Interner Bericht

Towards a Basic Reference Model of Open Distributed Processing

Reinhard Gotzhein
Fachbereich Informatik
Universität Kaiserslautern

Nr. 247/94

Fachbereich Informatik

Universität Kaiserslautern · Postfach 3049 · D-67653 Kaiserslautern

Towards a Basic Reference Model of Open Distributed Processing

Reinhard Gotzhein
Fachbereich Informatik
Universität Kaiserslautern

Nr. 247/94

Postfach 3049
D-67653 Kaiserslautern
Germany

phone: +49 631 205-3426
fax: +49 631 205-2640
email: gotzhein@informatik.uni-kl.de

Towards a Basic Reference Model of Open Distributed Processing

R. Gotzhein

University of Kaiserslautern, Postfach 3049, D-67653 Kaiserslautern, Germany
Email: gotzhein@informatik.uni-kl.de

Abstract

The Basic Reference Model of ODP introduces a number of basic concepts in order to provide a common basis for the development of a coherent set of standards. To achieve this objective, a clear understanding of the basic concepts is one prerequisite. This paper makes an effort at clarifying some of the basic concepts independently of standardized or non-standardized formal description techniques. Among the basic concepts considered here are: agent, action, interaction, interaction point, architecture, behaviour, system, composition, refinement, and abstraction. In a case study, it is then shown how these basic concepts can be represented in a formal specification written in temporal logic.

Keyword Codes: C.2.4; D.2.1

Keywords: Distributed Systems; Requirements/Specifications

1 Introduction

The objective of ODP is to address issues of cooperation between systems: "any task which requires more than one application process to accomplish is within the scope of [ODP]", so "the field of application of distributed processing is virtually unlimited" ([Gri89]). To make distributed processing open, specific standards for all kinds of applications will have to be developed. To achieve a coherent set of standards, the Basic Reference Model of Open Distributed Processing (RM-ODP, [ISO93,1-4]) is currently being developed. As part of this reference model, a number of basic ODP concepts[1] are informally introduced and formalized "by interpreting each concept in terms of the constructs of the different standardized formal description techniques" ([ISO93,2]), called *architectural semantics*, in [ISO93,4]. Following [ISO93,1], the objectives of this architectural semantics are:

- to define the basic ODP concepts formally;
- to provide the foundations for the development of ODP standards;
- to act as a bridge between the basic ODP concepts and the FDTs;

1 called "ODP basic modeling concepts" in [ISO93,2]

- to provide the basis for a uniform and consistent comparison between formal descriptions of the same standard in different FDTs.

While the purpose of formalizing the basic ODP concepts is to define their meaning uniquely and unambiguously, it is by no means obvious that this can be achieved by interpreting each concept in different FDTs. There are three main arguments:

- There is no formal definition of the basic ODP concepts on which their interpretation in different standardized FDTs can be based.
- Some basic ODP concepts are elementary, i.e., they cannot be defined in terms of other already defined concepts.
- There is no formal relationship between concepts of different FDTs used to interpret basic ODP concepts.

As a result, it is not possible to argue that the meaning of basic ODP concepts is defined uniquely and unambiguously. In fact, from the contents of [ISO93,4], it can be concluded that different meanings are associated with some concepts in different FDTs. Furthermore, a formal basis for the comparison between formal descriptions of the same standard in different FDTs is not achieved.

A clear understanding of the basic ODP concepts is vital for the success of ODP as such. This paper makes an effort at clarifying some of the basic ODP concepts independently of standardized or non-standardized FDTs. To achieve this, a very small number of elementary concepts, which are assumed to be sufficiently well-understood, are selected as a starting point. Based on these elementary concepts, further concepts are then formally defined using a mathematical notation. We emphasize that this treatment is independent of how these basic concepts may be expressed in any of the standardized or non-standardized FDTs. In a case study, it is then shown how they can be represented in a formal specification written in temporal logic.

2 Basic ODP concepts

Following [ISO93,1-4], we will now address a number of concepts that we consider as basic for the area of ODP. Some of these concepts are elementary in the sense that they cannot be defined in terms of other, already known concepts. Together, these concepts can form the core of a Basic Reference Model of ODP (RM-ODP). RM-ODP should be viewed as a common semantical model which can be used to give a conceptual meaning to formal descriptions. By referring to a single semantical model, formal descriptions written in different FDTs become comparable. The degree of comparability depends on the richness of RM-ODP, i.e., on the number of basic concepts that are incorporated, *and* on the expressiveness of the FDTs, i.e., to what extent the basic concepts can be represented in formal descriptions. For two formal descriptions FD_1 and FD_2 written in different FDTs, we have the situation shown in Figure 2.1. The meaning of FD_1 and FD_2 is defined in terms of semantical models $\mathcal{M}_1$ and $\mathcal{M}_2$, respectively (for instance, acceptance trees and transition systems, with suitable refinement relations). Additionally, parts of FD_1 and FD_2 represent concepts of RM-ODP. This also gives meaning to the formal

descriptions from another point of view. A necessary condition is that the meanings with respect to $\mathcal{M}_i$ and RM-ODP must not be in contradiction. However, they may address different aspects of a system. For instance, the structure of a formal description can be used to represent a conceptual system architecture, although it is not assigned a meaning in the FDT semantics itself.

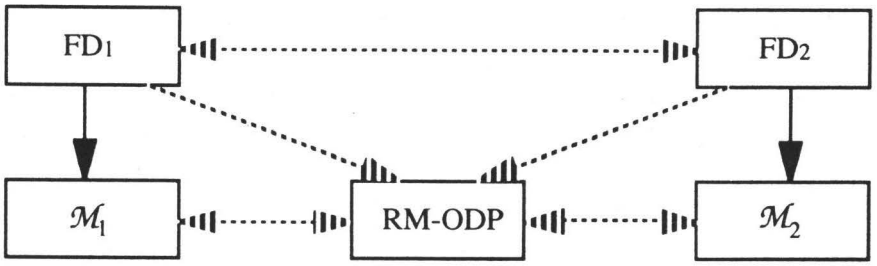

Figure 2.1: Comparison of formal descriptions in the context of RM-ODP

2.1 Elementary concepts

To lay the ground for the definition of basic ODP concepts, we start with the informal introduction of a very small number of elementary concepts. They form the starting point for the definition of further basic ODP concepts. Due to their elementary nature, these concepts cannot be formally defined. All that can be stated at this point is that these concepts are disjoint.

Definition (*elementary concepts*):

- An *agent*[2] $ag \in \mathcal{AG}$ is a component performing actions.
- An *interaction point* $ip \in \mathcal{IP}$ is a conceptual location where actions may occur.
- An *action*[3] $a \in \mathcal{Act}$ is something that happens.
- Agent, interaction point, and action are disjoint concepts, i.e. $\mathcal{AG} \cap \mathcal{IP} = \mathcal{AG} \cap \mathcal{Act} = \mathcal{IP} \cap \mathcal{Act} = \{\}$.

An action is performed by an agent or a set of agents, it may be internal or may occur at some interaction point or a set of interaction points. An agent thus is the carrier of actions, it can be characterized by its behaviour. This behaviour (a notion still to be defined) consists of actions local to the agent. Actions may also be non-local, such as interactions or transactions. Non-local actions may be performed by a *set* of agents and may occur at a *set* of interaction points. Interactions and transactions may also be considered as high-level actions, i.e., actions that can be decomposed into smaller units. Depending on what kind of action is taken as atomic on a given level of abstraction, the behaviour of a system can be characterized in different ways.

---

2 The notion of *agent* is closely related to the notion of *object* in [ODP93,2].

3 sometimes called *action occurrence*

2.2 Architecture

With the elementary concepts *agent* and *interaction point*, more complex structures termed *architectures* can be composed:

Definition (*architecture*):

- An *architecture* $\mathcal{Arch}$ is a structure $\langle \mathcal{AG}, \mathcal{IP}, \mathcal{ArchF} \rangle$, where
  - $\mathcal{AG}$ is a non-empty set of agents,
  - $\mathcal{IP}$ is a set of interaction points, and
  - $\mathcal{ArchF}: \mathcal{AG} \rightarrow 2^{\mathcal{IP}}$ is a total function called *architecture function* associating with each agent a set of interaction points.

