2002 Society for Design and Process Science
Printed in the United States of America

ON ARCHITECTURAL SUPPORT FOR BEHAVIOUR REFINEMENT IN DISTRIBUTED SYSTEMS DESIGN

Dick Quartel
Luís Ferreira Pires
Marten van Sinderen
Centre for Telematics and Information Technology, University of Twente
7500 AE Enschede, the Netherlands

During the top-down design of distributed systems, abstract designs have to be replaced by more concrete designs, which add details that define how these systems can be implemented using available building blocks. Behaviour refinement is a design operation in which abstract behaviours are replaced by more concrete behaviours. Methods that guide and enforce the correctness of these replacements are necessary. This paper presents a set of methods to perform behaviour refinement, based on a careful consideration of the architectural concepts of action and causality relation. Correctness is enforced by validation of the conformance relation between an abstract and a concrete behaviour. Rules are provided to determine whether a concrete behaviour conforms to an abstract behaviour.

1. Introduction

Systematic design methodologies for distributed systems have to be based on precise design concepts and guidelines on how to combine and manipulate these concepts (Quartel, *et al*., 1997, Rechtin, 1992, van Sinderen, *et al*., 1995). The formal support to these design methodologies must be developed based on these combinations and manipulations.

Experience with formal methods shows that their inability to support the manipulation of design concepts in a satisfactory way obstructs the acceptance of these methods. In particular the FDT LOTOS (Bolognesi, *et al*., 1995), despite its sound mathematical basis and its (limited) support for behaviour refinement, has failed so far to be introduced in large scale distributed software design for similar reasons (Vissers, *et al*., 1993).

During the design process of distributed systems, we may replace abstract designs by more concrete designs. We consider the relation between an abstract design and a more concrete design based on the assumption that an abstract design is a prescription for implementation. An abstract design prescribes *what* should be implemented, while a more concrete design prescribes *how* this abstract design should be implemented. The notions of abstract design and concrete design are relative, since a more concrete design may be considered as an abstract design in a next design step.

Behaviour refinement is a design operation in which an abstract behaviour is replaced by a more concrete behaviour. A conformance relation defines which concrete behaviours are valid refinements (implementations) of the abstract behaviour. This conformance relation should guarantee that what is prescribed in the original abstract behaviour is indeed preserved by the more concrete behaviour.

This paper presents design methods for behaviour refinement, which are based on precise manipulations of design concepts for behaviour definition. These design methods can be used in the design process of distributed systems, in particular distributed software applications, to enforce correctness and precision. The approach taken in this paper is precise, although the formal semantics of the design concepts and design methods are not explicitly indicated in the paper. The work presented in this paper can be seen as pre-formal, i.e., as an establishment of a sound architectural (conceptual) basis for the formal support that has been presented in (Quartel, 1998).

This paper is further structured as follows: section 2 gives an overview of the design process for distributed (software) systems, section 3 presents the design concepts used in this paper, section 4 discusses behaviour refinement in detail, section 5 defines a method to abstract from actions that are inserted during behaviour refinement, section 6 defines a method for handling actions that are refined by activities that can successfully terminate in many (possibly alternative) ways, and section 7 illustrates some of the methods presented in this paper with an example design. Section 8 draws some conclusions.

2. Distributed systems development process

This section gives an overview of a systematic development process for distributed systems. This overview aims at stressing the role and importance of design operations for behaviour refinement in the design trajectory of such systems.

2.1. Entity and behaviour domains

In most design methodologies for distributed systems design one can recognize the following concepts:

- *(functional) entities*: logical or physical parts of a system;
- *actions*: units of activity performed by an entity;
- *interactions*: units of activity performed by multiple entities in cooperation;
- *(inter)action points*: locations where (inter)actions occur.

It follows from the identification of entities on one hand, and actions and interactions on the other hand, that the design of a distributed system can be represented from two distinct but related domains:

- *entity domain*, where the entities and their interconnection structure are defined;
- *behaviour domain*, where the actions and interactions performed by entities are properly defined, in terms of the behaviour of these entities.

A design at an abstraction level consists of a collection of interconnected entities and their corresponding behaviours.

2.2. Design trajectory

In a top-down design trajectory, the design of the distributed system is manipulated in successive design steps, which make it possible for designers to move from an abstraction level to a more concrete one. Each design step brings the resulting design closer to the implementation, in terms of designs that can be more easily mapped onto concrete components.

An entity may represent, at a certain abstraction level, a complex structure of more concrete entities. An (inter)action may also represent, at a certain abstraction level, a complex structure of more concrete (inter)actions. Therefore, during the design trajectory of distributed systems one can identify the following design steps (see, e.g., (Sinderen, *et al.*, 1992)):

- *entity decomposition*: applied to a set of entities in a design, it replaces some or each of these entities by multiple entities;

- *interface refinement*: applied to a set of (inter)actions in a design, it replaces some or each of these (inter)actions by multiple (inter)actions.

Since the entity and behaviour domains are related, i.e., an entity should have a behaviour assigned to it, entity decomposition also affects the behaviour domain. This means that the original behaviour of an entity that is decomposed should also be decomposed so that each resulting entity has a corresponding behaviour assigned to it. Methods are necessary to guarantee the correctness of design operations in which behaviours are decomposed.

Fig. 1 shows a simple example, consisting of an entity E that is decomposed in two entities E_1 and E_2, which are connected via an interaction point. The notation used for behaviour representation in this example is explained in section 3. We rely on the intuitive understanding of the reader for the time being.

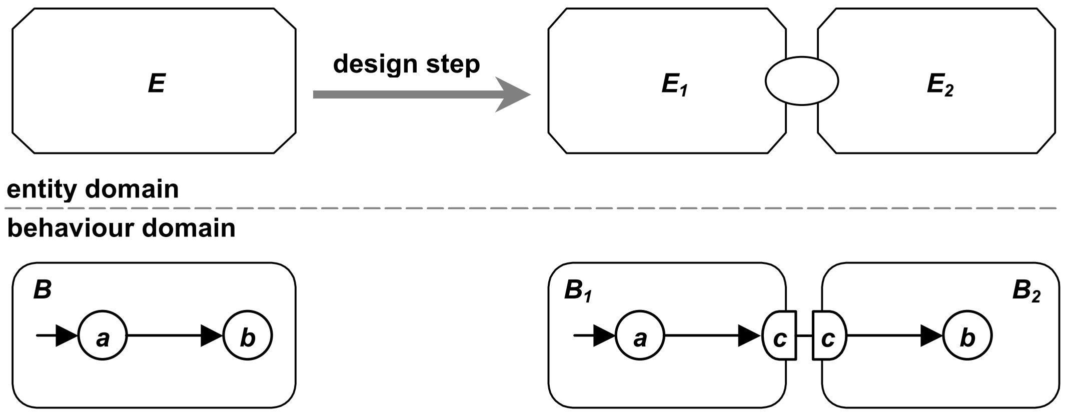


Fig. 1 Simple example of entity decomposition.

In this example, both actions a and b are executed by entity E, such that a is followed by b. In the decomposed design, action a is assigned to E_1 and action b is assigned to E_2. Intuitively one can deduce that an interaction between E_1 and E_2, which we call c, is necessary in order to keep the original relationship between actions a and b. Unfortunately, not all instances of behaviour refinement are so simple as this one in which correctness can be easily and intuitively assessed by hand.

An example of interface refinement is the decomposition of interaction c of Fig. 1 in two successive interactions c_1 and c_2, representing an indication that B_1 is ready to pass some control information to B_2 and a notification that B_2 has accepted this information from B_1. Fig. 2 depicts this example, abstracting from the actual information exchanged.

When applying entity decomposition or interface refinement, one should have methods to guarantee the correctness of the insertion of additional (inter)actions and the replacement of an (inter)action by multiple (inter)actions. The methods presented in this paper support entity decomposition and interface refinement. In order to simplify the presentation of the methods, we only consider refinements of behaviours that contain actions. This does not pose any problem, since an interaction can be seen as an action in case we consider the interaction from the point of view of the common behaviour of the entities participating in the interaction, i.e., abstracting from the individual responsibilities of the involved entities. For example, in Fig. 1 interaction c would be considered as an action of the common behaviour of E_1 and E_2. Extending the methods given in this paper to cover also interactions is therefore straightforward.

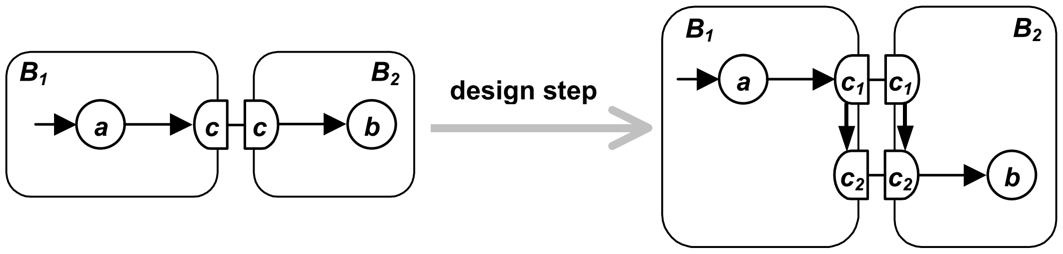


Fig. 2 Simple example of interface refinement.

3. Architectural model

This section briefly presents the basic architectural concepts of action and causality relation used to model monolithic behaviours. Repetitive or interacting behaviours are not considered in this paper. For the complete set of basic concepts for behaviour modelling we refer to (Quartel, *et al*., 1997, van Sinderen, *et al*., 1995, Quartel, 1998, Ferreira Pires, 1994).

3.1. Action

An action models the relevant characteristics of some activity in the real world, abstracting from characteristics that are considered irrelevant at the point in the design process where the action is defined. These relevant characteristics are represented by the following action attributes:

- *information*: the information values that are established in the action, which model the result produced by the activity being modelled;
- *time*: the moment of time when the action occurs, which models the time moment when the result has been produced and is made available to other activities;
- *location*: the location of an action, which models the place where the result of the activity is made available.

Each action is considered to happen only once or not at all. We assume that we can unambiguously refer to an action by using an *action identifier*. The information, time and location attribute values of an action *a* are denoted by the symbols ι_a, τ_a and λ_a, respectively.

3.2. Causality relation

Relations between actions are modelled as a composition of causality relations. A *causality relation* defines for an individual action, called the *result action*, the condition for the occurrence of this action. This condition consists of:

- a *causality condition*, which defines how the occurrence of the result action depends on the occurrences or non-occurrences of other actions;
- *action attribute constraints*, which define how the occurrence of the result action and, possibly, the information, time and location attribute values of the result action, depend on the information, time and location attribute values established by actions in the causality condition;
- a *probability attribute*, which defines the probability of the occurrence of the result action when the causality condition and action attribute constraints are satisfied.

An action can only occur at the time moments in which its causality condition and attribute constraints are satisfied. Once the action occurs, its attribute values can be referred to by other action occurrences that are enabled by the occurrence of this action.

Action attribute constraints are not elaborated in this paper, but introduced by means of an example in section 3.4.

3.3. Causality conditions

Two basic causality conditions have been identified:

- *enabling*: $b \rightarrow a$, the occurrence of b is a condition for the occurrence of a; only after b has occurred, a is allowed to occur and can refer to the attribute values of b. Action b is called an *enabling action* of action a;
- *disabling*: $\neg b \rightarrow a$, the non-occurrence of b is a condition for the occurrence of a; only when b does not occur before nor simultaneously with a, a is allowed to occur. Action b is called a *disabling action* of action a, since the occurrence of b disables the occurrence of a in case a has not occurred already before b.

Combinations of enabling and disabling conditions can be defined to model more complex conditions using the operators *and* ($\wedge$) and *or* ($\vee$). Some examples are:

- $b \wedge \neg c \rightarrow a$, the occurrence of b and the non-occurrence of c are both conditions for the occurrence of a. Action a can only refer to the attribute values of b;
- $b \vee c \rightarrow a$, either the occurrence of b or the occurrence of c is a condition for the occurrence of a. In case both b and c have occurred before a occurs, only one of these actions enables a, such that a is causally related to this action and occurs independently of the other action. In a behaviour execution, action a can only refer to the action that enables it, either b or c, but not both. The choice of which action enables a is undefined, since it is sorted out non-deterministically during the execution of the behaviour. In case one wants to allow a to refer to both b and c, the alternative condition $b \wedge c$ should be added to the causality condition of a, i.e., the causality condition would become $b \vee c \vee (b \wedge c) \rightarrow a$.

