Specialization of object behaviors and requirement specifications[1]

Gregor v. Bochmann
Reinhard Gotzhein
Université de Montréal
Dept. d'IRO
CP 6128, Succursale A
Montréal, Québec, Canada

Abstract: Given two behavior descriptions, the question whether one is a specialization of the other is important in many situations; for instance an implementation may be considered a specialization of the specification, or in other cases, two specification may have to be compared. Different notions of "specialization" have been developed in different contexts; for instance, a subrange type may be considered a "specialization" of its base type, objects with additional functions may be considered "specializations", a partial functions becomes more "specialized" by having it defined more completely, and a state machine with less non-determinism may be considered a "specialization" of another. This paper shows that all these different notions may be considered to be special cases of a comparison relation, called "reduction", which is based on two more basic relations which correspond to the notions of "safety" and "non-blocking". The paper defines these concepts in a formal framework and presents certain important properties of these relations. It also shows how these concepts may provide a formal framework for the systematic constructions of specifications that have certain given properties. A more general notion of "requirements specification" is also introduced which allows the separation of the maximally allowed behavior ("safety") and a minimal behavior to be implemented, where certain features are explicitly specified as optional.

Keywords: subtyping, specialization, object behavior, multiple inheritance, non-determinism, specifications, conformance, implementation relations, reduction, extension.

---

[1] This research was supported by a grant from the Canadian Institute for Telecommunications Research under the NCE program of the Government of Canada

1 Introduction

The implementation of an information system usually goes through several steps of the software development process, such as requirement analysis, functional specification, detailed design, code generation and testing. The system descriptions obtained during one of these steps is usually obtained from the previous description by some form of specialization. For instance, we may say that the detailed design is a "specialization" of the functional specification. The notion of specialization is also used when a "generic" system must be adapted to several different user requirements. For each case, the "specific" design may be considered as a "specialization" of the generic one.

In the context of object-oriented programming languages, two kinds of "specializations" have been considered [America, 87b #23]: (1) code-sharing, usually called "inheritance", which means that a subclass inherits from its superclass the operations (methods) and the procedural code associated with them, and (2) inheritance of properties, often called subtyping, which means that certain properties, in particular the interface definition, of the superclass remain valid for the subclass. A subclass may define additional operations and may introduce additional properties, and as such, it may be considered a specialization of the superclass.

Depending on the different specification paradigms used for the specification of systems, various kinds of specialization have been considered, such as the following:

(a) In the context of classical programming languages, a type is considered a set of values, and a subtype ("specialization") is a subset of such values.

(b) In the context of incompletely specified functions (with "don't care" arguments), a specialization is a function for which certain "don't care" situations obtain a specific result.

(c) In the context of object-oriented languages, a class with additional operations (methods) is a specialization (as discussed above).

(d) In the context of sequential machines with input/output behavior, a machine which restrains the number of possible execution traces (sequences of input/output events) is a specialization.

(e) In the context of non-deterministic machines, a number of different "specialization" relations have been considered [Brinksma, 86 #301][Nicola, 87 #767]. One of these relations is called *reduction* which means that a specialized process only performs traces which are also performed by the more general process, and only blocks in situations where the more general process also may block.

It is important to define what the precise meaning of "specialization" is. One possible approach is to define the notion of "specialization" between type definitions as follows: An object type C' specializes a type C if an object instance of type C can be replaced, within the overall system, by an instance of the type C' without invalidating certain important system properties. Certain papers explore this approach in the context where the important property is type checking[Black, 87 #104] [Cardelli, 88a #330]. Other papers address the comparison of object behaviors and their influence on the overall system behavior [Cusack, 89 #387][America, 89 #22].

The intent of this paper is to show that a unified concept of "specialization", as defined in Section 3, can be used in all the above contexts.

The paper is structured as follows. A general framework for modelling object behaviors is given in Section 2. It is based on the notions of actions (also called operations or methods) with inputs and outputs and the state of the object which may change during the execution of an action. Different kinds of behaviors are distinguished, such as constant behaviors which always remain in the same state, and state-deterministic behaviors the state of which can be deduced from the trace of the past actions. Nondeterminism may be introduced either through the fact that the ouput of a given action is not determined, or by the fact that the next state of an object belongs to a *set* of possible states, called *state-nondeterminism*, and only subsequent observation or experimentation may determine the state in which the object was.

Section 3 defines several relations for comparing behaviors. They are the basis for the definition of a general notion of *reduction*, which is an order relation between behaviors. It seems that this notion is a natural definition of "specialization" in the context of the so-called *undefined by default* convention, which means that for situations which are not defined, any action is allowed. Another relation, called *extension*, seems natural in the context where undefined situations lead to blocking.

Section 4 deals with the construction of specifications which denote behaviors that have certain specialization relations with one another. The major part of the section deals with the construction of type definitions and type checking, as included in many programming and specification languages. The type checking may be considered as a verification of the reduction relation between a behavior expression and a declared behavior.

In Section 5, a more general notion of "requirements specification" is introduced. We consider an approach where a requirements specification is a pair of two behavior descriptions, one representing the safety requirements (a kind of "maximum" behavior) and one denoting the "minimum" behavior requirements.

2 A framework for modeling object behavior

2.1 Introduction

We consider systems that consist of a number of *object instances* (or *objects* for short). Each object has a *behavior* that characterizes its ability to interact with other objects within the system during its lifetime. There may be a large number of object instances that have the same or similar behaviors. In the following, we will discuss how these behaviors can be formally modeled, and how a comparison between different behaviors can be made.

In the context of system specifications, one usually defines, for a given set of object instances, certain requirements which must be satisfied by the behavior of these object instances. The requirement could, for instance, be the statement that the behavior of the object instances must be equivalent to a *single* given behavior. However, more freedom is often allowed for the implementation. In such cases, it seems more adequate to define the requirement in terms of a *set* of behaviors, where the behavior of the object instances must belong to this set.

Being part of a system, each object has an environment consisting of other objects that may or may not belong to the system. In the course of system evolution, it is possible that the environment is extended, which may result in the need for additional behavior of an object. This additional behavior, however, should not affect the execution in the old environment. Also, it may be the case that the object behavior contains options, and that it is sufficient to implement some of them. In these cases, a systematic way of comparing object behaviors is needed. We will discuss in Section 3 several relations for comparing behaviors, which may be considered to represent the notion of "specialization"

2.2 Basic definitions of behaviors

In our framework, a behavior is given in terms of external actions that may be offered to the environment. In the general case, the sets of offered actions may depend on the object's history, which is modeled as the sequence of previous actions, called *trace*. The basic notions of *action*, *offered actions*, *trace*, and *behavior* are defined subsequently.