A set $\mathcal{AG}^* \subseteq \mathcal{AG}$ of agents has one or more interaction points in common if and only if $\bigcap_{ag \in \mathcal{AG}^*} \mathcal{ArchF}(ag) \neq \{\}$. We require as a rule of composition that a common interaction point is introduced explicitly into the architecture whenever a group of agents has the capability to interact directly. Depending on the kind of interaction, two or more agents may in general be involved in interactions. Whether such interactions will actually take place also depends on the behaviour. Also, interaction points will be used to interconnect different architectures (see composition of architectures below).

Figure 2.2a shows a graphical representation of an architecture $\mathcal{Arch} = \langle \{ag_1, ag_2, ag_3\}, \{ip\}, \mathcal{ArchF}(ag_1) = \mathcal{ArchF}(ag_2) = \mathcal{ArchF}(ag_3) = \{ip\} \rangle$ consisting of three agents ag_1, ag_2, and ag_3 that have a common interaction point ip. From this architecture we can infer that ag_1, ag_2, and ag_3 have the capability to interact, however, we can not yet say whether they will actually do so. This can only be derived from the behaviour of the agents and the interaction point.

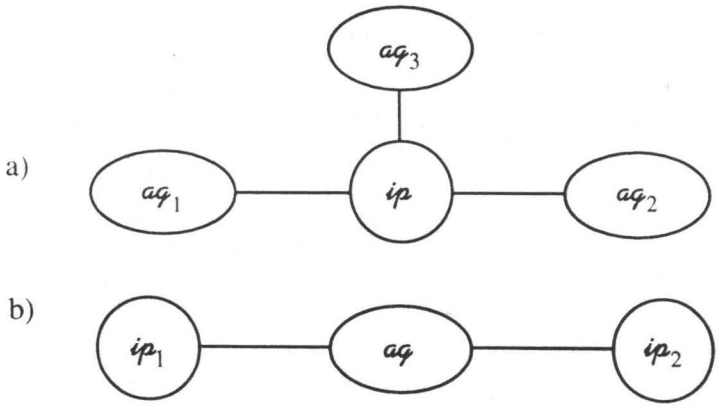

Figure 2.2: Graphical representation of architectures

Figure 2.2b shows an architecture $\mathcal{Arch} = \langle \{ag\}, \{ip_1, ip_2\}, \mathcal{ArchF}(ag) = \{ip_1, ip_2\} \rangle$ consisting of an agent ag with two associated interaction points ip_1 and ip_2. From a global point of view, this architecture could be considered as incomplete, because it does not show the agents which have the capability to interact with ag through ip_1 and ip_2. However, such a situation is frequently encountered in the context of open systems, which are incomplete in the same sense: although an

open system forms part of a larger system, the environment into which it is embedded later on is not considered from the beginning. In most cases, it is not even known in advance, and it is the task of the design to specify the open system such that it shows the intended behaviour in *every* possible environment. With respect to Figure 2.2b, we can say that the agents not shown in the architecture form the environment of ag. Following our rules of composition, we require that the interaction points an agent has with its environment are introduced explicitly, but we allow that the agents forming the environment may be omitted. They can, however, be added by composing two architectures, as the following definition shows:

Definition (*composition of architectures*):

- Let $Arch = \langle AG, IP, Arch_{AG} \rangle$ and $Arch' = \langle AG', IP', Arch_{AG'} \rangle$ be architectures. The *composition of architectures* $Arch$ and $Arch'$, written "$Arch \circ Arch'$ ", is defined as $\langle AG^\circ, IP^\circ, Arch_{AG}^\circ \rangle$, where
  - $AG^\circ =_{Df} AG \cup AG'$
  - $IP^\circ =_{Df} IP \cup IP'$
  - $Arch_{AG}^\circ : AG^\circ \rightarrow 2^{IP^\circ}$ is defined as follows:

$$Arch_{AG}^\circ(ag) =_{Df} \begin{cases} Arch_{AG}(ag) & ag \in AG \setminus AG' \\ Arch_{AG'}(ag) & \text{if } ag \in AG' \setminus AG \\ Arch_{AG}(ag) \cup Arch_{AG'}(ag) & ag \in AG \cap AG' \end{cases}$$

The definition allows architectures that are composed to have some parts in common. E.g., they may share a number of interaction points and/or a number of agents. If they have only interaction points in common, i.e., if $AG \cap AG' = \{\}$ holds, we have separate architectures that are interconnected at these (external) interaction points. As a result, we obtain a high degree of *modularity*, since each part of an overall architecture can be modeled separately and be merged into the final architecture by composition. Also, composition of architectures can be used to model *architectural extension*. From the definitions, it follows directly that $Arch \circ Arch'$ is an architecture, i.e., the dual role of agent and interaction point is preserved, and that $Arch \circ Arch = Arch$ and $Arch \circ Arch' = Arch' \circ Arch$ hold.

The architecture $Arch$ shown in Figure 2.2a can be obtained by composition of $Arch_1 = \langle \{ag_1\}, \{ip\}, Arch_{AG}(ag_1) = \{ip\} \rangle$, $Arch_2 = \langle \{ag_2\}, \{ip\}, Arch_{AG}(ag_2) = \{ip\} \rangle$, and $Arch_3 = \langle \{ag_3\}, \{ip\}, Arch_{AG}(ag_3) = \{ip\} \rangle$, i.e., $Arch = Arch_1 \circ Arch_2 \circ Arch_3$.

2.3 Behaviour

With the elementary concept *action*, more complex structures termed *behaviours* can be composed. We do not elaborate here on how this composition may be expressed in a specification language, but define composition in terms of the underlying semantical model:

Definition (*behaviour*):

- A *behaviour* $Behav$ is a structure $\langle Act, Behav_R \rangle$, where
  - Act is a set of actions,

- $\mathcal{BehavR} \subseteq \mathcal{Act} \times \mathcal{Act}$ is a partial order on the set of actions[4].

A behaviour imposes explicit constraints on the possible sequences of actions. Further constraints can be added by, e.g., composing behaviours. Behaviours can be composed if they are mutually consistent, i.e. if there are no contradictions between the partial orderings of actions:

Definition (*consistent behaviours*):

- A pair of behaviours $\mathcal{Behav} = \langle\, \mathcal{Act}, \mathcal{BehavR} \,\rangle$ and $\mathcal{Behav}' = \langle\, \mathcal{Act}', \mathcal{BehavR}' \,\rangle$ is called *consistent* if the following condition is satisfied:
  - $\forall a,a' \in \mathcal{Act} \cap \mathcal{Act}'.\ ((a,a') \in \mathcal{BehavR} \wedge a \neq a' \supset (a',a) \notin \mathcal{BehavR}')$.

As a special case, behaviours that have no actions in common are consistent. Consistency as defined here is a reflexive and symmetrical relation on behaviours.

Definition (*composition of behaviours*):

- Let $\mathcal{Behav} = \langle\, \mathcal{Act}, \mathcal{BehavR} \,\rangle$ and $\mathcal{Behav}' = \langle\, \mathcal{Act}', \mathcal{BehavR}' \,\rangle$ be consistent behaviours. The *composition of behaviours* $\mathcal{Behav}$ and $\mathcal{Behav}'$, written "$\mathcal{Behav} \circ \mathcal{Behav}'$ ", is defined as $\langle\, \mathcal{Act}^{\circ}, \mathcal{BehavR}^{\circ} \,\rangle$, where
  - $\mathcal{Act}^{\circ} =_{Df} \mathcal{Act} \cup \mathcal{Act}'$,
  - $\mathcal{BehavR}^{\circ} =_{Df} \mathcal{Cl}(\mathcal{BehavR} \cup \mathcal{BehavR}')$, where $\mathcal{Cl}(R)$ denotes the transitive closure of R.

A behaviour can be understood as a set of restrictions on the possible execution sequences. The definition allows behaviours that are composed to have some restrictions in common. The composition of behaviours can then be interpreted as the union of sets of restrictions. As in the case of architectures, we have a high degree of *modularity*, since each restriction can be modeled separately and be merged into the overall behaviour by composition. Also, composition of behaviours can be used to model *behaviour extension*. From the definitions, it follows directly that $\mathcal{Behav} \circ \mathcal{Behav}'$ is a behaviour, and that $\mathcal{Behav} \circ \mathcal{Behav} = \mathcal{Behav}$ and $\mathcal{Behav} \circ \mathcal{Behav}' = \mathcal{Behav}' \circ \mathcal{Behav}$ hold.

In specification languages, there exist several ways to compose behaviours, such as sequential or parallel composition. We do not consider these kinds of composition basic modeling concepts, but language dependent concepts. Also, we do not place any restriction on what actions are chosen to be atomic actions in the language, thus allowing for different levels of abstraction.

