour definitions using causality relations would not oppose the object-oriented paradigm but rather serve as a technical filling.

Acknowledgements

Thanks are due to Harro Kremer for intensive discussion on the subjects covered in this paper. Dick Quartel is kindly acknowledged for his careful remarks and comments on a draft paper. The material in Section 8 is based on a specification of the service provider (and a similar protocol) in a timed variant of LOTOS ([2]) by Robert Huis in 't Veld. This will be reported by him in a forthcoming paper. Kazi Farooqui is acknowledged for his detailed comments on the relationship between abstraction levels and ODP viewpoints.

References

[1] L. J. Bannon and K. Schmidt. CSCW: Four characters in search of a context. In J. Bowers and S. Benford, editors, *Studies in computer supported cooperative work: theory, practice and design*, pages 3–16. North-Holland, 1991.

[2] T. Bolognesi, F. Lucidi, and S. Trigila. From timed Petri nets to timed LOTOS. In L. Logrippo, R. Probert, and H. Ural, editors, *Protocol Specification, Testing and Verification X*, pages 377–406. Elsevier Science Publishers B.V. (North-Holland), 1990.

[3] D. Bowen. Open distributed processing. *Computer Networks and ISDN Systems*, 23:195–201, 1991.

[4] C. Chabernaud and B. Vilain. Telecommunication services and distributed applications. *IEEE Network Magazine*, pages 10–13, 1990.

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

[6] J. Gunawardena. Causal automata. *Theoretical Computer Science*, 101:265–288, 1992.

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

[8] P. Linington. Introduction to the Open Distributed Processing Basic Reference Model. In J. de Meer, V. Heymer, and R. Roth, editors, *Open Distributed Processing*, pages 3–14. Elsevier Science Publishers B.V. (North-Holland), 1992.

[9] T. D. Little and E. Ghafoor. Multimedia synchronization protocols for broadband integrated services. *IEEE Journal on Selected Areas in Communications*, 9(9):1368–1381, 1991.

[10] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems Specification*. Springer-Verlag, 1992.

[11] C. Nicolaou. An architecture for real-time multimedia communication systems. *IEEE Journal on Selected Areas in Communications*, 8(3):391–400, 1990.

[12] G. Pinna and A. Poigné. On the nature of events. In I. Havel and V. Koubek, editors, *Mathematical Foundations of Computer Science 1992*, LNCS 629, pages 430–441. Springer-Verlag, 1992.

[13] E. Rechtin. The art of systems architecting. *IEEE Spectrum*, pages 66–69, October 1992.

[14] W. Reisig. *Petri Nets – An Introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

[15] A. Rensink. *Models and Methods for Action Refinement*. PhD thesis, University of Twente, Enschede, The Netherlands, 1993.

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

[17] R. Steinmetz. Synchronization properties in multimedia systems. *IEEE Journal on Selected Areas in Communications*, 8(3):401–412, 1990.

[18] R. van Glabbeek. The linear time - branching time spectrum. In J. Baeten and J. Klop, editors, *CONCUR'90. Theories of concurrency: unification and extension*, volume 458 of *Lecture Notes in Computer Science*, pages 278–297, Germany, 1990. Springer-Verlag.

[19] J. J. van Griethuysen. Open distributed processing (ODP). In E. Brinksma, G. Scollo, and C. A. Vissers, editors, *Protocol Specification, Testing, and Verification, IX*, The Netherlands, 1989. Elsevier Science Publishers B.V. (North-Holland).

[20] 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.

[21] 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.

[22] C. A. Vissers, M. van Sinderen, and L. Ferreira Pires. What makes industries believe in formal methods. In A. Danthine, G. Leduc, and P. Wolper, editors, *Protocol Specification, Testing, and Verification, XIII*, pages 3–26, The Netherlands, 1993. Elsevier Science Publishers B.V. (North-Holland).