An action is a triplet f(i;o), where f is an operation name, i is the value of the input parameter(s) of the operation, and o is the value of the output parameter(s) returned by the object as result of the operation.

Definition 2.1: Let F be a set of *operation names*, I be a set of *inputs*, and O be a set of *outputs*. Then Act = $F \infty I \infty O$ is the set of *actions*.

Examples:

CM1 (a coffee machine): There is one operation, denoted serve, one input, denoted coin, and one output, denoted coffee. The only action is serve(coin; coffee).

SQRT1 (an object providing the square root function): There is one operation, denoted sqrt, and the set of inputs and outputs is the set of non-negative real numbers. An action is written sqrt(i; j) where i and j are real numbers.

QUEUE1 (a FIFO queue of integers): There are two operations, called put and get; put has an integer input and no output, while get has no input and an integer output. The actions are written put(i;) and get(;j) where i and j are integer values.

The sequence of actions that have occurred since the object has been instantiated is called *trace* (or *history*). We assume here that objects execute one action at a time. Therefore, the history of an object can be modeled as a sequence of actions. After each trace, an object allows for a (possibly empty) set of actions, called *chosen set of offered actions*. Nondeterminism is modeled by allowing, for a given trace, several sets of offered actions. In this case, the object may choose one of them. Each set of offered actions corresponds to what is often called a *state*. These considerations lead to the following definition of behavior in general.

Definition 2.2: Let B be a set of pairs (t,**A**), where t is a sequence of actions, and **A** = {A_1,...,A_n} is a set of sets of actions. Then traces(B) is the set of all t such that there is (t,**A**) □ B. The elements of traces(B) are called *traces* of B. For a pair (t,**A**), the elements of **A** are called *sets of offered actions* after t. A *state* of B is a pair (t,A), where A □ **A**. B is called a *behavior* if it satisfies the following conditions:

a) if (t,**A**) □ B, then **A** □ {};
b) if (t,**A**) □ B and (t,**A**') □ B, then **A** = **A**';
c) <> □ traces(B);
d) if (t,**A**) □ B, A □ **A**, and a □ A, then t ^<a> □ traces(B);
e) if t ^<a> □ traces(B), then there are (t,**A**) □ B and A □ **A** such that a □ A.

<> denotes the empty trace, ^ is concatenation of traces.

Examples:

CM1 (a coffee machine providing continuous service): The set of actions was defined above. The behavior consists of the set of all pairs (t, {{serve(coin; coffee)}}), where t is any finite sequence of serve(coin; coffee) actions.

CM2 (a coffee machine that serves only one coffee): The set of actions is the same as for CM1. The behavior consists of the following two pairs (<>, {{serve(coin; coffee)}}) and (<serve(coin; coffee)>, {{}}) .

CM3 (a machine serving one coffee or one tea, nondeterministically): As CM2, but with an additional actions serve(coin; tea) and serve(bill); the behavior is the set of pairs (<>, {{serve(coin; coffee), serve(coin; tea)}}), (<serve(coin; coffee)>, {{}}), and (<serve(coin; tea)>, {{}}).

CM4 (a machine serving one coffee or tea according to the desire of the customer): As CM3, except that the actions serve(coin; coffee) and serve(coin; tea) are replaced by the actions select-coffee(coin; coffee) and select-tea(coin; tea), respectively.

DS1 (a drink server like CM2, but sometimes offering whisky): There is the additional action serve(bill; whisky). The behavior consists of the following two pairs (<>, {{serve(coin; coffee)}}) and (<serve(coin; coffee)>, {{}, {serve(bill; whisky)}}). After serving a coffee, DS1 may offer a whisky, depending on which state is entered.

SQRT1 (square root function for non-negative reals): The set of actions was defined above. The behavior is the set of all pairs (t, {{sqrt(i; j)}}), where j^2=i and t is any finite sequence of such actions sqrt(i; j).

QUEUE1 (a FIFO queue of integers which blocks when empty): The set of actions was defined above. The behavior is the set of all pairs (t, {{put(i;), get(;j)}}) and pairs (t', {{put(i;)}}) where t and t' are sequences of n put and interleaved with m get operations, and where n>m for t and n=m for t', and J is the input of the (m+1)th put operation in t.

QUEUE2 (idem, but the operation *get* outputs *error* when the queue is empty): A similar definition.

Definition 2.3: Let (t,A) be the current state of B, f be an operation, and i be an input. We say that B *blocks* for f(i), if there exists no output o such that f(i;o) □ A. B *blocks* for f, if for all i, it blocks for f(i). B *accepts* f(i), if it does not block for f(i). The (non-blocking) *domain of an operation* f in state (t,A) of B is the set of i such that B accepts f(i) in this state. The *domain of a behavior* B in state (t,A) (written "Dom(A)") is the set of f(i) accepted by B in this state. We say that B has an *undetermined output* for f(i), if there are at least two actions f(i;o) and f(i;o') in A with o □ o'.

Note: Any set of f(i) which has an empty intersection with the domain of B in state (t,A) may be considered a "refusal set" (in the sense of CSP [Hoare, 85 #526]), since B blocks for all f(i) in such a set.

Examples:

a) The behavior CM2 accepts serve(coin) after the trace <>. It blocks for serve after the trace <serve(coin; coffee)>.

b) In state (<>,{serve(coin; coffee),serve(coin; tea)}), CM3 blocks for serve(bill), and it has an undetermined output for serve(coin).

Definition 2.4:

a) A behavior B is *history independent* iff the set of set of offered actions **A** is the same for all traces t.

b) A behavior B is *state deterministic* iff the state of B is always uniquely determined by the trace.

c) A behavior is *constant* iff it is history independent and state deterministic.

d) A behavior B is *deterministic* iff it is state deterministic, and the output is always uniquely determined by operation and input.

A state deterministic behavior[2] may still have undetermined outputs, thus state determinism is weaker than determinism. We note that a history independent behavior is uniquely characterized by the set **A** = {A_1,...,A_n} of sets of offered actions; a constant behavior is characterized by a single set A of offered actions, and a state deterministic behavior is characterized by its set of traces.

Examples:

a) Among the above examples, only the behaviors CM1 and SQRT1 are history independent.

b) Among the above examples, only the behavior DS1 is not state deterministic.

c) Therefore, the behaviors CM1 and SQRT1 are constant.

d) Among the above examples, only the behaviors CM3 and DS1 are nondeterministic; CM3 because of its output non-determinism, DS1 because of its state non-determinism.

e) CM1 can be characterized by the set of offered actions {serve(coin; coffee)}. SQRT1 can be characterized by the set of offered actions {sqrt(i; j) | j^2=i }. CM4 can be characterized by the set of traces {<>, <select-coffee(coin; coffee)>, <select-tea(coin; tea)>}.