2.4 Systems

Having introduced elementary concepts and some rules how we can use them to define architectures and behaviours, let us now consider the notion of *system*.

Definition (*system*):

- A *system* $\mathcal{S}$ is a structure $\langle\, \mathcal{Arch}, \mathcal{Behav}, \mathcal{ActF} \,\rangle$, where
  - $\mathcal{Arch} = \langle\, \mathcal{AC}, \mathcal{IP}, \mathcal{ArchF} \,\rangle$ is an architecture,

---

4 sometimes called *action occurrences*

- $\mathcal{Behav} = \langle \mathcal{Act}, \mathcal{BehavR} \rangle$ is a behaviour,
- $\mathcal{ActF}: \mathcal{Act} \rightarrow 2^{\mathcal{AG} \cup \mathcal{IP}}$ is an action function associating with each action a set of agents and interaction points,
- $\forall a \in \mathcal{Act}.\ \mathcal{ActF}(a) \cap \mathcal{AG} \neq \{\}$,
- $\forall a \in \mathcal{Act}.\ \forall ip \in \mathcal{ActF}(a) \cap \mathcal{IP}.\ \exists ag \in \mathcal{ActF}(a).\ ip \in \mathcal{ArchF}(ag)$

A system is modeled by its architecture and its behaviour. Both aspects are to be respected when a system is refined and implemented. Additionally, there is a relationship between architecture and behaviour, which is expressed by the action function $\mathcal{ActF}$. This relationship must satisfy the constraint that an action must always be associated with some agent. Also, if an action is associated with some interaction point, then it must also be associated with some agent that is attached to that interaction point.

Definition (*distributed system*):

- If a system is *distributed*, then it consists of several agents[5], i.e., $|\ \mathcal{AG}\ | > 1$.

The notion of distribution does not refer to the external appearance, but to the internal organization of a system on a given level of abstraction.

With the preparations in previous sections, we define the composition of systems as follows:

Definition (*composition of systems*):

- Let $\mathcal{S} = \langle \mathcal{Arch}, \mathcal{Behav}, \mathcal{ActF} \rangle$ and $\mathcal{S}' = \langle \mathcal{Arch}', \mathcal{Behav}', \mathcal{ActF}' \rangle$ be systems such that $\mathcal{Arch} \circ \mathcal{Arch}' = \langle \mathcal{AG}^\circ, \mathcal{IP}^\circ, \mathcal{ArchF}^\circ \rangle$ and $\mathcal{Behav} \circ \mathcal{Behav}' = \langle \mathcal{Act}^\circ, \mathcal{BehavR}^\circ \rangle$ are defined. The *composition of systems* $\mathcal{S}$ and $\mathcal{S}'$, written "$\mathcal{S} \circ \mathcal{S}'$", is defined as $\langle \mathcal{Arch}^\circ, \mathcal{Behav}^\circ, \mathcal{ActF}^\circ \rangle$, where :
  - $\mathcal{Arch}^\circ =_{Df} \mathcal{Arch} \circ \mathcal{Arch}'$
  - $\mathcal{Behav}^\circ =_{Df} \mathcal{Behav} \circ \mathcal{Behav}'$
  - $\mathcal{ActF}^\circ: \mathcal{Act}^\circ \rightarrow 2^{\mathcal{AG}^\circ \cup \mathcal{IP}^\circ}$ is defined as follows:

$$\mathcal{ActF}^\circ(a) =_{Df} \begin{cases} \mathcal{ActF}(a) & a \in \mathcal{Act} \setminus \mathcal{Act}' \\ \mathcal{ActF}'(a) & \text{if } a \in \mathcal{Act}' \setminus \mathcal{Act} \\ \mathcal{ActF}(a) \cup \mathcal{ActF}'(a) & a \in \mathcal{Act} \cap \mathcal{Act}' \end{cases}$$

From the composition of architectures and behaviours, we obtain a high degree of *modularity*. Each system part can be modeled separately and be merged into the overall system by composition. Also, composition of systems can be used to model *system extension*. From the definitions, it follows directly that $\mathcal{S} \circ \mathcal{S}'$ is a system, and that $\mathcal{S} \circ \mathcal{S} = \mathcal{S}$ and $\mathcal{S} \circ \mathcal{S}' = \mathcal{S}' \circ \mathcal{S}$ hold.

Based on the notion of system, we can define further basic concepts:

---

5 We only consider architectural aspects at this point. A behavioural characteristic of a distributed system is its decentralized control.

Definition (*agent behaviour, interaction point behaviour, interface behaviour, interaction*):

Let $\mathcal{S} = \langle \mathcal{Arch}, \mathcal{Behav}, \mathcal{Act}\mathcal{A} \rangle$ be a system with $\mathcal{Arch} = \langle \mathcal{AG}, \mathcal{IP}, \mathcal{Arch}\mathcal{A} \rangle$ and $\mathcal{Behav} = \langle \mathcal{Act}, \mathcal{Behav}\mathcal{R} \rangle$.

- An *agent behaviour* $\mathcal{Behav}_{ag}$ being part of the system $\mathcal{S}$ is a behaviour consisting of all actions in which the agent $ag \in \mathcal{AG}$ participates. Formally: $\mathcal{Behav}_{ag} = \langle \mathcal{Act}_{ag}, \mathcal{Behav}\mathcal{R}_{ag} \rangle$ is a behaviour s. t. $ag \in \mathcal{AG}$, $\mathcal{Act}_{ag} = \{a \mid a \in \mathcal{Act} \wedge ag \in \mathcal{Act}\mathcal{A}(a)\}$, and $\mathcal{Behav}\mathcal{R}_{ag} = \mathcal{Behav}\mathcal{R}|_{\mathcal{Act}_{ag}}$.
- An *interaction point behaviour* $\mathcal{Behav}_{ip}$ being part of the system $\mathcal{S}$ is a behaviour consisting of all actions occurring at the interaction point ip. Formally: $\mathcal{Behav}_{ip} = \langle \mathcal{Act}_{ip}, \mathcal{Behav}\mathcal{R}_{ip} \rangle$ is a behaviour s. t. $ip \in \mathcal{IP}$, $\mathcal{Act}_{ip} = \{a \mid a \in \mathcal{Act} \wedge ip \in \mathcal{Act}\mathcal{A}(a)\}$, and $\mathcal{Behav}\mathcal{R}_{ip} = \mathcal{Behav}\mathcal{R}|_{\mathcal{Act}_{ip}}$.
- An *interface behaviour* $\mathcal{Behav}_{ag,ip}$ being part of the system $\mathcal{S}$ is a behaviour consisting of all actions in which the agent ag participates, and where the interaction point ip is among the locations. Formally: $\mathcal{Behav}_{ag,ip} = \langle \mathcal{Act}_{ag,ip}, \mathcal{Behav}\mathcal{R}_{ag,ip} \rangle$ is a behaviour s. t. $ag \in \mathcal{AG}$, $ip \in \mathcal{IP}$, $\mathcal{Act}_{ag,ip} = \{a \mid a \in \mathcal{Act} \wedge ag \in \mathcal{Act}\mathcal{A}(a) \wedge ip \in \mathcal{Act}\mathcal{A}(a)\}$, and $\mathcal{Behav}\mathcal{R}_{ag,ip} = \mathcal{Behav}\mathcal{R}|_{\mathcal{Act}_{ag,ip}}$.
- An *interaction* $\mathcal{Behav}_i$ being part of the system $\mathcal{S}$ is a behaviour with two or more participating agents occurring at a single interaction point that is associated with these agents. Formally: $\mathcal{Behav}_i = \langle \mathcal{Act}_i, \mathcal{Behav}\mathcal{R}_i \rangle$ is a behaviour such that $\mathcal{Act}_i \subseteq \mathcal{Act}$, $\mathcal{Behav}\mathcal{R}_i \subseteq \mathcal{Behav}\mathcal{R}$, $\left| \bigcup_{a \in \mathcal{Act}} \mathcal{Act}\mathcal{A}(a) \cap \mathcal{AG} \right| \geq 2$, $\exists ip \in \mathcal{IP}.\ \forall a \in \mathcal{Act}_i.\ \mathcal{Act}\mathcal{A}(a) \cap \mathcal{IP} = \{ip\}$, and $\forall a \in \mathcal{Act}_i.\ \forall ag \in \mathcal{Act}\mathcal{A}(a) \cap \mathcal{AG}.\ \mathcal{Act}\mathcal{A}(a) \cap \mathcal{IP} \subseteq \mathcal{Arch}\mathcal{A}(ag)$

It is by means of interaction that agents can mutually influence each other. This influence consists of exchange of information. If interactions are considered atomic actions on a high abstraction level, i.e. the set $\mathcal{Act}_i$ is a singleton, we obtain a simplified restriction. On a lower level of abstraction, an interaction may consist of a set $\mathcal{Act}_i$ of more elementary actions, where each such action is the portion a single agent has in the interaction. In general, it depends on the particular model which actions form an interaction. The above restriction requires that these actions are associated with two or more agents, that they occur at the same single interaction point, and that this interaction point is associated with each interacting agent.