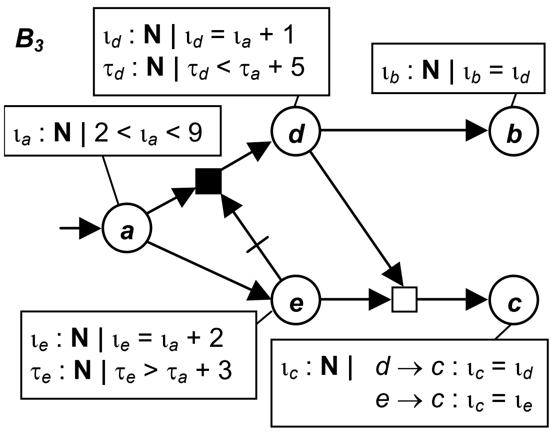
The *and* and *or* operators obey the properties of commutativity and associativity. Furthermore, the *and* operator distributes over the *or* operator, i.e., $\gamma_1 \wedge (\gamma_2 \vee \gamma_3) \approx (\gamma_1 \wedge \gamma_2) \vee (\gamma_1 \wedge \gamma_3)$, where γ represents a causality condition. The inverse property does not hold, i.e., the *or* operator does not distribute over the *and* operator. For example, considering causality relations $b \vee (c \wedge d) \rightarrow a_1$ and $(b \vee c) \wedge (b \vee d) \rightarrow a_2$, action a_1 may refer in an execution either to action b or to actions c and d, while action a_2 may refer also to actions b and c or to actions b and d.

Using the above properties, a condition γ of action a can be defined in disjunctive normal form, i.e., $\gamma = \gamma_1 \vee \gamma_2 \vee ... \vee \gamma_n$, such that each $\gamma_{i\ (1 \le i \le n)}$ consists of a conjunction (*and*) of enabling or disabling conditions. Conditions $\gamma_{i\ (1 \le i \le n)}$ are called *alternative causality conditions* since they represent necessary and sufficient conditions to allow the occurrence of a.

In general, multiple alternative causality conditions can be satisfied simultaneously. We define that the occurrence of a result action is caused by only one of its alternative causality conditions, and, therefore, only depends on the actions that determine the satisfaction of this alternative causality condition. In case multiple alternative conditions are satisfied during a system execution, a decision has to be made at execution time on which alternative condition effectively causes the occurrence of the result action, assuming that the result action occurs. This decision may be (partly) specified, or not. In the latter case, the actual decision is left to the implementer or to mechanisms of the implementation itself.

3.4. Behaviour definitions

A behaviour is defined by a set of causality relations, one relation per action of the behaviour. *Initial actions* are enabled from the beginning of the behaviour, which is represented by the start condition $\surd$. Fig. 3 depicts an example behaviour B_3, which consists of five actions a, b, c, d and e, represented using our graphical and textual notations. Actions a, d and b are defined to occur sequentially, although action e may disable the occurrence of d in case d has not occurred before e occurs. Action d should occur within 5 time units after action a has occurred, while action e is allowed to occur 3 time units after *a* has occurred. In addition, the occurrence of *c* is either enabled by the occurrence of *d* or is enabled by the occurrence of *e*. Therefore, ι_c either refers to ι_d or to ι_e, as represented by the clauses "$d \to c : \iota_c = \iota_d$" and "$d \to c : \iota_c = \iota_e$" in the constraint of *e*, respectively.



B_3
$\iota_d : \mathbf{N} \mid \iota_d = \iota_a + 1$
$\tau_d : \mathbf{N} \mid \tau_d < \tau_a + 5$
$\iota_b : \mathbf{N} \mid \iota_b = \iota_d$
$\iota_a : \mathbf{N} \mid 2 < \iota_a < 9$
d
b
a
e
c
$\iota_e : \mathbf{N} \mid \iota_e = \iota_a + 2$
$\tau_e : \mathbf{N} \mid \tau_e > \tau_a + 3$
$\iota_c : \mathbf{N} \mid \; d \to c : \iota_c = \iota_d$
$e \to c : \iota_c = \iota_e$

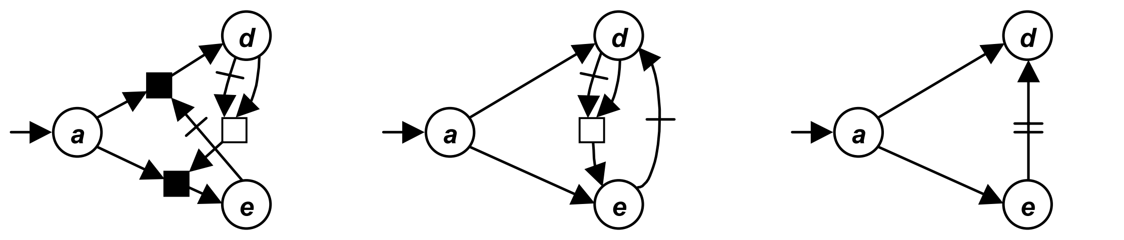
$$
\begin{aligned}
B_3 := \{ \quad & \surd \to a \;\; (\iota_a : \mathbf{N})\,[2 < \iota_a < 9], \\
& a \wedge \neg e \to d \;\; (\iota_d : \mathbf{N}, \tau_d : \mathbf{T})\,[\iota_d = \iota_a + 1, \\
& \qquad\qquad \tau_d < \tau_a + 1], \\
& a \to e \;\; (\iota_e : \mathbf{N}, \tau_e : \mathbf{T})\,[\iota_e = \iota_a + 2, \\
& \qquad\qquad \tau_e > \tau_a + 3], \\
& d \to b \;\; (\iota_b : \mathbf{N})\,[\iota_b = \iota_d], \\
& d \vee e \to c \;\; (\iota_c : \mathbf{N})\,[d \to c : \iota_c = \iota_d \\
& \qquad\qquad e \to c : \iota_c = \iota_e] \; \}
\end{aligned}
$$

Fig. 3 Example behaviour B_3.

Constraints on attribute values are represented between the symbols '[' and ']' in the textual notation, and to the right of the '|' symbol in the graphical notation. The attribute type is represented between the symbols '(' and ')' in the textual notation and to the left of the '|' symbol in the graphical notation. Enabling and disabling conditions are represented by the arrows $\longrightarrow$ and $\not\longrightarrow$, respectively, while their combination using the *and* and *or* operators are graphically represented by the symbols '■' and '□', respectively.

Reciprocal aspect of the disabling condition

The disabling condition $\neg e$ in the causality condition of action *d* defines that action *d* is neither allowed to occur after the occurrence of *e* nor allowed to occur simultaneously with *e*. The non-simultaneous occurrence of actions *e* and *d* is a reciprocal condition, which can be defined implicitly (as in Fig. 3) or explicitly, by adding condition $d \vee \neg d$ to the causality condition of *e*, i.e., $(a \wedge d) \vee (a \wedge \neg d) \to e$. By making this condition explicit, one can define references to the attribute values of action *d* in the causality relation of *e*. Fig. 4(i) depicts the graphical representation of the disabling



(i) basic representation (ii) shorthand 'and' (iii) shorthand disabling relation

Fig. 4 Graphical representation of disabling relation.

relation between *d* and *e* in which the reciprocal condition is made explicit. Fig. 4(ii) depicts the disabling relation, assuming the default *and* interpretation of causality conditions that are represented by distinct arrows pointing to the same action. Fig. 4(iii) depicts the shorthand notation we introduced for a disabling relation between two actions.

3.5. Probability attribute

A probability attribute can be associated with each alternative causality condition of some action *a*, defining the (conditional) probability that *a* occurs when this condition is satisfied. Two variants of the probability attribute are distinguished:

- the *simple probability attribute*: $\pi_a(\gamma)$ defines the probability that result action *a* occurs when assuming that alternative causality condition γ is satisfied;
- the *extended probability attribute*: $\pi^*_a(\gamma)$ defines the probability that result action *a* occurs due to (is caused by) alternative causality condition γ when assuming that γ is satisfied.

Fig. 5 shows examples that are used in the sequel to discuss the two variants of the probability attribute. Behaviours BS_1 and BE_1 represent a choice relation between actions *b* and *c*, which both depend on the occurrence of action *a*. A shorthand notation is used to represent the choice relation, which is composed of the (mutual) disabling conditions $\neg b$ and $\neg c$. Behaviours BS_2 and BE_2 represent a behaviour in which the occurrence of action *f* is enabled either by the occurrence of action *d* or the occurrence of action *e*.

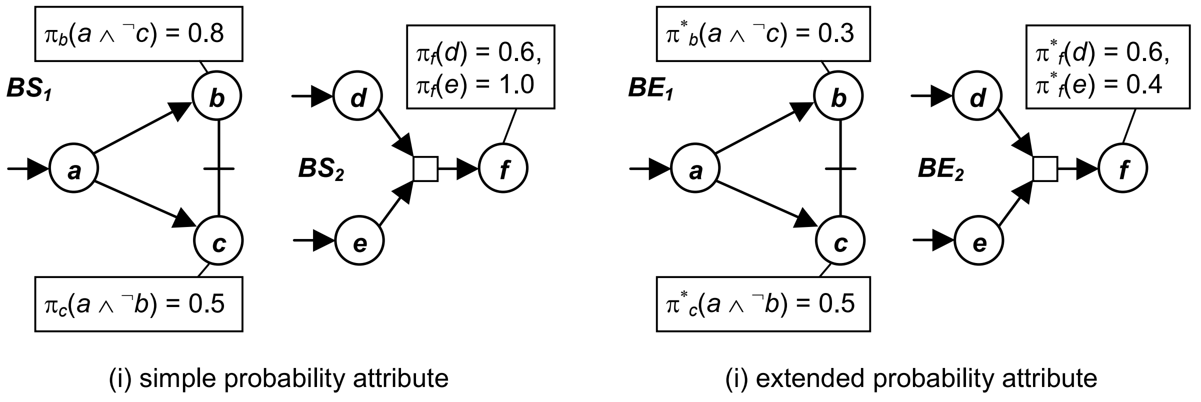


Fig. 5 Examples of probability attributes.

3.5.1. Simple probability attribute

Consider behaviour BS_1 in Fig. 5(i). Probability association $\pi_b(a \wedge \neg c)$ defines the conditional probability that action *b* occurs in an execution, assuming action *a* has occurred and action *c* does not occur before nor simultaneously with *b*. An analogous interpretation applies to $\pi_c(a \wedge \neg b)$. The probability of the occurrence of each of the actions *b* an *c* can not be derived from both probability associations, even in case the probability of the occurrence of *a* is known. This is because the probabilities of the satisfaction of conditions $\neg c$ and $\neg b$ are undefined. One can derive, however, the conditional probability that one of the actions *b* and *c* occurs (assuming *a* has occurred), which is equal to $\pi_b(a \wedge \neg c) + \pi_c(a \wedge \neg b) - (\pi_b(a \wedge \neg c) \times \pi_c(a \wedge \neg b)) = 0.9$. For the reasoning that leads to this formula we refer to (Quartel, 1998).

Consider behaviour BS_2 in Fig. 5(i). Probability associations $\pi_f(d)$ and $\pi_f(e)$ define the conditional probability that action f occurs, assuming actions d and e have occurred, respectively. The conditional probability that f occurs in an execution (assuming d or e, or both occur) is equal to $\pi_f(d) + \pi_f(e) - (\pi_f(d) \times \pi_f(e)) = 1.0$. However, no distinction can be made between the probability that f occurs due to d or due to e, since this probability is undefined.

3.5.2. Extended probability attribute

Consider behaviour BE_1 in Fig. 5(ii). Probability association $\pi^*_b(a \wedge \neg c)$ defines the conditional probability that action b occurs and is related to actions a and c, such that b occurs after a has occurred and c does not occur before nor simultaneously with b, when assuming that a occurs. An analogous interpretation applies to $\pi^*_c(a \wedge \neg b)$. Both probability associations define together the probability of the choice between actions b and c, such that b and c occur in 30 and 50 percent of the executions in which a occurs, respectively. The values of both probability associations must obey the consistency rule $\pi^*_b(a \wedge \neg c) + \pi^*_c(a \wedge \neg b) \leq 1$, which reflects that only one of the two actions in a choice relation can occur. However, one can not define, using the extended attribute, the probability that one of the actions b or c must occur after the occurrence of a, while leaving the distribution between the probabilities of both actions undefined. The latter can be defined using the simple attribute.