Other examples of behavior:

[2] In [Starke, 72 #1093], state deterministic behaviours have been called "observable"; in [Cerny, 92 #1092], they have been termed "observably nondeterministic".

COMM1 (communication service access point): The access point may accept or reject a connection request, if accepted data may be sent. The actions are con(;accept), con(;refuse), data(;ack); initially the first two actions are possible, the action con(;accept) leads to a new state, the so-called data transfer state, in which only the action data(;ack) is possible. This behavior is represented by the state diagram of Figure 1a.

SQRT2 (square root function, undefined for negative inputs): As SQRT1; this constant behavior is characterized by a set of offered actions which includes also all actions of the form sqrt(i; j) where i<0 (arbitrary j).

SQRT3 (square root function, provides error message for negative inputs): As SQRT1; this constant behavior is characterized by a set of offered actions which includes also all actions of the form sqrt(i; error) for i<0 .

SQRT4 (square root function, provides complex result for negative inputs): As SQRT1; this constant behavior is characterized by a set of offered actions which includes also all actions of the form sqrt(i; j*imaginary), where i<0 and $j^2=-i$.

SORT1 (sorting a sequence of input records according to the value contained in the first field of each record (sort key); in the case of equal sort keys, the order of the records is not determined): This constant behavior is characterized by the set of offered actions of the form sort(s1; s2) where s1 and s2 contain the same records and s2 is sorted. The behavior has undetermined outputs.

SORT2 (as SORT1, except that in the case of equal sort keys, the order of the records in the output is the same as for the input): The behavior is characterized by a subset of the offered actions of SORT1 (in an obvious way). The behavior is deterministic.

STOP (the behavior that blocks for all actions): This constant behavior is characterized by the empty set of offered actions.

ARBITRARY (accepts all): This constant behavior is characterized by the set A=Act of all actions offered.

CHAOS (may accept any set of actions or block, as defined in [Hoare, 85 #526]): This history independent behavior is characterized by **A**={A | A ∏ Act} is (highly) nondeterministic.

2.3 Specifications and behaviors

In the subsection above, we have introduced a framework for defining object behaviors in terms of actions, traces, states, etc. The number of actions involved in a given behavior is often infinite. Therefore it is not very convenient to specify a behavior directly in this behavior model. Instead, one usually introduces a specification language with a well-defined syntax and semantics. The syntax defines which (finite) expressions are valid specifications, and the semantics defines for each valid specification either a single behavior, or a set of behaviors that satisfy the specification. In the case of the LOTOS language [ISO, 89a #557], for example, the semantics of a specification is a single transition system, which can be represented in our behavior formalism. For instance, the LOTOS expression " serve ?x:coin !coffee; stop [] serve ?x:coin !tea; stop " has as semantics the behavior CM3. Another example is the specification (in the form of a state diagram) of Figure 1a which denotes the behavior COMM1.

2.4 Conventions for undefined situations

We are interested in this paper in the notion of "specialization". Usually, one system specification is considered a specialization of another one, if it defines certain aspects of the system which were left undefined by the latter. Therefore the notion of "undefined" is important in this context. However, in a formal context, it is necessary to define exactly what it means that some aspect of a system is "undefined". In the following, we discuss different conventions that may be used to give a formal meaning to "undefined".

A special value *UNDEFINED*: A partial function, which is undefined for certain values of its argument type, may be converted into a (full) function (defined for all values of its argument type) by introducing an additional value UNDEFINED into the range of the function. Instead of saying "the result of the function is not defined" one says "the result of the function is UNDEFINED". This approach is also the basis for defining the semantics of recursive functions by the fixpoint operator . An example for this convention is provided by the behavior SQRT3.

Implicitly defined transitions: A incompletely specified finite state machine, for which no transition is specified for certain pairs in present state and input received, may be converted into a completely defined machine by introducing default transitions for the unspecified pairs of state and input. One such convention is used by SDL, where the default transition is a transition back to the same state with no output generated [Belina, 89 #82]. In the case of the specification of Figure 1a, this leads to the behavior denoted by the diagram of Figure 1b. Another possible convention is the introduction of an "error" state and default transitions that lead to this error state, which leads in the case of the specification of Figure 1a to the behavior denoted by Figure 1c.

Blocking by default: In the context of formal specification techniques, it is sometimes assumed that actions that are not explicitly specified are not possible. This is, for instance, the case for LOTOS, CSP, Estelle, Finite State Machines, and Petri Nets. If the environment offers an unspecified action, this action will be blocked. An example of such a behavior is SQRT1 which blocks for negative inputs.

Undefined by default: Alternatively, we may assume that actions that are not explicitly specified are possible, but their outputs are completely undetermined, as modelled in SQRT2 for negative inputs. We call this convention *"undefined by default"*. If the environment offers an unspecified action, an arbitrary output will be returned. In the case of behaviors with state changes, it is also necessary that the convention defines the next state. There are several possibilities; similar to the implicitly defined transitions discussed above, one may assume that the subsequent state is unchanged, or that an "error" state is reached. Different behaviors may be assumed for the latter, such as the behaviors STOP, ARBITRARY, or CHAOS, as defined above.

If one wants to use the *undefined by default* convention and allow for a blocking behavior for certain actions in certain states, it is convenient to introduce the notion of an alphabet.. The *alphabet* of an expression is the set of actions which are explicitely defined for that part of the specification. The undefined by default convention then only applies to actions that are **not** in the alphabet; if for a given state no transition is specified for a given action of the alphabet, the behavior blocks for that action. For example, let us consider the LOTOS expression "B = a; b; stop [] c; stop", which has the alphabet {a, b, c}. If we assume the *undefined by default* convention, the behavior denoted by this expression offers (initially) the actions a, c, and all other actions not in the alphabet, such as d, e, etc. It only blocks for action b, because this action is part of the alphabet. If we assume the *blocking by default* convention, which corresponds to the semantics defined for LOTOS, the behavior denoted by this expression offers (initially) the actions a and c, and blocks for all other actions that may be offered by the environment.

3 Comparison of behaviors

In this section, we consider various relations which may be used for comparing different behaviors. We are in particular interested in the situation where one behavior B' is a "specialization" of another behavior B. The meaning of specialization is not clear in this context and may mean different things. Sometimes, one wants that a system component with the specialized behavior B' may be used to replace any system component with behavior B in any context without affecting "adversely" the system behavior. Several possible formal definitions for "specialization" will be given in the following.