Since agent behaviour, interaction point behaviour, interface behaviour and interactions are behaviours, they can be composed as defined in Section 2.3. As a result, we can model a complex behaviour in a modular way and obtain the complete behaviour by composition of behaviours. This also applies to the behaviour of a single component, say, an agent, where the behaviour may be substructured into a number of partial orderings corresponding to different restrictions.

2.5 Refinement and abstraction

The concepts of *agent*, *interaction point*, *action*, *architecture*, *behaviour*, and *system* naturally lead to the dual notions of *refinement* and *abstraction*. In general, it is desirable that the refinement of a single component has no influence on the other components. Only then will it be possible to perform incremental system design and modular verification, which is a prerequisite for the development of large systems. By incremental system design, we mean that we can modify or replace a part of the system without affecting the other parts. Modular verification means that only the modified or replaced parts have to be verified, not the entire system. To allow for

incremental system design and modular verification, we have to make suitable restrictions with respect to architecture and behaviour.

With respect to architectures, we require that agents and interaction points be refined separately. In other words, a single component of the refinement (an agent or interaction point) is uniquely related to a single component of the refined architecture. Also, we require that the number of interaction points an agent is associated with remains the same. These and further architectural constraints can be formalized as follows[6]:

Definition (*architectural refinement*):

- Let $Arch = \langle AG, IP, Arch_A \rangle$ and $Arch' = \langle AG', IP', Arch_A' \rangle$ be architectures. $Arch'$ is an *architectural refinement* of $Arch$ (written "$Arch'$ $refines_{Arch}$ $Arch$") if and only if there is a refinement function $ref_{Arch}: AG \cup IP \rightarrow 2^{AG' \cup IP'}$ such that the following restrictions hold:
  - Each component of $Arch$ is refined, i.e., ref_{Arch} is a total function.
  - The refinement of an agent must include at least one agent. Formally: $\forall ag \in AG.\ ref_{Arch}(ag) \cap AG' \neq \{\}$.
  - The refinement of an interaction point must include at least one interaction point: $\forall ip \in IP.\ ref_{Arch}(ip) \cap IP' \neq \{\}$.
  - Each agent and each interaction point is refined separately, i.e., the refinement is disjoint: $\forall x,y \in AG \cup IP.\ (x \neq y \text{ implies } ref_{Arch}(x) \cap ref_{Arch}(y) = \{\})$.
  - AG' is the set of exactly those agents resulting from the refinement, i.e., $AG' = (\bigcup_{ag \in AG} ref_{Arch}(ag) \cup \bigcup_{ip \in IP} ref_{Arch}(ip)) \setminus IP'$.
  - IP' is the set of exactly those interaction points resulting from the refinement, i.e., $IP' = (\bigcup_{ip \in IP} ref_{Arch}(ip) \cup \bigcup_{ag \in AG} ref_{Arch}(ag)) \setminus AG'$.
  - If an agent $ag \in AG$ is associated with an interaction point $ip \in IP$, then exactly one agent of the refinement of ag must be associated with exactly one interaction point of the refinement of ip. Formally:

$$\forall ip \in IP.\ \forall ag \in AG.\ ip \in Arch_A(ag) \text{ implies}$$
$$\Big(\Big| ref_{Arch}(ip) \setminus AG' \cap \bigcup_{ag' \in ref_{Arch}(ag) \setminus IP'} Arch_A'(ag') \Big| = 1 \text{ and}$$
$$\Big| \{ ag' \in ref_{Arch}(ag) \setminus IP' \mid Arch_A'(ag') \cap ref_{Arch}(ip) \setminus AG' \neq \{\} \} \Big| = 1 \Big)$$

Figure 2.3a shows the graphical representation of a possible refinement of the interaction point ip (compare Figure 2.2a), which on a lower level of abstraction comprises an agent ag that can interact with ag_1, ag_2, and ag_3 through ip_1, ip_2, and ip_3, respectively. It is necessary to introduce interaction points in the refinement, because otherwise the rule of composition of architectures about their explicit introduction would be violated. Also, we notice that the duality between agents and interaction points is nicely carried into the refinement. On the other hand, when moving from the composition of ip_1, ip_2, ip_3, and ag to ip, we obtain an architectural abstraction.

---

6 In [Rei86], a different notion of architectural refinement is introduced, which is based on Petri nets. Here, the refinement of a single component can have an influence on other components. Also, system behaviour can only be introduced on the lowest level of refinement.

Figure 2.3b shows a possible refinement of the agent ag (compare Figure 2.2b), which now consists of agents ag_1 and ag_2 with a common interaction point ip. To retain the external appearance as defined for agent ag, subsets of the agents introduced in the refinement are associated with the external interaction points of ag. As before, the duality between agents and interaction points is nicely carried into the refinement.

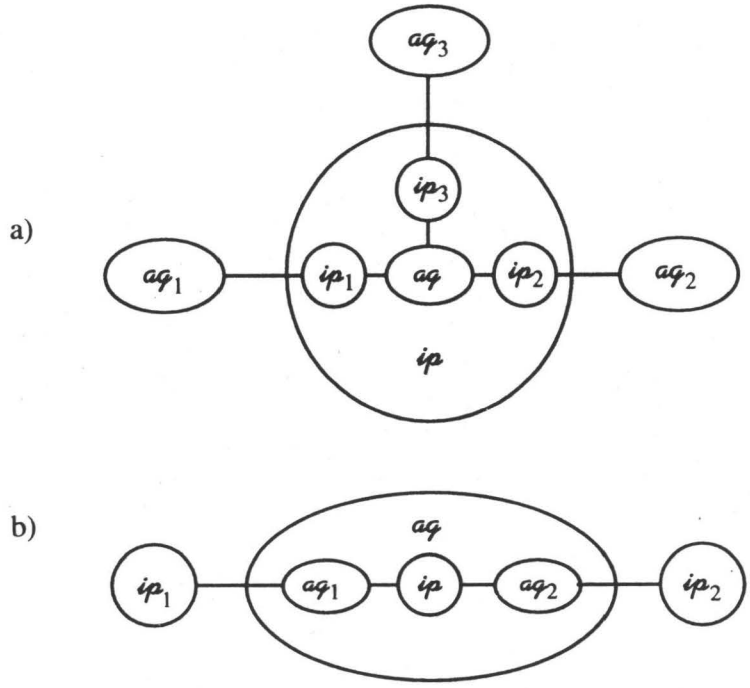

Figure 2.3: Graphical representation of architectural refinement and abstraction

Clearly, the architectures shown in Figure 2.3a and 2.3b are architectural refinements of those shown in Figures 2.2a and 2.2b, respectively. However, the architectures shown in Figure 2.4 do *not* refine the architecture of Figure 2.2b. The reason is that such refinements would have an impact on the remaining components, therefore preventing incremental system design and modular verification. Note that both architectures violate the last restriction of the definition of architectural refinement.

Definition (*architectural abstraction*):

- Let $Arch$ and $Arch'$ be architectures. $Arch'$ is an *architectural abstraction* of $Arch$ (written "$Arch'$ $abstracts_{Arch}$ $Arch$ ") if and only if $Arch$ $refines_{Arch}$ $Arch'$.

This definition formalizes the duality of architectural abstraction and architectural refinement: $refines_{Arch} = abstracts_{Arch}^{-1}$. Note that both relations are preorders on architectures, i.e., reflexive and transitive. Only preorders are suitable refinement relations in the stepwise design of systems. If, for instance, refinement would not be transitive, then for a sequence $Arch_1,...,Arch_n$ of architectures, we could have $Arch_i$ $refines_{Arch}$ $Arch_{i+1}$, for $1 \leq i \leq n-1$, but $Arch_1$ $refines_{Arch}$ $Arch_n$ can not be derived.