Consider behaviour BE_2 in Fig. 5(ii). Probability associations $\pi^*_f(d)$ and $\pi^*_f(e)$ define the conditional probability that action f occurs due to (and is related to) the occurrences of actions d and e, respectively. This allows one to define a distribution between the probabilities that f is related to d and that f is related to e, in the executions in which both d and e occur. However, since no assumptions should be made on the probability of the occurrences of d and e, the following consistency rule must be obeyed: $\pi^*_f(d) + \pi^*_f(e) \leq 1$. Consequently, it is impossible to define, using the extended attribute, that action f must occur (i.e., with probability 1) after d or c occurs, in case the probability of the occurrence of b or c is smaller than 1. The latter can be defined using the simple attribute.

3.5.3. Combined use

The simple and the extended probability attributes are just alternative and complementary ways to define probability, since some behaviour properties can be defined using the simple attribute and others can be defined using the extended attribute. The combined use of both types of probability attributes in a single behaviour definition may therefore be desirable, and is indeed possible in case certain rules defined in (Quartel, 1998) are obeyed.

3.6. Execution relations

The causality conditions of two actions a and b may define none, one or more alternative execution relations between these actions. An *execution relation* defines how two actions depend on each other during a particular behaviour execution. As opposed to execution relations, causality relations define for all possible behaviour executions how the actions depend on each other. The notion of execution relation is introduced to facilitate the definition of the method for behaviour refinement presented in section 5.2.

Two execution relations between actions a and b are distinguished:

- an *enabling relation*, which is graphically represented as $\textcircled{a} \rightarrow \textcircled{b}$. This relation defines an ordering between the occurrences of actions a and b, such that the occurrence of b depends on the preceding occurrence of a. Two variants are distinguished and represented using our textual notation:
  - $\{\surd \to a, a \to b\}$, which defines that a occurs independently of b;

  - $\{\neg b \rightarrow a, a \rightarrow b\}$, which defines that a depends on the non-occurrence of b. This variant has to be used in case the causality condition of b allows b to occur before a, i.e., one alternative causality condition of b contains the disabling condition $\neg a$;
- an *exclusion relation*, which is graphically and textually represented as $\textcircled{a}\!-\!\!\!+\!\!\!-\!\textcircled{b}$ and $\{\neg b \rightarrow a, \neg a \rightarrow b\}$, respectively. This relation defines a conflict (choice) between the occurrences of actions a and b, such that the non-occurrence of b depends on the non-occurrence of a, and vice versa.

For example, considering the behaviour of Fig. 3, the disabling condition $\neg e$ of action d and the implicit condition $d \vee \neg d$ of action e define two alternative execution relations: *(i)* the enabling relation $\{\neg e \rightarrow d, d \rightarrow e\}$ in which d occurs before e, or *(ii)* the exclusion relation $\{\neg e \rightarrow d, \neg d \rightarrow e\}$ in which either e occurs or d occurs.

An execution relation between actions a and b can be defined indirectly via a third action c through the conjunction of an execution relation between a and c and an execution relation between c and b. This is called an *indirect* execution relation. For example, considering the behaviour of Fig. 3, the conjunction of the execution relations $\textcircled{a}\!\longrightarrow\!\textcircled{d}$ and $\textcircled{d}\!\longrightarrow\!\textcircled{b}$ may hold during an execution, which implies that the occurrence of b depends on the preceding occurrence of a.

4. Behaviour refinement

The objective of behaviour refinement is to replace an abstract behaviour by a more concrete behaviour that conforms to this abstract behaviour. Behaviour refinement allows a designer to add more detail to the abstract behaviour, such that the concrete behaviour is closer to the real system behaviour (Quartel, *et al.*, 1999, 1997, 1995).

Actions of abstract behaviours are called *abstract actions* and actions of concrete behaviours are called *concrete actions*. We assume that the occurrence of each abstract action corresponds to the occurrence(s) of one or more concrete actions. This assumption makes it possible to compare the abstract behaviour with the concrete behaviour, by comparing the abstract actions with their corresponding concrete actions. This comparison is needed in order to assess whether the concrete behaviour conforms to the abstract behaviour.

The following conformance requirements are identified:

1. *preservation of relations*: the structure of relations between abstract actions should be preserved by the structure of relations between their corresponding concrete actions;
2. *preservation of attribute values*: attribute values of abstract actions should be preserved by the attributes of their corresponding concrete actions.

Concrete actions that correspond to abstract actions are also called *reference actions*, since they are considered as reference points in the concrete behaviour for assessing conformance. In this paper, abstract actions are denoted by the action identifiers of their corresponding reference actions appended with a prime. Concrete actions that are not reference actions are called *inserted actions*, since they are inserted during behaviour refinement.

4.1. Basic types of refinement

Two basic types of behaviour refinement are identified:

- *causality refinement*, which consists of replacing causality relations between abstract actions by causality relations involving their corresponding concrete actions and some inserted actions;
- *action refinement*, which consists of replacing an abstract action by multiple concrete actions and their causality relations.

Instances of behaviour refinement may consist of one of these basic types of refinement or a combination of both. The essential difference between causality refinement and action refinement is in the way attributes of abstract actions are distributed over the attributes of concrete actions.

4.1.1. Causality refinement

Causality refinement allows one to model the relations between abstract actions in more detail through the introduction of inserted actions. Inserted actions model additional activities in the concrete behaviour, which were not considered (relevant) in the abstract behaviour.

When causality refinement is performed, activities that are originally modelled by the abstract actions are not further detailed in the concrete behaviour. Each abstract action corresponds, therefore, to a *single* reference action. This also implies that the attribute values established by an abstract action should be preserved by its corresponding reference action.

4.1.2. Action refinement

Action refinement allows one to model in more detail an activity that is represented by a single abstract action. The activity is decomposed into multiple related sub-activities. This decomposition is represented by a *concrete action structure*, which consists of multiple concrete actions, one per sub-activity, and their causality relations. An essential characteristic of action refinement is that at least one of the attributes of an abstract action is distributed over the attributes of multiple concrete actions.

A concrete action structure that replaces an abstract action makes its attribute values available through the occurrence of one or more of its final actions. These final actions are the reference actions that ultimately correspond to the abstract action. The following generic cases are distinguished:

- *single final action*: a concrete action structure has a single final action, such that this concrete action structure makes all its attribute values available when this final action occurs;
- *conjunction of final actions*: a concrete action structure has multiple independent final actions, such that this concrete action structure makes all its attribute values available when all these final actions occur;
- *disjunction of final actions*: a concrete action structure has multiple alternative final actions, such that this concrete action structure makes all its attribute values available when one of these final actions occurs;
- any combination of conjunctions and disjunctions of final actions.

Considering the cases of a single final action, a conjunction of final actions and a disjunction of final actions, which are indicated by the abbreviations *(sf)*, *(cf)* and *(df)*, respectively, the preservation of attribute values conformance requirement is interpreted as follows:

- the *information values* of an abstract action should be preserved in:
  - *(sf)* the information attribute of the final action, *(cf)* the union of the information attributes of the final actions, or *(df)* the information attribute of the actual final action that occurs; and
  - the information attributes of inserted actions that can be referred to via the final action(s).

  Information values of the abstract action are established in the final action(s) or are established in the inserted actions that can be referred to via the final action(s);
- the *time moment* of an abstract action should be preserved by: *(sf)* the time moment of the final action, *(cf)* the time moment of the latest final action, or *(df)* the time moment of the actual final action that occurs. The abstract action occurs when all (information) values of the concrete action structure are available;
- the *location* of an abstract action should be preserved by: *(sf)* the location of the final action, *(cf)* the collection of the locations of the final actions, or *(df)* the location of the actual final

action that occurs. The location of an abstract action represents the location(s) of the final action(s);
- the *probability* of an abstract action should be preserved by: *(sf)* the probability that the final action occurs, *(cf)* the probability that all final actions occur, *(df)* the probability that one of the final actions occur. The probability that the abstract action occurs is the probability that the concrete action structure terminates successfully.

4.1.3. Example

The difference between causality refinement, in which a causality relation is replaced by actions and causality relations, and action refinement, in which an action is replaced by a concrete action structure with a single final action, is rather subtle. Fig. 6 illustrates this difference. In this figure, we consider an abstract behaviour that consists of abstract actions a' and b', where action b' establishes two information values v_{b1} and v_{b2}. Abstract action a' corresponds to reference action a and abstract action b' corresponds to reference actions b and b_2.

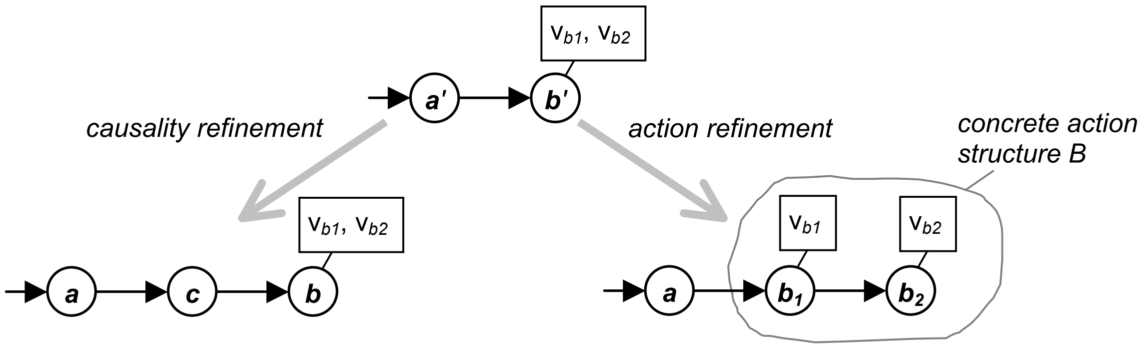


Fig. 6 Difference between causality refinement and action refinement.

Performing causality refinement we can replace the enabling relation between abstract actions a' and b' by two enabling relations through inserting action c, which allows b to indirectly refer to the attribute values of a. Furthermore, reference action b should establish the same attribute values as abstract action b'.

Performing action refinement we can replace abstract action b' by concrete action structure B, consisting of inserted action b_1 and reference action b_2. The attribute values v_{b1} and v_{b2} are established by actions b_1 and b_2, respectively. Reference action b_2 preserves the attribute values of abstract action b', since action b_2 can refer to value v_{b1} of action b_1.

Examples of action refinement with multiple final actions are presented in section 6.

4.2. Use of abstraction

An abstract behaviour can be replaced by different alternative concrete behaviours. Depending on the choice of a concrete behaviour, different concrete actions and their causality relations are added to the abstract behaviour. Since this choice is determined by specific design objectives, behaviour refinement can not be automated in its totality.

In contrast, the abstraction of a concrete behaviour is unique given what we should abstract from. When abstracting from certain concrete actions and their causality relations, the abstraction of this concrete behaviour is completely determined by the remaining concrete actions and their causality relations. Rules can be provided to calculate this abstraction. These rules can, in principle, be automated.

The uniqueness of an abstraction allows one to assess the conformance between an abstract behaviour and a concrete behaviour, by comparing the abstraction of the concrete behaviour with the original abstract behaviour. Therefore, we distinguish the following successive design activities in an instance of behaviour refinement:

1. *delimitation of the abstract behaviour*: we only consider the refinement of behaviours that are influenced by a finite number of abstract actions. For example, in case of recursive behaviours one should identify the finite behaviour parts that are (infinitely) repeated;
2. *refinement of the abstract behaviour into a concrete behaviour*: in this activity we determine how the abstract behaviour is implemented by the concrete behaviour;
3. *determination of the abstraction of the concrete behaviour*: a method to perform this activity is presented below;
4. *comparison of the abstraction of the concrete behaviour with the original abstract behaviour*: both behaviours should comply to a certain correctness relation. If this is not the case, the concrete behaviour is not considered as a correct implementation of the abstract behaviour, such that one must return to design activity 2.

Determining an abstraction of a behaviour

The following steps define a method to determine the abstraction of a concrete behaviour:

1. *identify reference actions and inserted actions in the concrete behaviour*. This means that the identified reference actions have to be considered as:
   - (single) reference actions that are obtained by causality refinement; or
   - groups of reference actions that are formed by grouping the final actions of each concrete action structure obtained by action refinement;
2. *abstract from inserted actions*. We can do this by using the abstraction method presented in section 5;
3. *replace each group of reference actions by an abstract action*. We can do this by using the abstraction method presented in section 6.

4.3. Correctness relations

The conformance between an abstract behaviour and the corresponding concrete behaviour is assessed in terms of a correctness relation between this abstract behaviour and the abstraction of the corresponding concrete behaviour. Depending on the specific conformance requirements, this correctness relation can be:

- an *equivalence relation*, which defines that the concrete behaviour should preserve all behaviour properties of the abstract behaviour; or
- a *partial ordering relation*, which defines that the concrete behaviour should preserve a subset of the behaviour properties of the abstract behaviour.