In the context of non-deterministic machines, a number of different "specialization" relations have been considered[Brinksma, 86 #301][Nicola, 87 #767]. One these definitions of specialization, called reduction, has the following two aspects:
(1) Safeness: the traces of the specialized process are included in the traces of the more general one, and
(2) Non-blocking: the specialized process only blocks in situations where the more general one may also block.
These aspects intuitively ressemble the notions of "safeness" and "liveness" in [Alpern, 87 #16], although there are certain differences as discussed below. Instead of using the above notion of safeness, we use in this paper a stronger relation, which we call *constrainment*. Combined with the non-blocking property, it leads to what we call *reduction*, which is slightly stronger than the reduction of [Brinksma, 86 #301].

We define in the following two "specialization" relations, called *reduction* and *extension*. They are defined in terms of simpler relations, called *constrainment*, *domain coverage*, and *constrainment on the domain*. Informally, a behavior B' is constrained by a behavior B, if for all traces of B and B', B' can offer only actions that B can offer, too (this is "safeness", as defined above). B' covers the domain of B, if for all traces of B and B', B' can accept everything that B accepts (this corresponds to "non-blocking", as defined above). B' is constrained by B on its domain, if for all traces of B and B', for the domain of B, B' can offer only actions that B can offer, too; B' may also offer actions f(i;o) that B cannot offer, if f(i) is outside the domain of B. We say that B' is a reduction of B if B' is constrained by B and covers its domain. We say that B' is an extension of B if B' is constrained by B on its domain and covers its domain.

In the following, we define these relations formally, first for the case of constant behaviors, then for history independent behaviors and then for the general case. The formalization for history independent behaviors, which are characterized by a set **A** of sets of offered actions, can also be applied to considerations concerning the comparison of classes of behaviors and type checking in object-oriented languages, as far as a class or a type can be represented by a set of sets of offered actions.

3.1 Comparison of constant behaviors

Definition 3.1: Let B and B' be constant behaviors that are characterized by A and A', respectively.

a) A' *is constrained by* A (written “A' $<_c$ A”) iff for all actions, f(i;o) □ A' implies f(i;o) □ A, i.e., iff A' ∏ A.

b) A' *covers the domain of* A (written “A' $>_d$ A”) iff for all actions f(i;o) □ A, there is an action f(i;o') □ A', i.e., iff Dom(A) ∏ Dom(A').

c) A' *is constrained by* A *on its domain* (written “A' $<_{cd}$ A”) iff for all operations f and inputs i, if A accepts f(i), then for all o, if f(i;o) □ A', then f(i;o) □ A.

d) A' *reduces* A (written “A' $<_r$ A”) iff A' $<_c$ A and A' $>_d$ A.

e) A' *extends* A (written “A' $>_e$ A”) iff A' $<_{cd}$ A and A' $>_d$ A.

Examples:

a) SQRT1 $<_c$ SQRT2, SQRT1 $<_c$ SQRT3, SQRT1 $<_c$ SQRT4. Note that B' $<_c$ B implies B' $<_{cd}$ B.

b) SQRT3 $<_r$ SQRT1, SQRT4 $<_r$ SQRT1. Note that B' $<_r$ B implies B' $<_c$ B, B' $>_d$ B, B' $<_{cd}$ B, and B' $>_e$ B.

c) SQRT2 $>_e$ SQRT1. According to the definition of $>_e$, B' $>_e$ B implies B' $<_{cd}$ B and B' $>_d$ B.

d) SORT2 $<_r$ SORT1.

e) STOP $<_c$ B and B $>_d$ STOP for every constant behavior B.

f) B $<_c$ ARBITRARY and ARBITRARY $>_d$ B for every constant behavior B with the same set of actions as ARBITRARY.

3.2 Comparison of sets of constant behaviors

We consider in this subsection the comparison of sets of constant behaviors, where each constant behavior is characterized by a set A of offered actions. We consider a set **A** = {A_1,...,A_n} of n constant behaviors and want to compare it with a set **A'** = {A_1,...,$A_{n'}$} of n' constant behaviors. These considerations apply to the comparison of two history independent behaviors characterized by **A** and **A'**, respectively. It also applies to the case where the sets **A** and **A'** represent two classes or types of behaviors, represented directly by these respective sets.

Definition 3.2: Let B and B' be sets of constant behaviors that are characterized by **A** and **A'**, respectively. We define **A'** Rel **A** iff for all A' □ **A'**, there is A □ **A** such that A' Rel A, where Rel is replaced uniformly by one of the relations $<_c$, $>_d$, $<_{cd}$, $<_r$, or $>_e$, which are interpreted as in Definition 3.1.

Corrolary: **A**' ∏ **A** implies **A**' Rel **A**, where Rel may be any of the relations $<_c$, $>_d$, $<_{cd}$, $<_r$, or $>_e$.

Examples:
a) {SQRT1, SQRT2} $<_c$ {SQRT2}, and $<_c$ {ARBITRARY}, but not $<_c$ {SQRT1} .
b) any set $<_c$ {ARBITRARY} $<_c$ CHAOS and CHAOS $<_c$ {ARBITRARY}, if the behaviors have the same set of actions.

Note: In this context CHAOS denotes the set **A** characterizing the CHAOS behavior; and ARBITRARY denotes the set A of actions characterizing the ARBITRARY behavior.

3.3 Comparison of general behaviors

For the general case of behaviors as defined by Definition 2.2, we define corresponding relations for the comparison of behaviors as follows.

Definition 3.3: Let B and B' be arbitrary behaviors, as considered in Definition 2.2. We define B' Rel B iff for any trace t, (t,**A**') □ B' and (t,**A**) □ B implies **A**' Rel **A** holds, where Rel is replaced uniformly by one of the relations $<_c$, $>_d$, $<_{cd}$, $<_r$, or $>_e$, and **A**' Rel **A** is interpreted as in Definition 3.2.

This definition covers the special cases of constant and history independent behaviours discussed in the subsections above.

Examples:
a) CM2 $<_c$ CM1, CM1 $>_d$ CM2, and CM3 $>_e$ CM2.
b) QUEUE1 $<_c$ QUEUE2, and QUEUE2 $>_e$ QUEUE1.
c) STOP $<_c$ B and B $>_d$ STOP for every behavior B.
d) B $<_c$ ARBITRARY and ARBITRARY $>_d$ B for every behavior B with the same set of actions as ARBITRARY.
e) ARBITRARY $<_r$ CHAOS, if ARBITRARY and CHAOS have the same set of actions.