Intuitively, a behaviour $Behav'$ refines a behaviour $Behav$, if $Behav'$ is equivalent to or more specific than $Behav$. This requires the refinement relation to be reflexive and transitive. What we can state about behavioural refinement is that actions be refined separately, and that $BehavR$ be respected by the refinement. Without making rather specific assumptions, it is not straightforward to define behaviour refinement more precisely.

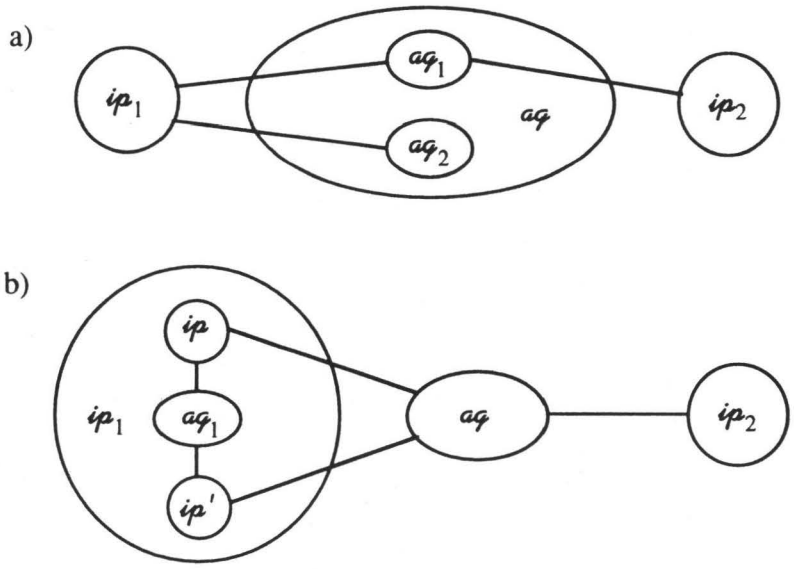

Figure 2.4: Illegal architectural refinements

Definition (*behavioural refinement and abstraction*):

- Let $Behav = \langle Act, BehavR \rangle$ and $Behav' = \langle Act', BehavR' \rangle$ be behaviours. $Behav'$ is a *behavioural refinement* of $Behav$ (written "$Behav'$ $refines_{Behav}$ $Behav$") if and only if there is a refinement function ref_{Act}: $Act \rightarrow 2^{Act'}$ such that the following restrictions hold:
  - Each action of Act is refined, i.e., ref_{Act} is a total function.
  - The refinement of an action must consist of at least one action: $\forall a \in Act.\ ref_{Act}(a) \neq \{\}$.
  - Each action is refined separately, i.e., the refinement is disjoint: $\forall x,y \in Act.$ $(x \neq y$ *implies* $ref_{Act}(x) \cap ref_{Act}(y) = \{\})$.
  - $BehavR$ is respected by the refinement, i.e. $\forall a_1, a_2 \in Act.\ ((a_1,a_2) \in BehavR$ *implies* $\forall a_1' \in ref_{Act}(a_1).\ \forall a_2' \in ref_{Act}(a_2).\ (a_1', a_2') \in BehavR')$.
- Let $Behav$ and $Behav'$ be behaviours. $Behav'$ is a *behavioural abstraction* of $Behav$ (written "$Behav'$ $abstracts_{Behav}$ $Behav$") if and only if $Behav$ $refines_{Behav}$ $Behav'$.

We do not require Act' to be the set of exactly those actions resulting from the refinement. In fact, this would be rather the exception, since further actions will in general be introduced as a consequence of behaviour refinement. In cases where ref_{Act} is the identity function, such a notion may be defined in terms of projection.

With respect to systems, we require that architectural and behavioural refinements exist, and that the action function of the refinement respects the architectural refinement. These constraints can be formally expressed as follows:

Definition (*system refinement and abstraction*):

- Let $\mathcal{S} = \langle Arch, Behav, ActF \rangle$ and $\mathcal{S}' = \langle Arch', Behav', ActF' \rangle$ be systems, $Arch = \langle AG, IP, ArchF \rangle$, $Behav = \langle Act, BehavR \rangle$, $Arch' = \langle AG', IP', ArchF' \rangle$, $Behav' = \langle Act', BehavR' \rangle$. $\mathcal{S}'$ is a *system refinement* of $\mathcal{S}$ (written "$\mathcal{S}'$ $refines_{\mathcal{S}}$ $\mathcal{S}$") if and only if there are *refinement functions* $ref_{Arch}: AG \cup IP \rightarrow 2^{AG' \cup IP'}$, and $ref_{Act}: Act \rightarrow 2^{Act'}$ such that the following restrictions hold:
  - $Arch'$ $refines_{Arch}$ $Arch$ for the refinement function ref_{Arch};
  - $Behav'$ $refines_{Behav}$ $Behav$ for the refinement function ref_{Act};
  - $\forall a \in Act.$
    $(\forall ag \in ActF(a) \cap AG.\ \exists ag' \in AG'.\ \exists a' \in ref_{Act}(a).\ (ag' \in ref_{Arch}(ag) \wedge ag' \in ActF'(a')) \wedge$
    $\forall ip \in ActF(a) \cap IP.\ \exists ip' \in IP'.\ \exists a' \in ref_{Act}(a).\ (ip' \in ref_{Arch}(ip) \wedge ip' \in ActF'(a')))$
- Let $\mathcal{S}$ and $\mathcal{S}'$ be systems. $\mathcal{S}'$ is a *system abstraction* of $\mathcal{S}$ (written "$\mathcal{S}'$ $abstracts_{\mathcal{S}}$ $\mathcal{S}$") if and only if $\mathcal{S}$ $refines_{\mathcal{S}}$ $\mathcal{S}'$.

This definition formalizes the duality of system abstraction and system refinement: $refines_{\mathcal{S}} = abstracts_{\mathcal{S}}^{-1}$. Note that both relations are reflexive and transitive.

3 A case study

We will now give a complete example of how the basic concepts introduced in Section 2 can be represented in a formal description technique such that their meaning is preserved and specialized. The FDT chosen for this purpose is many-sorted first-order temporal logic, which belongs to the category of property-oriented techniques (see [Got93] for further details and references). With respect to Figure 2.1 (Section 2), we select a semantical model $\mathcal{M}$ and define the meaning of so-called requirement specifications with respect to this model. The meaning of requirement specifications in terms of RM-ODP is established by defining a structural relationship.

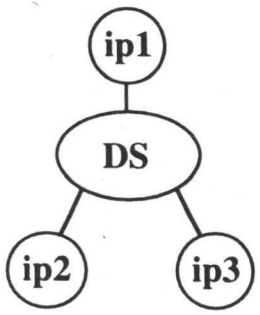

Figure 3.1: Graphical representation of the drink server architecture

A drink server DS takes orders for tea and coffee from customers at interaction point ip1. For each tea order, one cup of tea is served at ip2. For each coffee order, one cup of coffee is served at ip3. The abstract architecture of DS is shown in Figure 3.1. Its internal architecture is not revealed. DS can later be embedded into an environment, for instance, a self-service restaurant or a faculty club, by composition of systems.

Drink servers often work in rounds, i.e., when a drink is ordered, it is served before the next drink can be ordered. More sophisticated drink servers might be able to take new orders while still serving drinks. Such concurrent behaviour should not be excluded.

When the behaviour of DS is specified, no assumptions about the environment are to be made. This means that only the *readiness* of DS to take orders and to serve drinks can be described. Whether orders will be taken when DS is placed into some environment depends on whether they are actually given, and whether drinks will actually be served depends on the readiness of the environment to accept them. We abstract from limitations of resources, i.e., tea and coffee are always available, and from failures.

3.1 A temporal logic

In the temporal logic described below, a requirement specification RS will be a structure ⟨ Arch,Behav ⟩, consisting of its architecture, its behaviours, and its action function. A requirement specification RS will characterize a set of systems $\mathcal{S} = \langle \mathcal{Arch}, \mathcal{Behav}, \mathcal{Act\Phi} \rangle$ as defined in Section 2. Arch = ⟨ AG,IP,ArchF ⟩ will be directly related to the architecture $\mathcal{Arch} = \langle \mathcal{AG}, \mathcal{IP}, \mathcal{Arch\Phi} \rangle$ from Section 2. The other components of a system $\mathcal{S}$ will be characterized by Behav and the choice of atomic formulas.