For example, two alternatives of the preservation of attribute values conformance requirement are considered:

- *strong preservation*: all attribute values that are possible for an abstract action are also possible for the corresponding concrete actions; and
- *weak preservation*: some attribute values that are possible for an abstract action are not possible for the corresponding concrete actions.

Strong preservation can be assessed in terms of an equivalence relation “≈” on the attribute values of two abstract actions, while weak preservation can be assessed in terms of a partial ordering relation

"∠" on the attribute values of two abstract actions. Fig. 7 depicts an example of strong and weak preservation of attribute values. Behaviours B_1' and B_2' represent the abstractions of two alternative refinements B_1 and B_2 of abstract behaviour B'. Since abstract actions b' and b_1' can occur at the same time moments, behaviours B' and B_1 obey the strong preservation of attribute values. In contrast, abstract action b_2' can only occur at a subset of the time moments at which abstract action b' can occur. Consequently, behaviours B' and B_2 obey the weak preservation of attribute values.

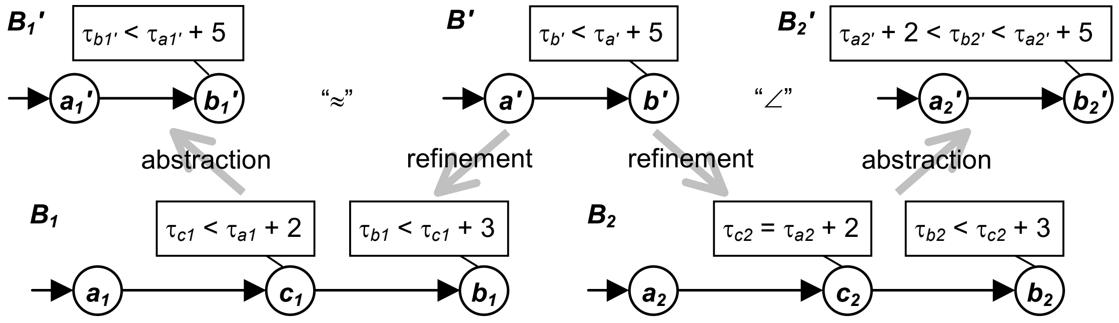


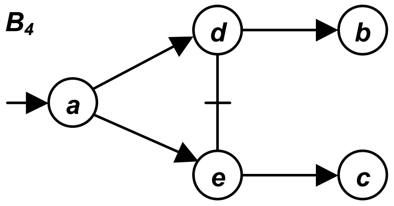
Fig. 7 Strong and weak preservation of attribute values.

5. Abstraction from inserted actions

This section presents a method to deduce the abstract behaviour of a given concrete behaviour, by abstracting from the inserted actions and their influence on the concrete behaviour. This method contains steps and rules that have to be followed in order to abstract from a *single* inserted action (called z). The abstraction of multiple inserted actions can be performed by consecutively abstracting from each single inserted action in any order.

5.1. Causality context of an inserted action

When abstracting from a single inserted action z in a concrete behaviour B, the remaining actions in B are considered as reference actions. The influence of an inserted action z on these reference actions can be delimited to the causality context of z. The *causality context* of z only considers the actions in B that are directly related to z, i.e., *(i)* the actions that have a causality condition which contains action z, and *(ii)* the actions that are defined in the causality condition of z. The causality context of z is denoted by $Con(z)$ and the actions in $Con(z)$ are called the *context actions* of z. In Fig. 8, the example behaviour B_4 has $Con(e) = \{a, d, c\}$ and $Con(a) = \{d, e\}$.



$$B_4 := \{ \quad \surd \rightarrow a,\ d \rightarrow b,\ e \rightarrow c,\ a \wedge \neg e \rightarrow d,\ a \wedge \neg d \rightarrow e \}$$

Fig. 8 Example behaviour B₄.

An inserted action indirectly relates the occurrences of its context actions. Furthermore, an inserted action may allow a context action to indirectly refer to the attributes of another context action. For example in Fig. 8, action *e* defines an indirect execution relation between actions *a* and *c*, such that the occurrence of *c* indirectly depends on the occurrence of *a*. Furthermore, action *c* may refer indirectly to the attributes of action *a*.

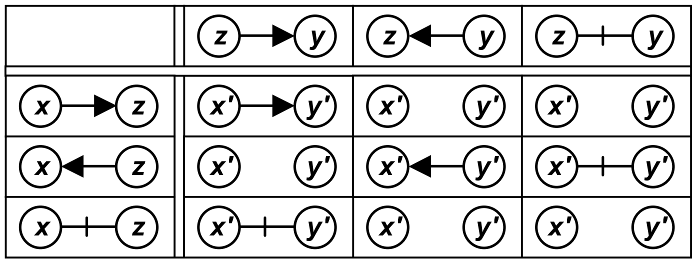
In order to abstract from the influence of an inserted action *z* on the concrete behaviour, one should be able to abstract from indirect execution relations, indirect attribute references and probability associations involving *z*, i.e.:

- alternative causality conditions of context actions that define indirect execution relations between these actions via *z*, should be replaced by equivalent conditions that define these execution relations directly;
- information, time and location attribute constraints of context actions that define indirect references between the attributes of these actions via the attributes of *z*, should be replaced by equivalent constraints that define these references directly;
- probability attribute constraints of context actions that define probability associations involving *z*, should be replaced by equivalent constraints that are defined in terms of probability associations only involving these context actions.

5.2. Abstraction of indirect execution relations

Table 1 shows for each possible indirect execution relation between concrete actions *x* and *y*, the equivalent execution relation between the corresponding abstract actions *x'* and *y'*, respectively, when abstracting from inserted action *z*.

Table 1. Indirect execution relations.



	$z \longrightarrow y$	$z \longleftarrow y$	$z \dashv\vdash y$
$x \longrightarrow z$	$x' \longrightarrow y'$	$x' \quad y'$	$x' \quad y'$
$x \longleftarrow z$	$x' \quad y'$	$x' \longleftarrow y'$	$x' \dashv\vdash y'$
$x \dashv\vdash z$	$x' \dashv\vdash y'$	$x' \quad y'$	$x' \quad y'$

Legend:

$x \longrightarrow z$ represents the enabling relation: {√ → *x*, *x* → *z*} or {¬*z* → *x*, *x* → *z*};

$x \dashv\vdash z$ represents the exclusion relation: {¬*z* → *x*, ¬*x* → *z*};

$x \quad y$ represents independence: {√ → *x*, √ → *z*};

Table 1 is based upon the following abstraction rules:

- *transitivity of enabling*: considering that a concrete action *x* is an enabling condition of inserted action *z*, and *z* is an enabling condition of concrete action *y*, then *x'* is an enabling condition of *y'* and the condition of *y'* is either equal to √ in case the condition of *x* is √, or is equal to ¬*x'* in case the condition of *x* is ¬*z*;

- *inheritance of exclusion*: considering that an inserted action z is an enabling condition of a context action x, the exclusion between z and another context action y is inherited by actions x' and y', i.e., the conditions of x' and y' are the disabling conditions $\neg y'$ and $\neg x'$, respectively.

In the case of indirect execution relations that do not meet the constraints of the above rules, context actions x and y occur independently. Consequently, abstract actions x' and y' occur independently in these cases.

5.2.1. Execution structures

An inserted action may relate different pairs of context actions during the same execution. This is represented by a conjunction of multiple indirect execution relations, one for each pair of related context actions. A conjunction of multiple indirect execution relations is also called an *execution structure*. For example, Fig. 9 represents a possible conjunction of three indirect execution relations defined by behaviour B_4 of Fig. 8, when assuming that action e is an inserted action.

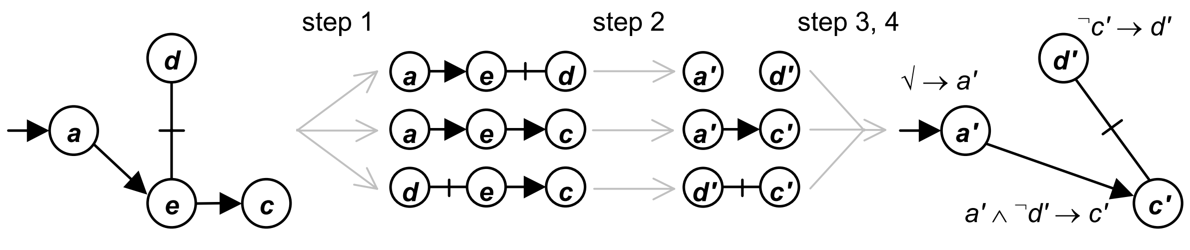


Fig. 9 Execution structure.

A method to determine the abstraction of an execution structure consists of the following steps:

1. identify all indirect execution relations;
2. determine the abstraction of each indirect execution relation using the rules of Table 1;
3. compose the condition of each abstract action as the conjunction of its conditions in the abstractions of step 2;
4. simplify the condition of each abstract action using the following rules: $C \wedge C = C$, $C \vee C = C$ and $\surd \wedge C = C$, where C represents an arbitrary causality condition.

Fig. 9 illustrates the application of this method. The condition of c' is composed as the conjunction of the conditions in the abstractions $\{\surd \to a', a' \to c'\}$ and $\{\neg d' \to c', \neg c' \to d'\}$ of step 2, which gives: $a' \wedge \neg d' \to c'$.

5.2.2. Alternative execution structures

Alternative execution structures are defined when the causality conditions of the inserted action or its context actions define alternative causality conditions using the disjunction operator. Alternative execution structures represent alternative conjunctions of indirect execution relations that may hold during an execution. In principle, each alternative condition doubles the number of alternative execution structures, although identical execution structures may be obtained.

In the example behaviour B_5 of Fig. 10 we assume that action d is an inserted action. Since the disabling relation between actions d and b is composed of the disjunction of the exclusion relation $\{\neg d \to b, \neg b \to d\}$ and the enabling relation $\{\neg d \to b, b \to d\}$, two alternative execution structures are defined by the causality context of d.

Based upon the notion of alternative execution structures, the method to derive the abstract behaviour B' of a concrete behaviour B by abstracting from inserted action z can be defined as follows:

1. identify the causality context of inserted action z, i.e., the concrete actions in *Con(z)*;
2. identify the alternative execution structures between z and its context actions;
3. determine the abstraction of each execution structure using the method of section 5.2.1;
4. compose the condition of each abstract action from the condition of its corresponding concrete action by replacing the sub-conditions that are conditions of this concrete action in one (or more) alternative execution structure(s) by (the disjunction of) the corresponding condition(s) in the abstraction(s) of these execution structure(s) as obtained in step 3;
5. simplify the condition of each abstract action, if possible.

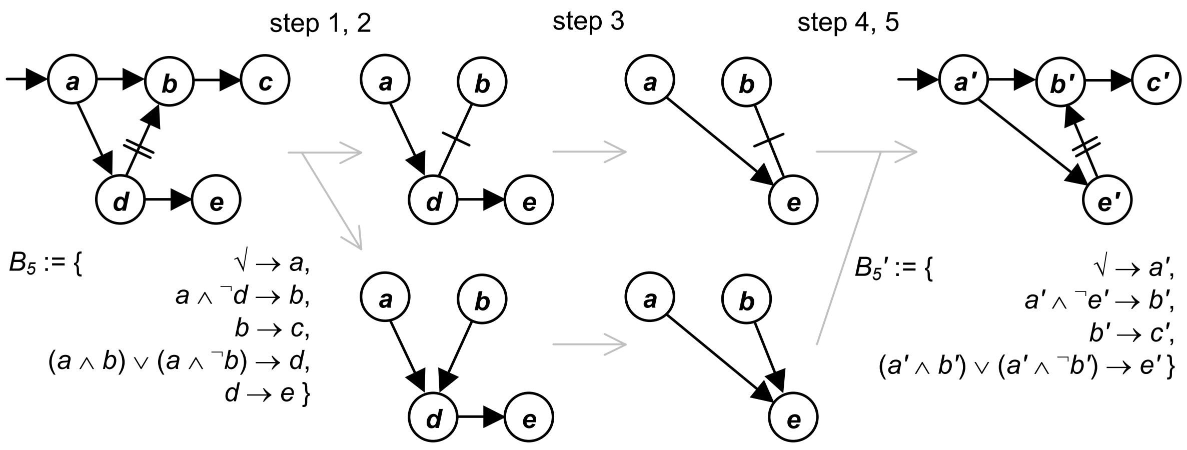
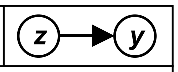
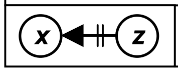
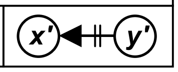

step 1, 2
step 3
step 4, 5
a
b
c
d
e
a'
b'
c'
e'
$B_5 := \{ \quad \surd \rightarrow a,$
$a \wedge \neg d \rightarrow b,$
$b \rightarrow c,$
$(a \wedge b) \vee (a \wedge \neg b) \rightarrow d,$
$d \rightarrow e \}$
$B_5' := \{ \quad \surd \rightarrow a',$
$a' \wedge \neg e' \rightarrow b',$
$b' \rightarrow c',$
$(a' \wedge b') \vee (a' \wedge \neg b') \rightarrow e' \}$

Fig. 10 Example behaviour B_5.