3.4 Discussion

Corollary: (The proof of these statements is given in the Appendix)
a) Let B, B' be behaviors. Then B' $<_c$ B implies traces(B') ∏ traces(B).
b) Let B, B' be behaviors. Then B' $<_c$ B implies B' $<_{cd}$ B.
c) Let B, B' be behaviors. Then B' $<_r$ B implies B' $>_e$ B.
d) The relations $<_c$ and $<_r$ are transitive.
e) The relation $>_d$ is transitive for constants, and sets of constant behaviors (history independent behaviors). It is not transitive for deterministic behaviors in general.

f) The relation $<_{cd}$ is not transitive for constant deterministic behaviors (and therefore neither for more general behaviors).
g) The relation $>_e$ is transitive for constants, sets of constant behaviors (history independent behaviors), and state deterministic behaviors. It is not transitive in general.
h) Let B, B' be deterministic behaviors. Then B' $<_r$ B iff B = B'.

We note that corollary (g) shows that the only room left for specializing object behavior according to the reduction relation "$<_r$" is the reduction of nondeterminism, either the reduction of undetermined output or the reduction of state nondeterminism.

Corollary (c) shows that $<_r$ is stronger than $>_e$. Under $>_e$, it is possible to reduce the number of *unspecified receptions* and to extend the functionality by adding actions, which is prohibited by $<_r$. This corresponds to "specialization" under the *"blocking by default"* convention, where the unspecified cases are assumed to block. For example, if we want to add a new functionality defined by the LOTOS expression "S' = d; b; stop" to the expression "S = a; b; stop [] c; stop" mentioned in Section 2.3, which we may write in the form "S" = a; b; stop [] c; stop [] d; b; stop", the behavior B", denoted by S", is an extension (but not a reduction) of both of the two behaviors B and B', i. e. B" $>_e$ B and B" $>_e$ B', where B and B' are the behaviors denoted by S and S', respectively.

However, if we assume the *"undefined by default"* convention, where for the unspecified cases all actions and all outputs are allowed, the adding of a new functionality corresponds to the reduction $<_r$. For the example of the expressions S, S' and S" above, the behavior B" is a reduction of both B and B', i. e. B" $<_r$ B and B" $<_r$ B', since the alphabet of S includes the alphabets of both S and S'. We therefore conclude that the *extension* relation is the natural specialization relation in the context of the *blocking by default* convention, while the *reduction* relation is natural in the context of the convention *undefined by default.*

In the context of actions without input nor output parameters, which correspond to models based on labelled transitions systems and languages such as CCS and Basic LOTOS, the relations defined above correspond to certain relations defined in the literature. In particular, we have the following:
(i) B' $>_d$ B is equivalent to saying that B' "conforms" to B, as defined in [Brinksma, 86 #301].
(ii) B' $<_r$ B implies that B' is a "reduction" of B, as defined in [Brinksma, 86 #301].
(iii) B' $>_e$ B implies that B' is an "extension" of B, as defined in [Brinksma, 86 #301].

4 Constructing specifications

In the discussion above, it has implicitly been assumed that several specifications exist, let us say S1 and S2, and that one wants to determine whether certain specialization relations hold between the behaviors denoted by them. Based on the relations defined between behaviors, one may introduce corresponding specialization relations between specifications. For instance, the relation $<_c$ (constrainment) would be defined to hold between the specifications S1 and S2 iff the same relation holds between the behaviors denoted by the respective specifications. One can then directly speak about specialization relations between specifications.

The reduction relation between a specification S2 and a specification S1 may imply, for instance, that a module satisfying S2 may replace a module satisfying S1 in the context of a given system. In many situations, the two specifications are given and one wants to verify the relation between them a-posteriori. In the process of system development, however, it would often be more useful if one had some method by which one can construct a specification S2 from a given specification S1 such that the relation is satisfied a priori (by construction).

Some work in this direction is reported in [Gotzhein, 92c #1084], where the modification of Estelle specifications is considered in relation with the specialization relations that hold between the old and new versions of a specification. The complete Estelle language is supported, however, the considered changes relate mainly to the finite-state-machine aspects of the specifications. Another problem is considered in [Khendek, 92a #1047], where one assumes that two behavior definitions B1 and B2 are given and that one wants to construct a new behavior B3 which is a specialization of both, B1 and B2. The specialization relation considered is the extension relation of [Brinksma, 86 #301]. In [Bochmann, 91c #284], it is shown that the parallel composition of B1 and B2 leads to a behavior which satisfies the constrainment relation $<_c$ in respect to both B1 and B2. Similar considerations can also be found in [Erdogmus, 90 #1092].

In the context of type checking in strongly typed programming languages, one is not concerned with the detailed behavior of functions, but only with the possible range of input and output parameters. Most programming languages include features for constructing new types from already given ones. The constructed types satisfy certain relations as discussed in the following subsection.

4.1 Defining types of behaviors

In this section we consider "types" of behavior, which we assume to be the same as a set of behaviors. While most of the discussions relate to sets of constant behaviors, the considerations may be generalized for more complex behaviors as described by Definition 2.2. In the following, we use the convention that variables representing sets of behaviors are written in **bold**.

Definition 4.1: A *type* is a set of behaviors.

The notion of type is used in the context of programming and specification languages, where the notion of strong typing corresponds to the verification of certain relations between the types of variables and other expressions within a given program or specification. In the following we discuss the most common approaches to defining useful types of behaviors, as typically provided by object-oriented typed languages (see for instance [Meyer, 88 #727]).

Type construction by enumeration of behaviors: A simple method of describing a type is by enumerating the behaviors which are included in it.

Type construction by constructor operations and associated axioms: In many cases, the number of object behaviors within a class are too large to take the enumeration approach to type definition. Using an algebraic approach to the definition abstract data types, a type is defined by a certain number of basic constants with one or more constructor operations and associated axioms about the results returned by the constructor and other operations on the basic constants and (recursively) on the results of the operations. A well-known example is the class of Integers with the basic constant *zero* and the constructor operation *succ* .

Once a certain number of base types are defined, it is possible to define other types by stating the properties satisfied by all behaviors within that type. The following definitions are examples of such class definitions.

Type construction by functional range: The set of behaviors for which the results of the operation x are confined to the type **R** is defined as { B | f(i;o) □ B implies o □ **R**}. This set of behaviors can also be defined as the set of behaviors that are constrained by the behavior Range(f, **R**) (formally {B | B $<_c$ [3] Range(f, **R**)}), where Range(f, **R**) is the behavior which includes for all inputs i and all outputs o □ **R** the action f(i;o), and which is "undefined by default" for all other operations.