In many-sorted first-order temporal logic, atomic formulas are formed starting from the following sets of symbols:

- a denumerable list S of symbols called *sorts*;
- a denumerable list V of symbols called *individual variables*; each $x \in V$ is attached to a sort $s \in S$, which is expressed by writing x^s;
- for each integer $n \geq 0$, a denumerable set $F^{(n)}$ of *n-ary function symbols*; each $f \in F^{(n)}$ is associated with sorts $s,s_1,...,s_n \in S$ (written "$f_{s,s_1,...,s_n}$");
- for each integer $n \geq 0$, a denumerable set $R^{(n)}$ of *n-ary relation symbols*; each $r \in R^{(n)}$ is associated with sorts $s_1,...,s_n \in S$ (written "$r_{s_1,...,s_n}$").

With these preparations, the syntax of formulas can be defined as follows:

i) For all $x^s \in V$: x^s is a term of sort s.

ii) For all $n \geq 0, f_{s,s_1,...,s_n} \in F^{(n)}$, and terms $t_1,...,t_n$ of sorts $s_1,...,s_n$: $f_{s,s_1,...,s_n}(t_1,...,t_n)$ is a term of sort s.

iii) For all $n \geq 0$, $r_{s_1,...,s_n} \in R^{(n)}$, and terms $t_1,...,t_n$ of sorts $s_1,...,s_n$: $r_{s_1,...,s_n}(t_1,...,t_n)$ is a formula.

iv) Let φ be a formula, x^s be an individual variable, then $\forall x^s.\varphi$ and $\exists x^s.\varphi$ are formulas.

v) Let φ be a formula, then $\neg \varphi$ is a formula.

vi) Let φ_1, φ_2 be formulas, then $(\varphi_1 \wedge \varphi_2)$, $(\varphi_1 \vee \varphi_2)$, $(\varphi_1 \supset \varphi_2)$, $(\varphi_1 \equiv \varphi_2)$ are formulas.

vii) Let φ be a formula, then $\Box\,\varphi$ and $\Diamond\,\varphi$ are formulas.

viii) Let φ be a formula, then $[\varphi]$ (read "action of type φ") is a formula.

ix) Let φ be a formula, then $\#[\varphi]$ (read "number of actions of type φ") is a term of sort $\mathbb{N}_0$.

The semantics of the temporal operators is defined with respect to a model $\mathcal{M} = \langle E, Q, \Sigma, Q_0 \rangle$, where

- $E = (E_1,...,E_n)$ is a family of non-empty sets of objects (containing agents, interaction points, natural numbers, etc.);
- Q is a set of states, where each state is given by a set F of functions and a set R of relations on E (the states are not states in the ordinary sense with state variables, but contain only what will be necessary to characterize architecture and behaviour);
- $Q_0 \subseteq Q$ is a non-empty set of initial states;
- $\Sigma \subseteq Q^\omega$ is a set of infinite state sequences $\sigma = \langle \sigma_0\sigma_1...\sigma_n... \rangle$ with initial states from Q_0, i.e., $\sigma_0 \in Q_0$ for all $\sigma \in \Sigma$;

Formulas of many-sorted first-order logic are interpreted in a model $\mathcal{M}$ by associating *for each state* $\sigma_j \in Q$ sort symbols with sets of objects, function symbols with functions, relation symbols with relations as follows:

- to every sort symbol $s \in S$, a set $D^s \in \{E_1,...,E_n\}$ is attached; for notational convenience, we will use the same identifiers as sort symbols and to refer to the attached object sets, i.e.: $E_i \in S$ and $D^{E_i} = E_i$;
- for each integer $n \geq 0$: to each n-ary function symbol $f_{s,s_1,...,s_n} \in F^{(n)}$, a function f: $D^{s_1} \times ... \times D^{s_n} \rightarrow D^s$ is attached;
- for each integer $n \geq 0$: to each n-ary relation symbol $r_{s_1,...,s_n} \in R^{(n)}$, a relation $r \subseteq D^{s_1} \times ... \times D^{s_n}$ is attached.

The propositional operators ($\neg$, $\wedge$, $\vee$, $\supset$, $\equiv$), existential and universal quantification ($\exists$, $\forall$) are interpreted as usual. The semantics of the temporal operators and the function $\#[\varphi]$ are defined with respect to the satisfaction relation $\models$. For a model $\mathcal{M}$, $\models$ is a relation between $\mathcal{M}$, a sequence $\sigma \in \Sigma$, a position j and a formula φ (written "$\mathcal{M},(\sigma,j) \models \varphi$").

i) $\mathcal{M},(\sigma,j) \models \Box\,\varphi$ iff $\forall k \geq j.\ \mathcal{M},(\sigma,k) \models \varphi$

ii) $\Diamond\,\varphi =_{Df} \neg\,\Box\,\neg\varphi$

iii) $\mathcal{M},(\sigma,j) \models [\varphi]$ iff $\mathcal{M},(\sigma,j) \models \varphi$ and ($j>0$ implies $\mathcal{M},(\sigma,j-1) \not\models \varphi$)

iv) $\mathcal{M},(\sigma,j) \models \#[\varphi] =_{Df}$

$$\begin{cases} 0 & \text{if } j = 0 \text{ and } \mathcal{M},(\sigma,j) \models \neg[\varphi] \\ 1 & \text{if } j = 0 \text{ and } \mathcal{M},(\sigma,j) \models [\varphi] \\ \mathcal{M},(\sigma,j-1) \models \#[\varphi] & \text{if } j > 0 \text{ and } \mathcal{M},(\sigma,j) \models \neg[\varphi] \\ \mathcal{M},(\sigma,j-1) \models \#[\varphi] + 1 & \text{if } j > 0 \text{ and } \mathcal{M},(\sigma,j) \models [\varphi] \end{cases}$$

When we use temporal logic to characterize a system, we require that a specification must hold in the *initial* state of execution (properties holding throughout the execution can be defined using the 'henceforth' operator '$\Box$'). To express this formally, we use the notion of initial validity. A formula φ is *initially-satisfied in a model $\mathcal{M}$ for a sequence* $\sigma \in \Sigma$, written $\mathcal{M},\sigma \models_i \varphi$, iff $\mathcal{M},(\sigma,0) \models$

φ is true. φ is *initially-valid in* $\mathcal{M}$ iff φ is initially-satisfied for all $\sigma \in \Sigma$. Finally, φ is *initially-valid*, written $\models_i \varphi$, iff φ is initially-valid in all models $\mathcal{M}$.

3.2 Specification of the drink server

To specify the drink server, we first explain how some of the basic concepts in Section 2 are represented in the temporal logic in Section 3.1. We introduce sort symbols AG, IP, 2^{IP}, and a function symbol Archf which is associated with sorts 2^{IP} and AG. The intention is to interpret the sort symbols as the set of agents, interaction points and power set of interaction points, and the function symbol as the architecture function. It will then be straightforward to specify architectures.

Next, we decide to use elementary actions that can be composed into interactions. An interaction may occur between two agents at a common interaction point and consists of offer and accept, denoted by abstract operations ! and ?, respectively. It has an interaction type p (order or serve) and an interaction parameter x (tea or coffee). We write "ag.ip.!(p(x))" to denote that the agent ag offers an interaction of type p with parameter value x at interaction point ip. Similarly, "ag.ip.?(p(x))" denotes that entity ag accepts an interaction, where p and x will have values of a previous interaction offer.

The formula "at ag.ip.!(p(x))" holds when the entity ag is prepared to offer an interaction of type p with parameter value x at interaction point ip; "after ag.ip.!(p(x))" holds immediately after completion of the offer. Note that in the first-order framework, at and after are relations. We overload these relations by demanding at ag.ip.!(p(x)) $\supset$ at ag.ip.! (i.e. (ag,ip,!,p,x) $\in$ at implies (ag,ip,!) $\in$ at), so the formula "at ag.ip.!" holds when the agent ag is prepared to offer some interaction at ip. Similarly, after ag.ip.!(p(x)) $\supset$ after ag.ip.!. The formula "at ag.ip.?" holds when ag is prepared to accept an interaction at ip, "after ag.ip.?(p(x))" holds just after ag has accepted p(x) at ip. As above, after ag.ip.?(p(x)) $\supset$ after ag.ip.?.