Fig. 10 illustrates the application of these steps. The condition of b' is composed from condition $a \wedge \neg d$ by replacing sub-condition $\neg d$ by the disjunction of the corresponding conditions $\neg e'$ and $\neg e'$ in the abstractions of both execution structures, i.e., $\neg e' \vee \neg e'$, since $\neg d$ is the condition of action b in both execution structures. Furthermore, action name a is substituted by a', and sub-condition $\neg c' \vee \neg c'$ is simplified to $\neg c'$.

The abstraction of behaviour B_2 is based upon a combination of the *transitivity of enabling* and *inheritance of exclusion* abstraction rules. Table 2 depicts this composite abstraction rule, which is called *inversion of causality*, because it replaces the disabling of x by z with the disabling of x' by y', giving the intuitive impression that the causality between z and y was turned around.

Table 2 Inversion of causality.

The rule represented in Table 2 is obtained from Table 1 by taking the disjunction of the execution relations in the intersection of the second column and the second and fourth rows. In general, the abstraction of the conjunction of any two composite execution relations can be obtained in this way.

5.3. References to action attributes

An action can only refer to the attribute values of action occurrences. In case an action does not occur, no attribute values are established. Therefore, indirect attribute references between context actions are only possible in the case of indirect execution relations to which the *transitivity of enabling* abstraction rule applies. The following rules to abstract from references to information, time and location attributes of inserted actions should be applied in combination with this abstraction rule:

- references to information and location attributes of inserted actions should be replaced by their possible values or constraints;
- references to the time attributes of inserted actions should be replaced by their possible values or constraints, taking into account implicit time constraints.

Fig. 11 illustrates the application of both rules. The reference to the information attribute of inserted action *c* is abstracted from by simply substituting the constraint of *c* in the constraint of action *b*. However, a similar substitution in case of the reference to the time attribute of inserted action *c* renders the constraint [$\tau_{b'} < \tau_{a'} + 5$], which is an incorrect abstraction since *b'* happens after *a'*. In general, implicit time constraints imposed by enabling conditions also have to be considered when abstracting from references to the time attribute of inserted actions. In this case, the constraint [$\tau_c = \tau_a + 2$] should also be substituted in the implicit time constraint $\tau_c < \tau_b$, which renders the additional constraint [$\tau_{a'} + 2 < \tau_{b'}$].

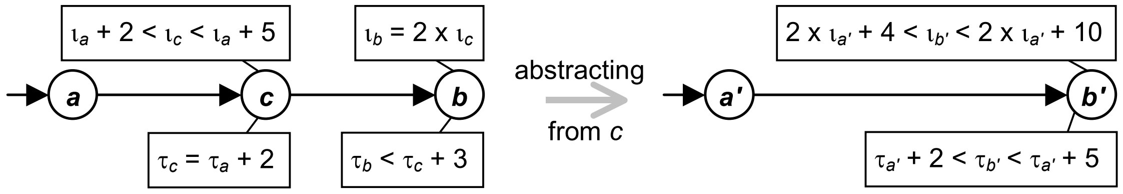

$\iota_a + 2 < \iota_c < \iota_a + 5$
$\iota_b = 2 \times \iota_c$
a
c
b
$\tau_c = \tau_a + 2$
$\tau_b < \tau_c + 3$
abstracting
from c
$2 \times \iota_{a'} + 4 < \iota_{b'} < 2 \times \iota_{a'} + 10$
a'
b'
$\tau_{a'} + 2 < \tau_{b'} < \tau_{a'} + 5$

Fig. 11 Abstraction of indirect information, time and location attribute references.

5.4. Probability attribute references

The probability of the occurrence of an action is determined by the probability of the occurrence of the enabling actions and the probability of the non-occurrence of the disabling actions in its causality condition. In this way, an action refers to the probability attributes of the enabling and disabling actions in its causality condition. Therefore, rules are needed to determine the abstraction of probability associations involving *(i)* enabling condition *z*, and *(ii)* disabling condition $\neg z$. These rules are presented below and illustrated with examples, both for the simple and for extended probability attribute. For the reasoning underlying the rules and their formalisation, we refer to (Quartel, 1998).

The abstraction rules are defined below for some context action *c*. The abstraction of the probability association $\pi_c(\gamma_c)$ is denoted as $\pi_{c'}(\gamma_{c'})$, where $\gamma_{c'}$ is obtained from γ_c using the abstraction method of section 5.2.

5.4.1. Simple probability attribute

The abstraction of a simple probability association $\pi_c(\gamma_c)$ involving inserted action z is defined by the following rules:

1. γ_c contains enabling condition z (i.e., $\gamma_c = z \wedge \gamma$):

   (i) $\pi_{c'}(\gamma_{c'}) = \pi_c(\gamma_c)$, if c also depends indirectly on the occurrence of z

   (ii) $= \pi_z(\gamma_z) \times \pi_c(z \wedge \gamma)$, otherwise

2. γ_c contains disabling condition z (i.e., $\gamma_c = \neg z \wedge \gamma$):

   (i) $\pi_{c'}(\gamma_{c'}) = \pi_c(\gamma_c)$, if - c also depends indirectly on the non-occurrence of z, or

   (ii) - z depends (in)directly on the occurrence of c, or

   (iii) - c depends (in)directly on the non-occurrence of a third action b and b must occur after z

   (iv) $< \pi_c(\gamma_c)$, otherwise

Abstraction rule 1 is explained by the example behaviours in Fig. 12. Behaviour B_6 illustrates the application of rule 1(ii). Action c indirectly depends on the occurrence of action a via inserted action z. When abstracting from z, the probability that c' occurs when enabling condition a' is satisfied must be determined, which is represented by $\pi_{c'}(a')$. This probability association represents the uncertainty that z occurs after a has occurred, and the uncertainty that c occurs after z has occurred. Consequently, $\pi_{c'}(a')$ is determined by the composition of $\pi_z(a)$ and $\pi_c(z)$, such that $\pi_{c'}(a') = \pi_z(a) \times \pi_c(z) = 0.4$.

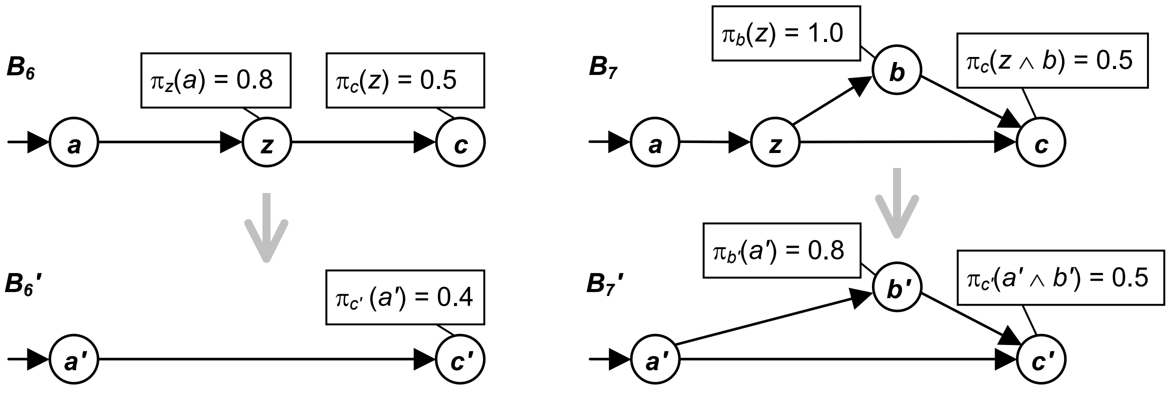


Fig. 12 Abstraction from probability associations involving enabling condition *z*.

Behaviour B_7 illustrates the application of rule 1(i). Action c indirectly depends on the occurrence of inserted action z via action b. In this case, the uncertainty introduced by z on the occurrence of c is already represented by enabling condition b via which c depends on z. This implies that the value of $\pi_{c'}(a' \wedge b')$ is equal to the value of $\pi_c(z \wedge b)$.

Abstraction rule 2 is explained by extending the example behaviour of Fig. 10 with probability attributes, while leaving out action e. Fig. 13 depicts the steps taken to abstract from inserted actions b and d. Furthermore, this example illustrates that we can abstract from these actions in any order. Left to the vertical dashed line we depict two sequences of abstraction steps, one for each alternative execution structure, which abstract first from b and then from d. The corresponding sequences that abstract from b and d in the reverse order is depicted right to the dashed line. Because both b and d are abstracted from, and to perform the integration steps 4 and 5 of the method in section 5.2.2 only once, the execution structures are not delimited to the causality contexts of b and d. Some of the abstraction steps are explained in the sequel.

Step I.1 illustrates abstraction rules 2(iii) and 2(iv). We consider the abstraction of probability association $\pi_d(a \wedge \neg b)$. The uncertainty of the occurrence of d consists of the uncertainty introduced by action d itself, i.e., $\pi_d(a \wedge \neg b)$, and the uncertainty caused by the possible disabling of d by inserted action b. The latter uncertainty is (partly) represented by disabling condition $\neg c'$, since b enables c. The following two cases are distinguished:

- action c occurs in every execution in which b occurs, i.e., $\pi_c(b) = 1.0$. In this case actions c and d completely inherit the choice between actions b and d. This implies that the uncertainty of the occurrence of d due to its possible disabling by b is completely represented by disabling condition $\neg c'$, such that $\pi_{d'}(a' \wedge \neg c') = \pi_d(a \wedge \neg b)$;
- action c may occur in some, but not all executions in which b occurs, i.e., $\pi_c(b) < 1.0$. In this case actions c and d partly inherit the choice between actions b and d. This implies that the uncertainty of the occurrence of d due to its possible disabling by b is only partly represented by disabling condition $\neg c'$, such that $\pi_{d'}(a' \wedge \neg c') < \pi_d(a \wedge \neg b)$.

Step IV.1 illustrates abstraction rule 2(ii). We consider the abstraction of probability association $\pi_b(a \wedge \neg d)$. Since disabling action d depends on the occurrence of b in this execution structure, the uncertainty of the occurrence of b consists solely of the uncertainty introduced by action d itself, such that $\pi_{b'}(a') = \pi_b(a \wedge \neg d)$.

Step III.1 illustrates abstraction rule 2(i). We consider the abstraction of probability association $\pi_c(b)$. Action c indirectly depends on the non-occurrence of inserted action d via enabling condition b, i.e., conditions $\gamma_c = b$ and $\gamma_c = b \wedge \neg d$ are equivalent. In this case, the value of $\pi_{c'}(b')$ is equal to the value of $\pi_c(b)$ (= $\pi_c(b \wedge \neg d)$), since the uncertainty introduced by d and its causality condition is already represented by enabling condition b.

Step III.1 illustrates also abstraction rule 2(iv). We consider the abstraction of probability association $\pi_b(a \wedge \neg d)$. The uncertainty of the occurrence of b consists of the uncertainty introduced by action b itself, i.e., $\pi_b(a \wedge \neg d)$, and the uncertainty caused by the possible disabling of b by inserted action d. Since the amount of the latter uncertainty is undefined, but assumed to be larger than zero, we define that $\pi_{b'}(a') < \pi_b(a \wedge \neg d)$.

Step II.2 concerns the abstraction of inserted action d. Since none of the context actions depends on d, this action can simply be removed without any changes to the context actions and their attributes (except for appending a prime).

The final step integrates the abstractions of the alternative execution structures. This integration is defined as follows: $\{ \surd \vee \surd \rightarrow a'', a'' [\pi_{c''}(a'') < 0.9] \vee a'' [\pi_{c''}(a'') = 0.9] \rightarrow c'' \}$. The resulting abstract behaviour can be simplified to $\{ \surd \rightarrow a'', a'' [\pi_{c''}(a'') < 0.9] \rightarrow c'' \}$, since the constraints $[\pi_{c''}(a'') < 0.9]$ and $[\pi_{c''}(a'') = 0.9]$ are originated from alternative execution structures (having non-zero probability).