Type construction by functional domain: The set of behaviors including an operation with a given name f and covering a given domain of input parameter behaviors **D**, is defined as { B | for all i □ **D,** there exists o with f(i;o) □ B }. This set of behaviors can also be defined as the set of behaviors that cover the domain of the behavior Domain(f, **D**) (formally {B | B $>_d$ Domain(f, **D**)}), where Domain(f, **D**) is the behavior which includes for all outputs o and all inputs i □ **D** the action f(i;o), and which is "undefined by default" for all other operations. (We note that this implies that a behavior of this type will never block for the operation f with an input parameter in **D**).

Type construction by functional signature: We call a *functional signature* f<**D**,**R**> the definition of an operation name f, and two types **D** and **R**. For a given functional signature, we define the type of behaviors that *satisfy the functional signature* to be the class **FunSig**(f<**D, R**>) = {B | B $<_c$ Range(f, **R**) and B $>_d$ Domain(f,**D**)}.

Note: The class of mathematical functions, named f, from domain **D** to range **R** is a subset of **FunSig**(f<**D**,**R**>); namely those elements of this set with no output nondeterminism for all elements of the domain and no action outside this domain. The concept of "partial functions", which have no defined result for certain elements of the domain, may be interpreted in several different manners, as discussed in Section 2.4.

[3] In this context, the relations $<_c$ and $<_{cd}$ are equivalent since the domain of Range(f, **R**) is maximal.

The above definitions lead to the well-known theorem (see for instance [Black, 87 #104]) about the contravariant typing relationship between the types of functions and their input parameters, which may be stated as follows.

Proposition 4.1 (functional subtyping): Given two functional signatures f<**D**,**R**> and f<**D'**,**R'**> for the same operation name f, the relations "**R' ∏ R**" and "**D ∏ D'**" imply that the set of behaviors satisfying f<**D'**,**R'**> also satisfy f<**D**,**R**>, that is, **FunSig**(f<**D', R'**>) ∏ **FunSig**(f<**D, R**>).

Type construction by object signature: As usual in object- oriented languages, we define an *object signature* to be a set of functional signature definitions with disjoint operation names. The set of operation names is called the *alphabet* of the object signature. For a given object signature s, we define the type of behaviors that *satisfy the signature*, written "**Sig(s)**", to be the intersection of all the **FunSig**(f<**D**,**R**>) where f<**D**,**R**> are the functional signatures included in s.

This definition implies the following theorem which corresponds to the well-known convention of object-oriented languages, that by adding a new operation to a given object signature, one obtains a corresponding type of behaviors which is a subtype (i.e. subset) of the original one. This can be stated as follows.

Proposition 4.2 (object-oriented subtyping): If the object signature s' is obtained from a signature s by the addition of the functional signature f<**D**,**R**>, then **Sig(s') ∏ Sig(s)**.

5 Specification of optional behaviors

We consider in the following that a system specification includes a module A for which a requirements specification SA is given. An implementation IA of that module should satisfy the requirements SA. We discuss in this section what the meaning of "IA satisfies SA" should be.

In many cases, the requirements define precisely the behavior of the implementation. In this case, "IA satisfies SA" could mean "behavior of IA = behavior denoted by SA" and we assume that the behaviors a defined in the framework discussed in Section 2. In many cases, however, the requirements allow for many different implementation behaviors. In this case we may assume that a specification denotes a specific behavior, and we say that the statement "IA satisfies SA" means that "behavior of IA Rel behavior denoted by SA" where Rel is one of the specialization relations discussed in Section 3. The reduction and extension relations seem to be good candidates for this purpose.

It seems, however, that in many cases one needs even more flexibility than provided by the scheme above. In particular, often a distinction is made between those behavior aspects which an implementation *must* perform and other aspects which are optional. This distinction has been made in the context of transition systems by Larson who distinguishes between MUST and MAY transitions [Borgesson, 92 #1093]. Such a distinction can also be introduced by distinguishing between safety and liveness assertions [Alpern, 87 #16]. Safety states that "nothing bad can happen", while liveness states that "certain good things will eventually happen"; these "good things" that are promised may be less that whatever is allowed to be implemented according to the safety constraints.

We notice that the specialization relations reduction and extension include two aspects, the constrainment relation $<_c$ or $<_{cd}$ and the domain coverage relation $>_d$. We think that the former plays the role of "safety" which indicates what may be implemented, and that the domain coverage relation defines what *must* be covered. If these two aspects are not always the same, we have to foresee that they may be specified independently. We are therefore led to the following definition of a specification.

Definition 5.1:

a) A *specification* S denotes (according to the semantics of the specification language) a pair (B_c, B_d) of two behaviors B_c and B_d. We write Sem_c (S) for B_c , and Sem_d (S) for B_d.

b) A behavior B *satisfies* a specification S in *respect to reduction* iff B $<_c$ Sem_c (S) and B $>_d$ Sem_d (S).

c) A behavior B *satisfies* a specification S in *respect to extension* iff B $<_{cd}$ Sem_c (S) and B $>_d$ Sem_d (S).

As noted above, the two behaviors Sem_c (S) and Sem_d (S) may be the same in many situations, but in general, they could be different. We assume in general that Sem_d (S) $<_c$ Sem_c (S) holds.

As an example we consider the behavior of the communication service COMM1 denoted by the diagram of Figure 1a. Since the objective of the service is to provide data transfer in the open state, the action con(;refuse) is not really required. We may therefore consider that the implementation of this action is optional, that is, a behavior as shown in Figure 1d is also acceptable. We therefore want a specification S such that Sem_c (S) is as shown in Figure 1a, and Sem_d (S) is as shown in Figure 1d. Such a specification may be given in the form of a diagram as shown in Figure 1e, or in the following linear specification which is written in an extended version of LOTOS:

```
Closed[con,data]  where
process Closed[con,data] : noexit =   con !accept ; Open[con,data]
                                       []  OPTIONAL con!refuse ; Closed[con,data]  endproc
process Open[con,data] : noexit = data !ack ; Open[con,data]   endproc
```

Another example of the use of these concepts can be found in the context of strongly typed programming languages, where the signature of a function indicates the domain of the input parameters for which the function must be defined and the range of the possible results. When the compiler type-checks a function definition, it verifies that the function definition *satisfies* the signature (although in practice, the domain coverage is usually not checked). Therefore, the type declaration of a function (i.e. the definition of its functional signature) is a *specification*, where the behaviors Sem_c and Sem_d are the behaviors **Range** and **Domain**, respectively, as defined in Section 4.1. The type checking of object-oriented programs [Black, 87 #104] follows the same principles.