With the temporal logic in Section 3.1, we can refer to the occurrence of an action of type φ by writing $[\varphi]$. If φ is an atomic formula, then $[\varphi]$ refers to an atomic action, i.e. an action that cannot be refined on the given level of abstraction. Given the set of formulas, this defines the set Act of action types. Together with a particular model $\mathcal{M}$, this determines the set of actions. Also, we can refer to the number of action occurrences of type φ by $\#[\varphi]$. In the following, we will focus on atomic actions only. For the atomic formulas described in the previous paragraph, we also have the action function ActF (defined on action types), since agent and interaction point are explicitly associated with each action type and thus with each action: ActF ([r ag.ip.op(p(x))]) $=_{Df}$ {ag,ip}, where r $\in$ {at,after} and op $\in$ {!,?}.

Based on Section 3.1, we now define a particular temporal logic by filling in sort symbols, function symbols and relation symbols. This logic will then be used to specify the architecture and the behaviour of the drink server.

- AG, IP, 2^{IP}, OP, P, X, $\mathbb{N}_0$ are sort symbols, interpreted as the set of agents, the set and the power set of interaction points, the set of abstract operations, the set of interaction types, the set of parameter values, and the set of natural numbers;
- DS is a constant of sort AG;

- ip1, ip2, ip3 are constants of sort IP;
- ! and ? are constants of sort OP;
- order and serve are constants of sort P;
- tea and coffee are constants of sort X;
- ArchF is a function symbol of arity 1, associated with sorts 2^{IP} and AG;
- at and after are relation symbols of arity 5, associated with sorts AG, IP, OP, P, X; we overload at and after to be also relation symbols of arity 3, associated with sorts AG, IP, OP.

$RS_{DS} = \langle Arch_{DS}, Behav_{DS} \rangle$

$Arch_{DS} = \langle \{DS\}, \{ip1,ip2,ip3\}, ArchF \rangle$ with $ArchF(DS) = \{ip1,ip2,ip3\}$

$Behav_{DS} = \bigwedge_{1 \le i \le 5} DS_i$

DS_1. $\Box \Diamond$ at DS.ip1.?

DS_2. $\Box$ ((#[after DS.ip1.?(order(tea))] > #[after DS.ip2.!(serve(tea))])
$\supset \Diamond$ at DS.ip2.!(serve(tea)))

DS_3. $\Box$ ($\neg$ (#[after DS.ip1.?(order(tea))] > #[after DS.ip2.!(serve(tea))])
$\supset \neg$ at DS.ip2.!(serve(tea)))

DS_4. $\Box$ ((#[after DS.ip1.?(order(coffee))] > #[after DS.ip3.!(serve(coffee))])
$\supset \Diamond$ at DS.ip3.!(serve(coffee)))

DS_5. $\Box$ ($\neg$ (#[after DS.ip1.?(order(coffee))] >#[after DS.ip3.!(serve(coffee))])
$\supset \neg$ at DS.ip3.!(serve(coffee)))

Table 3.1: Specification of the drink server DS

The requirement specification RS_{DS} of the drink server is listed in Table 3.1. The behaviour specification is composed of a number of properties, each stating a restriction on the allowed behaviour of DS. Property DS_1 expresses that DS is ready to take an order at ip1 from time to time. DS_2 states that if there is an unsatisfied tea order, DS will eventually be ready to serve tea at ip2. DS_3 covers the complementary situations, where it is required that DS is not ready to serve tea. DS_4 and DS_5 state analogous requirements in case of coffee orders.

As mentioned before, the requirement specification RS_{DS} characterizes a set of systems $\mathcal{S} = \langle \mathcal{A}rch, \mathcal{B}ehav, \mathcal{A}ct\mathcal{A} \rangle$ (see Section 2). Note that the restrictions on the composition of elementary concepts into more complex structures (architectures, interactions, behaviours) given in Section 2 are observed.

3.3 Refinement of the drink server

In the following design step, the internal architecture of the agent DS is revealed. DS is decomposed into a waiter W, a tea girl TG, a coffee boy CB, and internal interaction points ip4 and ip5 (see Figure 3.2). The behaviour of these agents and the semantics of the interaction points shall be defined such that the resulting specification $RS_{DS'}$ refines RS_{DS}.

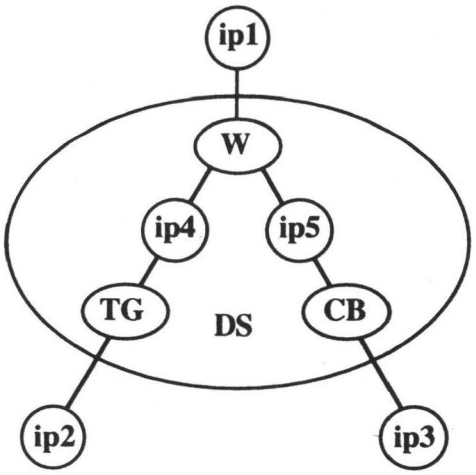

Figure 3.2: Refined architecture of the drink server

Informally, the waiter takes orders for tea and coffee from customers at interaction point ip1. If tea is ordered, the waiter asks the tea girl at ip4 to serve tea. If coffee is ordered, the waiter asks the coffee boy at ip5 to serve coffee. When asked to serve tea, the tea girl serves tea at ip2. When asked to serve coffee, the coffee boy serves coffee at ip3.

As before, we define a particular temporal logic by filling in sort symbols, function symbols and relation symbols:

- AG', IP', $2^{IP'}$, OP', P', X', and $\mathbb{N}_0$ are sort symbols, interpreted as the set of agents, the set and the power set of interaction points, the set of abstract operations, the set of interaction types, the set of parameter values, and the set of natural numbers;
- W, TG, and CB are a constants of sort AG';
- ip1, ip2, ip3, ip4, ip5 are constants of sort IP';
- ! and ? are constants of sort OP';
- order, serve, request and tea, coffee are constants of sorts P' and X', respectively;
- p and x are individual variables of sorts P' and X', respectively;
- ArchF' is a function symbol of arity 1, associated with sorts $2^{IP'}$ and AG';
- at and after are relation symbols of arity 5, associated with sorts AG', IP', OP', P', X'; we overload at and after to be also relation symbols of arity 3, associated with sorts AG', IP', OP'.

The specification $RS_{DS'}$ of the drink server refinement is listed in Table 3.2. The behaviour specification is composed of a number of properties stating restrictions on the allowed behaviour of W, and TG. The specification of the coffee boy CB is very similar to the specification of TG and therefore omitted.

$RS_{DS'} = \langle Arch_{DS}, Behav_{DS}, ActF_{DS} \rangle$

$Arch_{DS'} = \langle \{W,TG,CB\}, \{ip1,ip2,ip3,ip4,ip5\}, ArchF' \rangle$ with

ArchF'(W) = {ip1,ip4,ip5}, ArchF'(TG) = {ip2,ip4}, and ArchF'(CB) = {ip3,ip5})

$Behav_{DS'} = \bigwedge_{1 \le i \le 5} W_i \wedge \bigwedge_{1 \le i \le 3} TG_i \wedge \bigwedge_{1 \le i \le 3} CB_i \wedge \bigwedge_{1 \le i \le 3} ip4_i \wedge \bigwedge_{1 \le i \le 3} ip5_i$

W_1. $\Box \Diamond$ at W.ip1.?

W_2. $\Box$ ((#[after W.ip1.?(order(tea))] > #[after W.ip4.!(request)])
$\supset \Diamond$ at W.ip4.!(request))

W_3. $\Box$ ($\neg$ (#[after W.ip1.?(order(tea))] > #[after W.ip4.!(request)])
$\supset \neg$ at W.ip4.!(request))

W_4. $\Box$ ((#[after W.ip1.?(order(coffee))] > #[after W.ip5.!(request)])
$\supset \Diamond$ at W.ip5.!(request))

W_5. $\Box$ ($\neg$ #[after W.ip1.?(order(coffee))] > #[after W.ip5.!(request)]
$\supset \neg$ at W.ip5.!(request))

TG_1. $\Box \Diamond$ at TG.ip4.?

TG_2. $\Box$ ((#[after TG.ip4.?(request)] > #[after TG.ip2.!(serve(tea))])
$\supset \Diamond$ at TG.ip2.!(serve(tea)))

TG_3. $\Box$ ($\neg$(#[after TG.ip4.?(request)] > #[after TG.ip2.!(serve(tea))])
$\supset \neg$ at TG.ip2.!(serve(tea)))

$ip4_1$. $\Box \forall p,x.$ (#[after W.ip4.!(p(x))] $\geq$ #[after TG.ip4.?(p(x))])

$ip4_2$. $\Box$ (at W.ip4.! $\wedge$ at TG.ip4.? $\supset \Diamond$ [after W.ip4.!] $\wedge \Diamond$ [after TG.ip4.?])