Concrete behaviour B_8 in Fig. 13 models the cause of the possible non-occurrence of action c explicitly, in terms of the possible disabling of inserted action b by inserted action d. Abstract behaviour B_8'' abstracts from this cause, and models the uncertainty of the occurrence of c'' by means of the probability attribute of c''. We denote the type of refinement illustrated in Fig. 13 as, *making uncertainty explicit*.

5.4.2. Extended probability attribute

The abstraction of an extended probability association $\pi^*_c(\gamma_c)$ involving inserted action z is defined by the following rules:

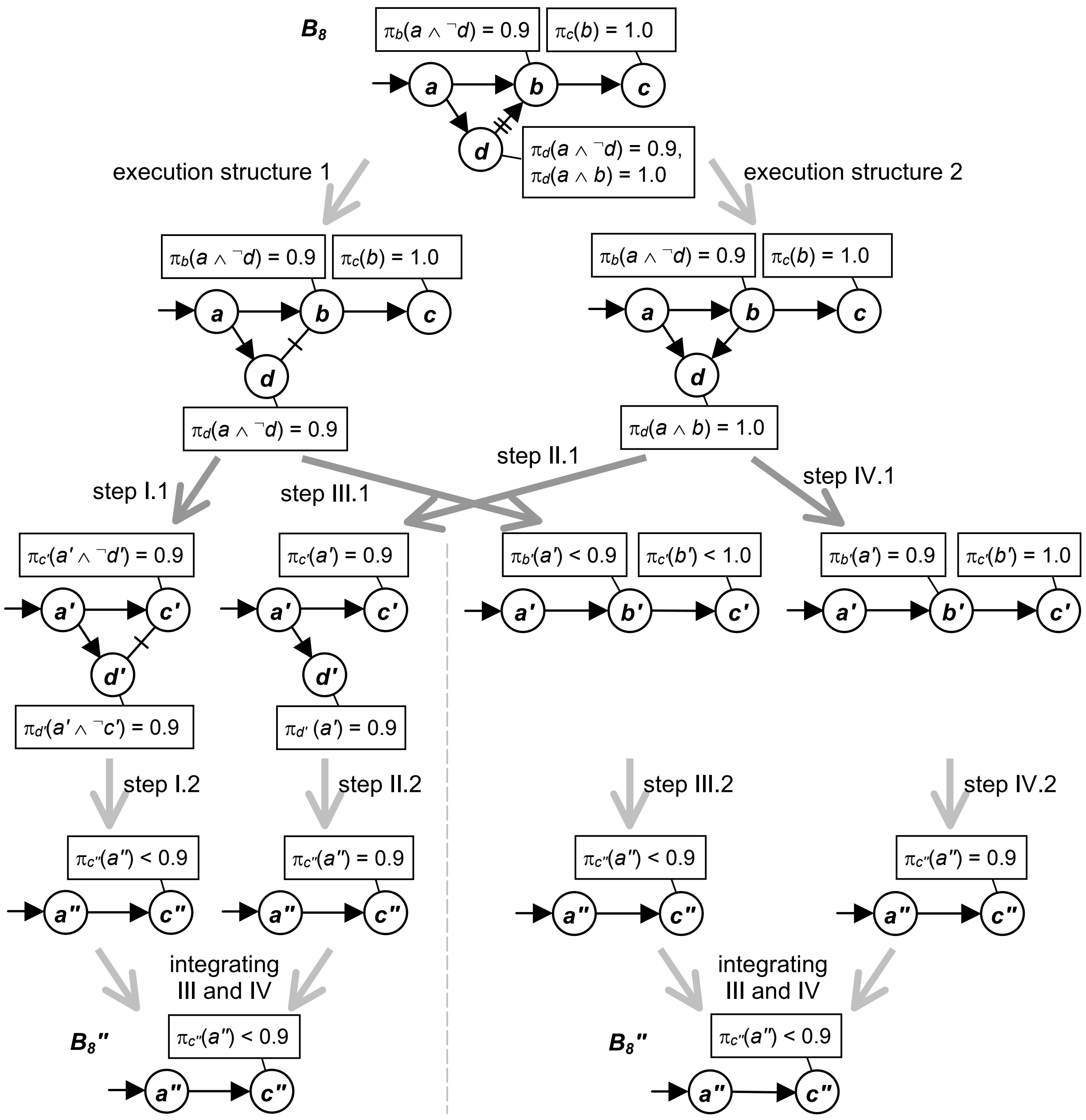


Fig. 13 Abstraction of probability associations involving enabling and disabling conditions.

1. γ_c contains enabling condition z (i.e., $\gamma_c = z \wedge \gamma$):
   (i) $\pi^*_{c'}(\gamma_{c'}) = \pi^*_c(\gamma_c)$, if c also depends indirectly on the occurrence of z
   (ii) $= \pi^*_z(\gamma_z) \times \pi^*_c(z \wedge \gamma)$, otherwise
2. γ_c contains disabling condition z (i.e., $\gamma_c = \neg z \wedge \gamma$):

(i) $\pi^*_{c'}(\gamma_{c'}) = \pi^*_c(\gamma_c)$

Abstraction rule 2 is simpler than the corresponding rule for the simple probability attribute, because the extended attribute defines the probability that action c occurs *due to* γ_c, when assuming γ_c is satisfied. Fig. 14 illustrates the application of the abstraction rules on the step-wise abstraction from inserted actions c and b in concrete behaviour B_9, which renders abstract behaviour B_9''.

6. Abstraction from final actions

The abstraction method of section 4.2 divides the abstraction of a concrete action structure in a concrete behaviour into *(i)* the abstraction of its inserted actions as explained in section 5, and *(ii)* the replacement of its final actions by an abstract action that models the completion of the concrete action structure.

The following steps are distinguished when replacing the final actions of a concrete action structure A by an abstract action a':

1. determine the causality relation of abstract action a':
   a. determine the causality condition of a' by integrating the causality conditions of the corresponding final actions;
   b. determine the abstraction of the attribute values of the final actions in terms of the possible values or constraints of the attributes of a';
2. determine the causality relations of the abstract actions outside A, denoted by b', which depend on a':
   a. determine the *completion condition* of A, in terms of the occurrences of its final actions that correspond to the occurrence of a'. Replace this completion condition by a corresponding condition in terms of a' in the causality relations of abstract actions b';
   b. replace references to the information, time and location attributes of the final actions of A in b' by references to the corresponding attributes of a' as defined in step 1b.

These steps are illustrated below by means of two examples.

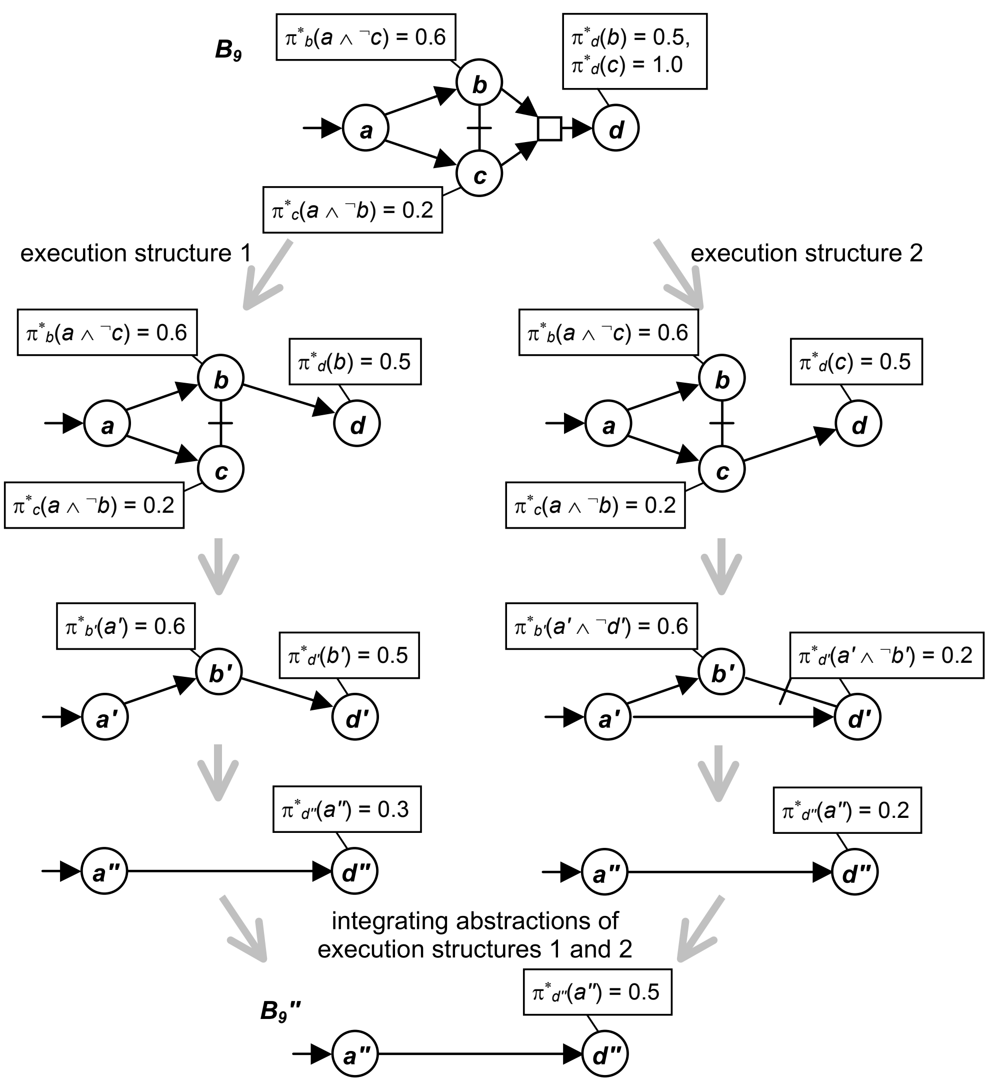
6.1. Example: receipt of segmented data packet

Fig. 15 depicts an example of the abstraction of a conjunction of final actions. Concrete action s models the sending of a data unit, which is segmented into three data segments. Concrete actions r_1, r_2 and r_3 model the receipt of these three data segments, and concrete action a models the reassembly of the original data unit.

Concrete actions r_1, r_2 and r_3 are the final actions of concrete action structure r, which models the receipt of the entire data unit. This concrete action structure is replaced by abstract action r' in the abstract behaviour.

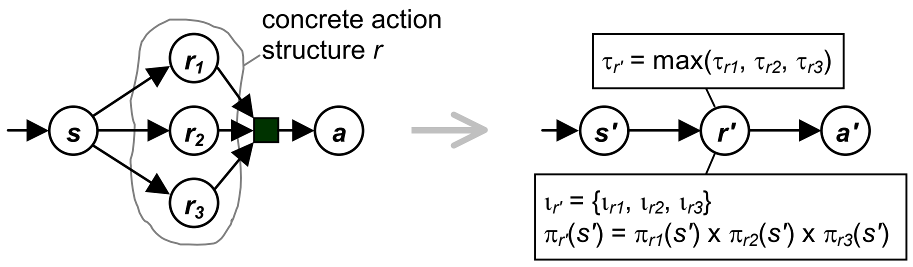
The completion condition of concrete action structure r corresponds to the occurrences of all final actions, i.e., $r_1 \wedge r_2 \wedge r_3$. Since this condition is equal to the causality condition of concrete action a, i.e., $r_1 \wedge r_2 \wedge r_3 \rightarrow a$, this condition is replaced by enabling action r' in the causality relation of the corresponding abstract action a', i.e., $r' \rightarrow a'$.

The causality condition of abstract action r' corresponds to the conjunction of the conditions of the final actions, i.e., $s \wedge s \wedge s$, which can be simplified to s. The time moment of r' corresponds to the time moment at which the last data segment is received. The information value established in r' consists of all segments of the original data unit, since concrete action structure r can only terminate when all segments have been received. Consequently, the conditional probability that r' occurs is equal to the conditional probability that all actions r_1, r_2 and r_3 occur, i.e., in case of the simple probability attribute $\pi_{r'}(s') = \pi_{r1}(s) \times \pi_{r2}(s) \times \pi_{r3}(s)$, and in case of the extended attribute $\pi^*_{r'}(s') = \pi^*_{r1}(s) \times \pi^*_{r2}(s) \times \pi^*_{r3}(s)$.