6 Conclusions

We have shown that a generalized *reduction* relation may be used to compare object behaviors covering a wide range of concerns, including set of values, functions, object-oriented class definitions, and sequential machines possibly including nondeterminism. This relation is based on two more basic relations for the comparison of object behaviors, namely (a) constrainment, which is based on the notion of actions offered by an object as a function of the trace of actions executed in the past, and (b) domain coverage, which is based on the notion of absense of blocking for certain kinds of actions.

This relation also provides a framework for the definition of requirement specifications, the conformance between implementations and specifications, as well as for the comparison between specifications, which is important for managing the multiple inheritance subtyping lattice of object-oriented specifications and for questions of module replacement and reutilisation. The relation may also be used within a framework for the dynamic modification of systems [Erradi, 92g #1089] where the reduction relation assures that the modified system conserves its important properties after any modifications introduced.

It is shown that our notion of *reduction* is a generalization of the reduction relation as defined for transition systems [Brinksma, 86 #301]. Also the *extension* relation defined in that context allows for a generalization, which is in fact quite closely related to the generalized notion of reduction. It is interesting to note that both notions may be considered to be natural representation of "specialization", depending on the assumption made about undefined behavior aspects (i.g. those aspects not explicitly defined).

Some work on the construction of behavior definitions which satisfy certain specialization relations with given behaviors has been reported in the introductory part of Section 4. Futher work is required in order to make these approaches more practical and to facilitate their practical use in the software and system development process.

In Sectio 5, we have introduced the notion of a specification which consists of two parts, one representing the safety requirements (a kind of "maximum" behavior) and the other representing the "minimum" behavior requirements which represents a minimum implementation. This is in contrast to most specification languages which assume that a specification is the definition of a *single* "behavior". Further work is needed to explore the relevance of our more complex notion of specification, and to find an appropriate notation for representing the two parts within a single syntactical structure, which should ideally be an extension of the notation used by an existing specification language for writing a single behavior definition.

Acknowledgements: The authors gratefully acknowledge clarifying discussions with, and comments on earlier versions of the manuscript by A. Das, R. Dssouli, M. Erradi, F. Khendek, M. Yao.

References

[Alpe 87] B. Alpern and F. B. Schneider, *Recognizing safety and liveness*, Distributed Computing 2 (1987), pp. 117-126.

[Amer 87b] P. America, *Inheritance and subtyping in a Parallel Object-Oriented Language*, in Proc. of Europ. Conf. on Object-Oriented Progr. (AFCET), 1987, pp. 281-289.

[Amer 89] P. America, *A behavioural approach to subtyping in object-oriented programming languages*, Philips J. Res. (Netherlands), Vol. 44, Nos. 2-3, pp.365-383, 1989.

[Beli 89] F. Belina and D. Hogrefe, *The CCITT-Specification and Description Language SDL*, Computer Networks and ISDN Systems, Vol. 16, pp.311-341, 1989.

[Blac 87] A. Black, N. Hutchinson, E. Jul, H. Levy and L. Carter, *Distribution and Abstract Types in Emerald*, IEEE Trans. on Software Engeneering, vol. se-13, no. 1, january 1987.

[Boch 91c] G. v. Bochmann, *On the specialization of object behaviors*, révision de la publication départementale P#687.

[Borg 92] A. Borgesson, K. Larson and A. Skou, *Generality in design and compositional verification using Tav*, Proc. FORTE'92, Formal Description Techniques V, North Holland Publ., 1993.

[Brin 86] E. Brinksma and G. Scollo, *Lotos specifications, their implementations and their tests*, Protocol Specification, Testing and Verification VI (IFIP Workshop, Montreal, 1986), North Holland Publ., pp. 349-360.

[Card 88a] L. Cardelli, *A semantics of multiple inheritance*, Information and Computation 76 (1988), pp. 138-164.

[Cusa 89] E. Cusack, *Refinement, conformance and inheritance*, Workshop on Theory and Practice of Refinement, Open Univ., UK, Jan. 1989.

[Erdo 90] H. Erdogmus, H. Hakan and R. Johnston, *On the specification and synthesis of communicating processes*, IEEE Trans. SE, Vol. 16, 12 (Dec. 1990), pp.

[Erra 92g] M. Erradi, R. Dssouli and G. v. Bochmann, *A framework for dynamic evolution of distributed systems specifications*, Réseaux et Informatique Répartie, to be published.

[Gotz 92c] R. Gotzhein, *Temporal logic and applications - A tutorial*, Computer Networks and ISDN Systems 24 (1992), pp. 203-218.

[Hoar 85] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985.

[Loto 89a] ISO, *IS8807 (1989), LOTOS: a formal description technique*,

[Khen 92a] F. Khendek and G. v. Bochmann, *Incremental construction approach for distributed system specifications*, submitted for publication.

[Meye 88] B. Meyer, *Object Oriented Software Construction*, C.A.R. Hoare Series Editor, Prentice Hall, 1988.

[DeNi 87] R. D. Nicola, *Extensional Equivalences for Transition Systems*, Acta Informatica, 24 (1987), 211-237.

Appendix: Proofs for Corollary of Section 3.4

a) Let B, B' be behaviors. Then B' $<_c$ B implies traces(B') $\prod$ traces(B).

Proof:

Assume that the proposition does not hold. It follows that there must be a (non-empty) trace t = t'^<a> s. t. t□traces(B'), t□traces(B), and t'□traces(B). Since B and B' are behaviors, we have **A**, **A'** such that (t',**A**)□B and (t',**A'**)□B', and there is A'□**A'** such that a□A'. From B' $<_c$ B, it follows that **A'** $<_c$ **A** holds, which implies that for A', there is A□**A** such that A'$\prod$A. It follows that a□A, and therefore, t'^<a> □ traces(B). This contradicts our assumption, therefore, the proposition holds.

b) Let B, B' be behaviors. Then B' $<_c$ B implies B' $<_{cd}$ B.

Proof:

It suffices to show that A' $<_c$ A implies A' $<_{cd}$ A. Assume that this does not hold. Then for some f(i) accepted by A, there is an output o such that f(i;o)□A' and f(i;o)□A. Therefore, A'$\prod$A does not hold, which contradicts our assumption.

c) Let B, B' be behaviors. Then B' $<_r$ B implies B' $>_e$ B.

Proof:

Follows immediately from b) and the definitions of $<_r$ and $>_e$.

d) The relations $<_c$ and $<_r$ are transitive in general.

Proof for $<_c$:

Follows immediately from the definitions.