$ip4_3$. $\Box$ (at TG.ip4.? $\wedge$ #[after W.ip4.!] > #[after TG.ip4.?] $\supset \Diamond$ [after TG.ip4.?])

Table 3.2: Specification of the drink server refinement

In addition to properties restricting the behaviour of the system's agents, we have properties defining the semantics of interaction points ip4 and ip5 (see [Got92d], [Got93]). Property $ip4_1$ determines that interactions p(x) accepted by TG at ip4 must have been previously offered by W at ip4. Recall that an interaction is modeled to consist of offer (denoted by !) and acceptance (denoted by ?), p is the interaction type, and x denotes a parameter value. $ip4_1$ ensures that ip4 does not create, duplicate or corrupt interactions. $ip4_2$ requires that if W is prepared to offer and TG is prepared to accept an interaction, then both agents will eventually proceed. $ip4_3$ states that if TG is prepared to accept, and more interactions have been offered than accepted, then TG will

eventually proceed. The semantics of ip5 is analogous to that of ip4 and therefore not listed in Table 3.2.

The specification $RS_{DS'}$ characterizes a set of systems $\mathcal{S}' = \langle Arch', Behav', Act\mathcal{T}' \rangle$ (see Section 2). Note that the restrictions on the composition of elementary concepts into more complex structures (architectures, interactions, behaviours) given in Section 2 are observed. As in the specification of DS, we define ActF' ([r' ag'.ip'.op'(p'(x'))]) $=_{Df}$ {ag',ip'}, where r $\in$ {at,after} and op $\in$ {!,?}.

As discussed earlier, a system $\mathcal{S}'$ refines a system $\mathcal{S}$, if $\mathcal{S}'$ is equivalent to or more specific than $\mathcal{S}$. To apply this definition to requirement specifications, we define a corresponding relation between RS' and RS, called "*refines*$_{rep}$", such that RS' *refines*$_{rep}$ RS implies that the systems characterized by RS' refine the systems characterized by RS. For the architecture, the relationship is straightforward. For the behaviour part, we define a *representation function* rep mapping the behaviour specification of RS to a formula on the abstraction level of RS'. It is then sufficient to show that the result of this mapping is logically implied by the behaviour of RS', which leads to the following definition:

- Let RS = $\langle$ Arch,Behav $\rangle$ and RS' = $\langle$ Arch',Behav' $\rangle$ be requirement specifications. RS' is a *refinement of* RS *under the representation function* rep (written "RS' *refines*$_{rep}$ RS") if and only if the following is satisfied[7]:
  - Arch' *refines*$_{Arch}$ Arch
  - $\models_i$ Behav' $\supset$ rep(Behav)

With the refinement function *ref*$_{Arch}$(DS) = {W,TG,CB,ip4,ip5}, *ref*$_{Arch}$(ip1) = {ip1}, *ref*$_{Arch}$(ip2) = {ip2}, and *ref*$_{Arch}$(ip3) = {ip3}, it follows that Arch' *refines*$_{Arch}$ Arch holds (for the definitions of *refines*$_{Arch}$ and *ref*$_{Arch}$, see *refines*$_{Arch}$ and *ref*$_{Arch}$ in Section 2.5). In particular, this architectural refinement respects the restriction that if an agent ag $\in$ AG is associated with an interaction point ip $\in$ IP, then exactly one agent of the refinement of ag must be associated with exactly one interaction point of the refinement of ip. Thus, the external appearance of the system characterized by RS_{DS} is preserved in the refinement.

To prove $\models_i$ Behav' $\supset$ rep(Behav), we define the representation function rep shown in Table 3.3. This function is defined recursively, following the formation rules of formulas. In particular, each atomic formulas of the logic applied to RS is mapped to a formula of the logic applied to RS'. From rep, it can be derived that an atomic action of RS corresponds to an atomic action of RS', i.e., we have a one-to-one-relationship. Also, non-atomic actions of RS are mapped to actions of RS' via rep. Thus, rep gives us the action refinement introduced in Section 2.5. It is clear that the restrictions on action refinement are observed, i.e., each atomic action of RS is refined, and each action is refined separately. In addition, the action function of the refinement respects the architectural refinement (compare Section 2.5).

Defining rep is a crucial step in the verification process, because it can cause bad results. Therefore, rep has been kept simple. In case of the drink server, it is then straightforward to prove that Behav' $\supset$ rep(Behav) is initially-valid.

---

7 *Refinement* as defined here has been termed *conformance* in [Got93].

rep (at DS.ip1.?)	= at W.ip1.?
rep (after DS.ip1.?(p(x)))	= after W.ip1.?(p(x))
rep (at DS.ip2.!(p(x))	= at TG.ip2.!(p(x))
rep (after DS.ip2.!(p(x))	= after TG.ip2.!(p(x))
rep (at DS.ip3.!(p(x))	= at CB.ip3.!(p(x))
rep (after DS.ip3.!(p(x))	= after CB.ip3.!(p(x))
rep ($\neg \varphi$)	= $\neg$ rep(φ)
rep ($\varphi_1 \wedge \varphi_2$)	= rep (φ_1) $\wedge$ rep(φ_2)
rep ($\forall x.\ \varphi_1$)	= $\forall x.$ rep (φ)
rep ($\Box\ \varphi$)	= $\Box$ rep(φ)
rep ([φ])	= [rep (φ)]
rep (#[φ])	= #[rep (φ)]

Table 3.3: The representation function rep

4 Conclusion

In this paper, we have made an effort at clarifying several basic ODP concepts. Since the definitions given here may affect the standardization process of ODP as a whole, the concepts have been chosen and defined very carefully. Also, they have been kept very general in order to allow for a broad spectrum of representations and specializations in formal descriptions. The intention of this work is to lay the grounds for a more substantial discussion about the meaning of basic ODP concepts. It is expected that in the course of this discussion, some of the concepts treated here will be specialized, and further concepts will be added.

Based on the results of this work, FDTs currently considered for the area of ODP should be evaluated. For each FDT, it should be investigated what basic concepts can be formally expressed in that language, and how this could be done. Every choice will have to respect the meaning of the basic ODP concepts, and may specialize their meaning where necessary. As a result, specifications written in different FDTs or at different design stages should become better comparable. As a further result, this should improve the basis for the development of large systems with a variety of components specified in different FDTs, and for their verification.

Acknowlegements. Special thanks go to C. Andrae, J. Bredereke, K. Madlener, and my colleagues in the Computer Science Department of the University of Kaiserslautern for valuable comments and discussions.

References

[Got92d] Gotzhein, R.: Formal Definition and Representation of Interaction Points, Computer Networks and ISDN Systems 25 (1992) 3-22

[Got93] Gotzhein, R.: Open Distributed Systems - On Concepts, Methods and Design from a Logical Point of View, Vieweg Wiesbaden, 1993

[Gri89] van Griethuysen, J. J.: Open Distributed Processing, in: E. Brinksma, G. Scollo, C. Vissers (eds.), Protocol Specification, Testing, and Verification IX, Proceedings, June 6-9, 1989

[ISO93,1] ISO/IEC JTC1/SC21: Information Technology - Open Distributed Processing - Basic Reference Model of Open Distributed Processing - Part 1: Overview and Guide to Use, ISO/IEC 10746-1, 1993

[ISO93,2] ISO/IEC JTC1/SC21: Information Technology - Open Distributed Processing - Basic Reference Model of Open Distributed Processing - Part 2: Descriptive Model, ISO/IEC 10746-2, 1993

[ISO93,3] ISO/IEC JTC1/SC21: Information Technology - Open Distributed Processing - Basic Reference Model of Open Distributed Processing - Part 3: Prescriptive Model, ISO/IEC 10746-3, 1993

[ISO93,4] ISO/IEC JTC1/SC21: Information Technology - Open Distributed Processing - Basic Reference Model of Open Distributed Processing - Part 4: Architectural Semantics, ISO/IEC 10746-4, 1993

[Rei86] Reisig, W.: Petri Nets in Software Engineering, in: W. Brauer, W. Reisig, G. Rozenberg (eds.), Petri Nets: Applications and Relationships to Other Models of Concurrency, Lecture Notes in Computer Science 255, 1986, pp.63-96