B_9
$\pi^*_b(a \wedge \neg c) = 0.6$
$\pi^*_d(b) = 0.5$, $\pi^*_d(c) = 1.0$
$\pi^*_c(a \wedge \neg b) = 0.2$
execution structure 1
execution structure 2
$\pi^*_b(a \wedge \neg c) = 0.6$
$\pi^*_d(b) = 0.5$
$\pi^*_c(a \wedge \neg b) = 0.2$
$\pi^*_b(a \wedge \neg c) = 0.6$
$\pi^*_d(c) = 0.5$
$\pi^*_c(a \wedge \neg b) = 0.2$
$\pi^*_{b'}(a') = 0.6$
$\pi^*_{d'}(b') = 0.5$
$\pi^*_{b'}(a' \wedge \neg d') = 0.6$
$\pi^*_{d'}(a' \wedge \neg b') = 0.2$
$\pi^*_{d''}(a'') = 0.3$
$\pi^*_{d''}(a'') = 0.2$
integrating abstractions of execution structures 1 and 2
B_9''
$\pi^*_{d''}(a'') = 0.5$

Fig. 14 Abstraction of extended probability associations.

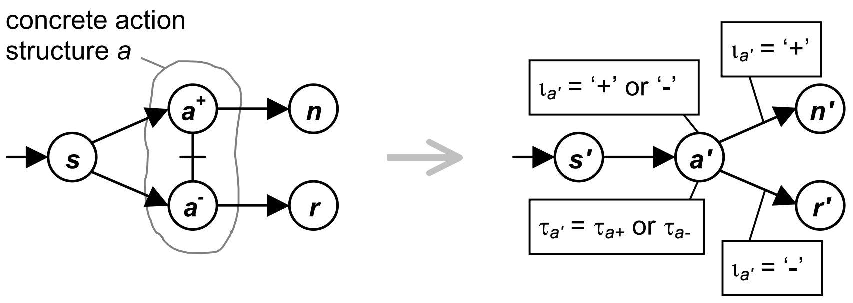


concrete action structure r
s
r_1
r_2
r_3
a
$\tau_{r'} = \max(\tau_{r1}, \tau_{r2}, \tau_{r3})$
s'
r'
a'
$\iota_{r'} = \{\iota_{r1}, \iota_{r2}, \iota_{r3}\}$
$\pi_{r'}(s') = \pi_{r1}(s') \times \pi_{r2}(s') \times \pi_{r3}(s')$

Fig. 15 Receipt of segmented data packet.

6.2. Example: receipt of acknowledgement

Fig. 16 illustrates an example of the abstraction of the disjunction of final actions. Concrete action *s* models the sending of a positive or negative acknowledgement, concrete actions a^+ and a^- model the receipt of a positive or negative acknowledgement, respectively, and concrete actions *s* and *n* model the sending of the next data unit or the retransmission of the last data unit, respectively.



concrete action structure *a*
a^+
s
n
a^-
r
$\iota_{a'}$ = '+' or '-'
$\iota_{a'}$ = '+'
s'
a'
n'
r'
$\tau_{a'} = \tau_{a+}$ or τ_{a-}
$\iota_{a'}$ = '-'

Fig. 16 Receipt of acknowledgement.

Concrete action structure *a* represents the receipt of either a positive or a negative acknowledgement. This concrete action structure is replaced by abstract action *a'* in the abstract behaviour.

The causality condition of *a'* corresponds to the disjunction of the conditions of the final actions, i.e., $s \vee s$, which can be simplified to *s*. Two distinct information values are established implicitly in concrete action structure *a*, i.e., a positive or a negative acknowledgement, which are represented explicitly by the information attribute values '+' and '–' of *a'*, respectively. The time moment of *a'* corresponds to the time moment at which either a^+ or a^- occurs, i.e., $\tau_{a'} = \tau_{a+}$ or $\tau_{a'} = \tau_{a-}$. The conditional probability of the occurrence of *a'* corresponds to the conditional probability that a^+ or a^- occurs, i.e., in case of the simple probability attribute $\pi_{a'}(s') = \pi_{a+}(s) + \pi_{a-}(s) - (\pi_{a+}(s) \times \pi_{a-}(s))$, and in case of the extended probability attribute $\pi^*_{a'}(s') = \pi^*_{a+}(s) + \pi^*_{a-}(s)$.

The completion condition of concrete action structure *a* corresponds to the occurrence of one of the final actions, i.e., $a^+ \vee a^-$. This condition, however, does not appear in the causality relations of actions *n* and *r*. The conditions of actions *s* and *r* correspond to a specific occurrence of concrete action structure *a* in which either a positive or negative acknowledgement is established, respectively. These conditions can be replaced by the combination of enabling condition *a'* and a constraint on the information value that is established in *a'*, i.e., $a'\,[\iota_{a'} = \text{'+'}] \rightarrow n'$ and $a'\,[\iota_{a'} = \text{'-'}] \rightarrow r'$.

6.3. (Im)possibility of abstraction

From our experience applying the method presented in this paper we observed that it is not always possible to apply our abstraction rules on a concrete action structure. For example, when we assume that in the example of Fig. 15 a concrete action that depends on the receipt of a single data segment exists, the condition of this action cannot be replaced by a condition in terms of the (non-)occurrence of abstract action *r'*. In this case, no abstraction of concrete action structure *r* as a single abstract action *r'* would be possible. However, this does not invalidate the method. The impossibility to apply the abstraction rules on a concrete action structure is either caused by an incorrect refinement of an abstract action, or by an incorrect identification of the reference actions in a concrete behaviour.

7. Example: client-server interactions

We consider the design of client-server interaction support to illustrate some of the refinement rules discussed in the previous sections. We have adopted the following conventions in graphical behaviour representations: an action identifier starts with the name of the (inter)action point at which the action happens (e.g., *Areq* is an action that happens at action point *A*), and information attributes are represented by their sorts (e.g., *server_type* represents an information value *v* of sort *server_type*). Furthermore, we place action identifiers next to their corresponding actions, instead of placing them inside text-boxes as required in our notation.

7.1. Initial design

The initial design focuses on the interaction between a client application and a data server, abstracting from the identification of the data server and from the remote communication with the data server. During the design process, this abstraction should be preserved for the client application, i.e., the design of the internal operation of the client application can be based on the interaction (pattern) as is established in this initial design.

Fig. 17 depicts the composition of entities distinguished in this design. The *client_application* entity interacts with the *data_server* entity at an interaction point *A*.

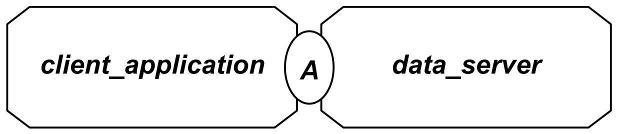


Fig. 17 Entity domain representation of the initial design.

Fig. 18 depicts the behaviour at interaction point *A*. The client application requests certain data from the data server through the *Areq* action. This action has an information attribute with two elements: the *server_type* element is used to specify the type of data server that is required, and the *question* element is used to specify the properties of the data requested. The *Areq* action is followed by either an *Arsp* action, in case the requested data was found, or an *Arej* action, in case the data server failed to find the requested data. The information value attribute *answer* of the *Arsp* action contains the requested data. Although this is not indicated by attributes of the *Arej* action, there may be two main causes for failure: the data server does not have the requested data, or a time-out occurs before the data server is able to respond. The specific rejection reason is not explicitly indicated in the sequel. This also applies to other 'reject' actions that are identified later on.

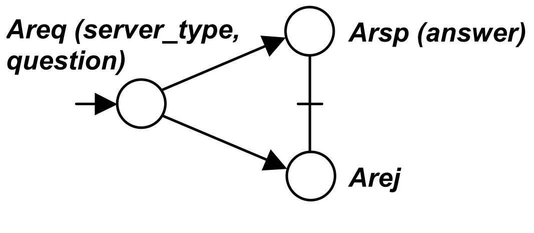


Fig. 18 Behaviour domain representation of the initial design.

7.2. Introduction of a trader component

The first design step focuses on the mechanism to locate a suitable data server. We assume that there are potentially many data servers that can provide the requested data. Hence, we introduce a trader component, which is able to provide the name of a specific data server, given the type of the data server and the name of the client application. Since we want to hide the existence of a trader from the client application, we introduce a fourth component, called the interface handler, which provides the original interface to the client application.

Fig. 19 shows the interconnection of the entities identified above. At this abstraction level, the *client_application* entity interacts with the *interface_handler* entity, instead of with the *data_server* entity, at interaction point *A*. The *interface_handler* entity interacts with the *trader* entity and with the *data_server* entity at interaction point *B* and *C*, respectively.

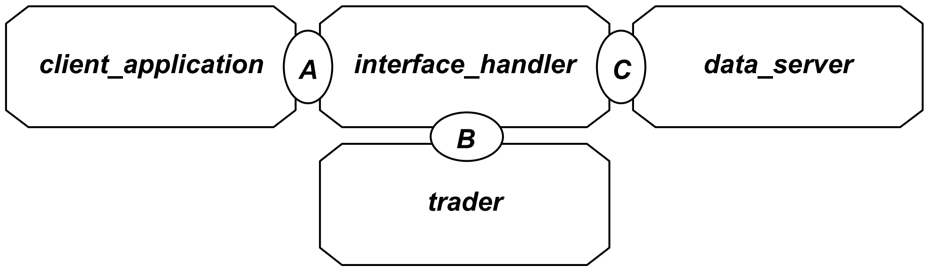


Fig. 19 Entity domain representation after the introduction of a trader.

Fig. 20 depicts the refined behaviour. After the *Areq* action, the interface handler asks the trader for a suitable data server through a *Breq* action. The *Breq* action has an information attribute with two elements, viz. *server_type* and *client_name*. These are used by the trader to determine a suitable data server. If the search for a suitable server is successful, the *Breq* action is followed by a *Brsp* action, with information attribute *server_name*. Otherwise, a *Brej* action is performed. The *Brsp* action is followed by a *Creq* action, in which data is requested from the data server. The properties of the requested data are given by the information attribute *question* and the data server is identified by means of the information attribute *server_name*. The *Creq* action is followed by either a *Crsp* action, in case the requested data was found, or a *Crej* action, in case the data server failed to find the requested data. The information attribute *answer* of the *Crsp* action contains the requested data. The *Crsp* action enables the *Arsp* action, which has *answer* as information attribute. An *Arej* action happens after a *Brej* or a *Crej*.

Conformance assessment

The concrete behaviour of Fig. 20 conforms to the abstract behaviour of Fig. 18. This can be assessed by applying the method of section 4.2 as follows:

1. identify concrete actions *Areq*, *Arsp* and *Arej* of Fig. 20 as single reference actions and identify the remaining actions as inserted actions;
2. abstract from the inserted actions (in any arbitrary order) using the method of section 5.2;
3. compare the abstract behaviour obtained in step 2 with the behaviour of Fig. 18. These two behaviours are equivalent.

Alternative applications of the method of section 4.2 that render the same result are possible. For example, one may consider concrete actions *Creq*, *Crsp* and *Crej* as a concrete action structure with two final actions, obtained through action refinement of an abstract action *C*. In this case, one replaces the concrete action structure by the abstract action *C* first, and subsequently abstracts from *C* by considering this action as an inserted action.

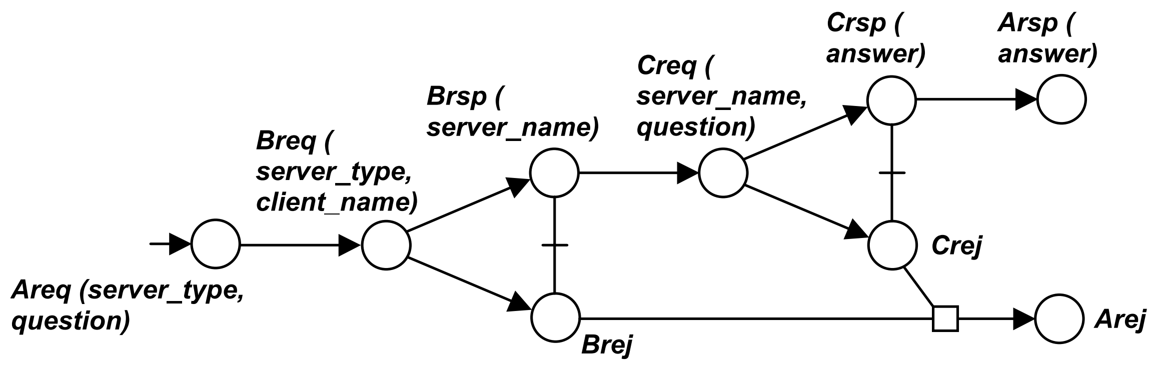

Crsp (answer)
Arsp (answer)
Creq (server_name, question)
Brsp (server_name)
Breq (server_type, client_name)
Crej
Areq (server_type, question)
Brej
Arej

Fig. 20 Behaviour domain representation after the introduction of a trader.

7.3. Federation of traders

This design step focuses on the (possibly concurrent) use of different connected traders. This step assumes that there is not a single trader that knows all possible data servers, but actually multiple traders and multiple trading domains. A trader in the domain associated with the client application (local domain) passes a request to a trader in a remote domain, in case it can not find a data server. This process is based on agreed procedures known as *trader federation*. In order to speed up the search process, the local trader may also contact multiple traders at the same time. In the following, we consider the situation where a local trader concurrently contacts two remote traders.

Fig. 21 depicts the entities involved in the trader federation considered in this example.

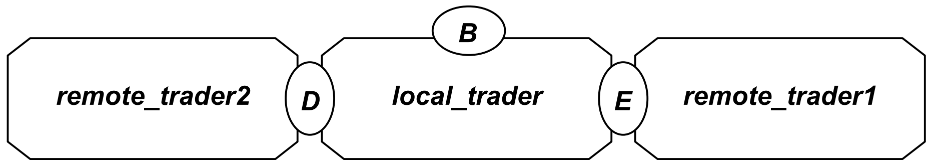


Fig. 21 Entity domain representation after the introduction of trader federation.

Fig. 22 shows the behaviour of the three federated traders.

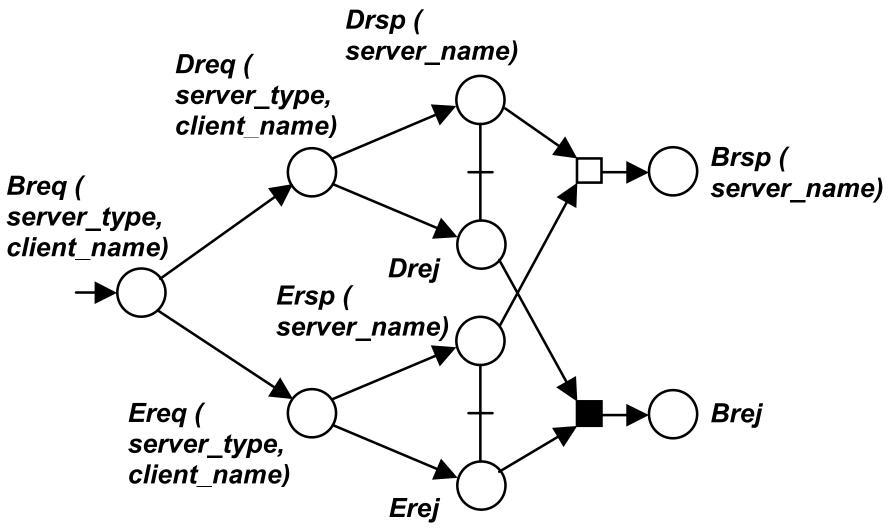
Conformance assessment

The concrete behaviour of Fig. 22 conforms to the abstract sub-behaviour of Fig. 20 consisting of actions *Breq*, *Brsp* and *Brej*, and their relations. This can be assessed by applying the method of section 4.2 analogously to the previous section. Fig. 23 depicts the abstraction of the concrete behaviour of Fig. 22.

7.4. Remote communication

The final design step focuses on the remote communication between components. We assume that the components identified so far reside on different computing nodes of a distributed system. Hence, the communication between components is accomplished via an intermediate component that may not be reliable (i.e., messages may get lost).

Fig. 24 depicts the entities involved in communication between the interface handler and the data server.



Drsp (server_name)
Dreq (server_type, client_name)
Brsp (server_name)
Breq (server_type, client_name)
Drej
Ersp (server_name)
Ereq (server_type, client_name)
Brej
Erej

Fig. 22 Behaviour domain representation after the introduction of trader federation.

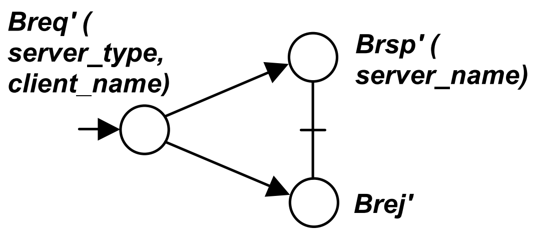


Fig. 23 Abstraction of the concrete behaviour in Fig. 22.

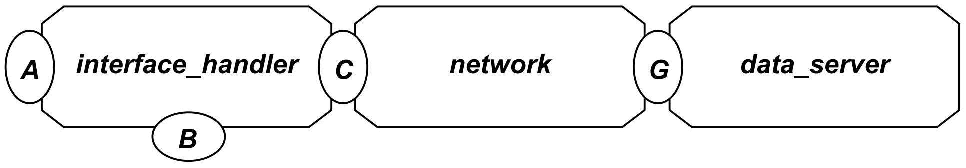


Fig. 24 Entity domain representation after the consideration of remote communication.

Fig. 25 depicts the behaviour of an instance of communication between the interface handler and the data server.

Conformance assessment

The concrete behaviour of Fig. 25 conforms to the abstract sub-behaviour of Fig. 20 consisting of actions *Creq*, *Crsp* and *Crej*, and their relations. This can be assessed by applying the method of section 4.2 as follows:

1. identify concrete actions *Freq* and *Fcnf* as single reference actions and identify concrete actions *Frej1* and *Frej2* as a group of reference actions;
2. abstract from inserted actions *Gind* and *Grsp* using the method of section 5.2;
3. replace final actions *Frej1* and *Frej2* by abstract action *Frej'*, which represents two alternative causes for rejection;
4. compare the abstract behaviour obtained in step 3 with the behaviour of Fig. 20 consisting of actions *Creq*, *Crsp* and *Crej*. These two behaviours are equivalent.

Fig. 26(i) and (ii) depict the abstract behaviours obtained in step 2 and 3, respectively.

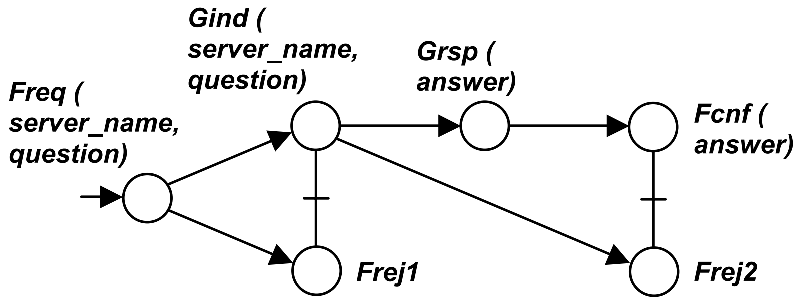


Fig. 25 Behaviour domain representation after the consideration of remote communication.

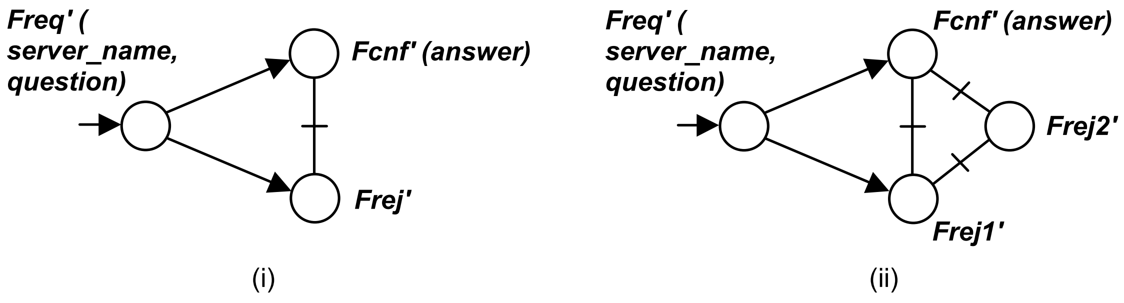

Freq' (server_name, question)
Fcnf' (answer)
Frej'
(i)
Freq' (server_name, question)
Fcnf' (answer)
Frej2'
Frej1'
(ii)

Fig. 26 Abstraction of the concrete behaviour in Fig. 25.

8. Conclusions and further work

In order to effectively support the design process of distributed systems, a design model has to be developed bearing in mind design operations such as behaviour refinement. Design operations are necessary to transform an abstract design into a more concrete design such that the concrete design conforms to the abstract design. Furthermore, these design operations should be generic and applicable throughout the design process.

This paper presents a method to perform behaviour refinement, which consists of replacing an abstract behaviour by a concrete behaviour. This method is based on the precise definition of the architectural concepts of action and causality relation, and precise rules for the manipulation of these concepts. Abstraction rules are defined to assess whether a concrete behaviour conforms to the original abstract behaviour.

We are confident that the work presented in this paper is more generally applicable than what can currently be found in the literature in the area of behaviour refinement. Our work is based on the manipulation of carefully developed architectural (design) concepts while to the best of our knowledge most other work on this subject has been performed based on the manipulation of properties of formal (mathematical) models with limited expressive power (Aceto, 1991, Vogler, 1993). Our design model allows clear analysis and design of distributed system behaviours, also supporting the modelling of (real) time and probability. Our method for behaviour refinement does not pose restrictions on the behaviours that can be considered, and can in principle be applied to arbitrary complex behaviours.

Extensions and improvements of this work aim at:

- the further development of our design language, called *Interaction System Design Language* (*ISDL*), which enables a designer to apply and express our design concepts in an intuitive and easy way. The name of this language reflects the concepts underlying our design model: interaction and interaction system, stressing the collaboration of functional entities;
- the development of complete formal support for ISDL in general, and for behaviour refinement in particular. Much work on this has already been done in (Quartel, 1998);
- the development of an integrated tool environment for ISDL, to support analysis techniques such as simulation, model checking and conformance assessment. The latter technique can be used to partly automate the behaviour refinement design operation.

9. Acknowledgement

We like to thank Prof. C.A. Vissers for the interesting and stimulating discussions on some of the topics of this paper, and for providing the means to perform this work.

10. References

Aceto, L. and Henessy M., 1991, “Adding action refinement to a finite process algebra,” Lecture Notes in Computer Science, Vol. 510, pp. 506-519.

Bolognesi, T., van de Lagemaat, J. and Vissers, C.A., 1995, LOTOSphere: software development with LOTOS, Kluwer Academic Publishers, The Netherlands.

Ferreira Pires, L., 1994, “Architectural notes: a framework for distributed systems development,” PhD thesis, University of Twente, Enschede, the Netherlands.

Quartel, D.A.C., van Sinderen, M.J. and Ferreira Pires, L., 1999, “A model-based approach to service creation,” Proceedings of the 7th IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems, IEEE Computer Society Press, pp. 102-110.

Quartel, D.A.C., 1998, “Action relations - Basic design concepts for behaviour modelling and refinement,” PhD thesis, University of Twente, Enschede, the Netherlands.

Quartel, D.A., Ferreira Pires, L., van Sinderen, M.J., Franken, H.M. and Vissers, C.A., 1997, “On the role of basic design concepts in behaviour structuring,” Computer Networks and ISDN Systems, Vol. 29, pp.413-436.

Quartel, D.A.C., Ferreira Pires, L., Franken, H.M. and Vissers, C.A., 1995, “An engineering approach towards action refinement,” Proceedings of the Fifth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems, IEEE Computer Society Press, pp. 266-273.

Rechtin, E., 1992, “The art of systems architecting,” IEEE Spectrum, October, pp. 66-69.

van Sinderen, M., Ferreira Pires, L., Vissers, C.A. and Katoen, J.-P, 1995, “A design model for open distributed processing systems,” Computer Networks and ISDN Systems, Vol. 27, pp. 1263-1285.

van Sinderen, M., Ferreira Pires, L. and Vissers, C.A., 1992, “Protocol design and implementation using formal methods,” The Computer Journal, Vol. 35, No. 5, pp. 478-491.

Vissers, C.A., van Sinderen, M. and Ferreira Pires, L., 1993, “What makes industries believe in formal methods, Protocol Specification, Testing, and Verification, XIII, Elsevier Science Publishers B.V. (North-Holland), pp. 3-26, The Netherlands.

Vogler, W., 1993, “Bisimulation and action refinement,” Theoretical Computer Science, Vol. 114, pp. 173-200.