Proof for $<_r$:

Recall that for arbitrary behaviors B, B', we have B' $<_r$ B iff B' $<_c$ B and B' $>_d$ B. Since $<_c$ is transitive in general (see d), it suffices to show

(i) B" $<_c$ B' □ B' $<_c$ B □ B" $>_d$ B' □ B' $>_d$ B □ B" $>_d$ B

in order to prove the transitivity of $<_r$. From a) and B" $<_c$ B', it follows that traces(B") $\prod$ traces(B'), therefore, it suffices to prove

(ii) traces(B") $\prod$ traces(B') □ B" $>_d$ B' □ B' $>_d$ B □ B" $>_d$ B

From traces(B") $\prod$ traces(B'), we obtain

(iii) ∀t,**A**". ((t,**A**")□B" □ **A**'. (t,**A**')□B')

From B'' $>_d$ B' □ B' $>_d$ B, we get

(iv) ∀t,**A**,**A'**,**A''**. ((t,**A**)□B □ (t,**A'**)□B' □ (t,**A''**)□B'' □ **A''** $>_d$ **A'** □ **A'** $>_d$ **A**)

Since $>_d$ is transitive for sets of constant behaviors, we get

(v) ∀t,**A**,**A'**,**A''**. ((t,**A**)□B □ (t,**A'**)□B' □ (t,**A''**)□B'' □ **A''** $>_d$ **A**)

Together with (iii), we obtain

(vi) ∀t,**A**,**A''**. ((t,**A**)□B □ (t,**A''**)□B'' □ **A''** $>_d$ **A**)

which is the definition of B'' $>_d$ B.

e) The relation $>_d$ is transitive for history independent behaviors, but not for deterministic behaviors, state deterministic behaviors, and general behaviors.

Proof:

Transitivity for history independent behaviors follows immediately from the definitions. To prove that it is not transitive for deterministic behaviors, we give a counter example. Define B, B', and B'' as follows:

B = { (<>,{{f(i;o)}}), (<f(i;o)>,{{f(i;o)}}), (<f(i;o) f(i;o)>, {{}}) }

B' = { (<>,{{f(i;o')}}), (<f(i;o')>,{{}}) }

B'' = { (<>,{{f(i;o)}}), (<f(i;o)>,{{}}) }

B'' $>_d$ B' and B' $>_d$ B hold, but B'' $>_d$ B does not, because after trace <f(i;o)>, B'' blocks for f(i), but B accepts f(i). Since B, B', and B'' are deterministic behaviors, the relation $>_d$ is also not transitive for state deterministic behaviors and general behaviors.

f) The relation $<_{cd}$ is not transitive for constant deterministic behaviors, constant behaviors, history independent behaviors, deterministic behaviors, state deterministic behaviors, and general behaviors.

Proof:

To prove that $<_{cd}$ is not transitive for constant deterministic behaviors, we give a counterexample. Define B, B', and B'' as follows:

B = A = { f(i;o) }

B' = A' = { }

B'' = A'' = { f(i;o') }

B'' $<_{cd}$ B' and B' $<_{cd}$ B hold, but B'' $<_{cd}$ B does not, because for f(i), o' is not an output of B. From this, it follows that $<_{cd}$ is also not transitive for constant behaviors, history independent behaviors, deterministic behaviors, state deterministic behaviors and general behaviors.

g) The relation $>_e$ is transitive for history independent behaviors, state deterministic behaviors, deterministic behaviors and constant deterministic behaviors, but not in general.

<u>Proof:</u>

Notation: Let A be a set of actions. $oi(A) =_{Df} \{f(i) \mid \exists o.\ f(i;o)\square A\}$. $A_{f(i)} =_{Df} \{f(i;o) \mid f(i;o)\square A\}$.

I. $>_e$ is transitive for history independent behaviors.

Let **A**, **A'**, **A"** be history independent behaviors. Recall that for history independent behaviors **A**, **A'**, we have **A'** $>_e$ **A** iff **A'** $<_{cd}$ **A** and **A'** $>_d$ **A**. Since $>_d$ is transitive for history independent behaviors (see e), it remains to be shown

(i) $\mathbf{A"} >_e \mathbf{A'} \square \mathbf{A'} >_e \mathbf{A} \square \mathbf{A"} <_{cd} \mathbf{A}$

Exploiting the definition of $>_e$ and using the above notation, the antecedent is the following:

(ii) $\forall A".(A"\square\mathbf{A"} \square \exists A'.(A'\square\mathbf{A'} \square oi(A') \prod oi(A") \square \forall f,i.(f(i)\square oi(A') \square A"_{f(i)} \prod A'_{f(i)}))) \square$
$\forall A'.\ (A'\square\mathbf{A'} \square \exists A.\ (A\square\mathbf{A} \square oi(A) \prod oi(A') \square \forall f,i.(f(i)\square oi(A) \square A'_{f(i)} \prod A_{f(i)})))$

Since $A"_{f(i)} \prod A'_{f(i)}$ holds for all $f(i)\square oi(A')$, it also holds for all $f(i)\square oi(A)$, because $oi(A) \prod oi(A')$. Therefore, we obtain

(iii) $\forall A".(A"\square\mathbf{A"} \square \exists A,A'.(A'\square\mathbf{A'} \square A\square\mathbf{A} \square \forall f,i.(f(i)\square oi(A) \square A"_{f(i)} \prod A'_{f(i)} \prod A_{f(i)})))$

This leads us to

(iv) $\forall A".(A"\square\mathbf{A"} \square \exists A.(A\square\mathbf{A} \square \forall f,i.(f(i)\square oi(A) \square A"_{f(i)} \prod A_{f(i)})))$

which is the definition of **A"** $<_{cd}$ **A**.

II. $>_e$ is transitive for state deterministic behaviors, deterministic behaviors, and constant deterministic behaviors.

In the following, B, B', and B" denote state deterministic behaviors, where B" $>_e$ B' and B' $>_e$ B hold. We begin by proving the following lemma:

(i) $traces(B) \leftrightarrow traces(B") \prod traces(B')$

Assume that (i) does not hold. Then there must be a (non-empty) trace t = t'^<a> s. t. $t\square traces(B)\leftrightarrow traces(B")$, $t\square traces(B')$, and $t'\square traces(B')$, where a=f(i;o). Since B, B', and B" are state deterministic, there are uniquely determined sets A, A', and A" such that $(t',A)\square B$, $(t',A')\square B'$, and $(t',A")\square B"$. If $t\square traces(B)\leftrightarrow traces(B")$ and $t\square traces(B')$, then $a\square A\leftrightarrow A"$ and $a\square A'$. We observe the following:

- From B" $>_e$ B' $\square$ $f(i;o)\square A"$ $\square$ $f(i;o)\square A'$, it follows that $f(i)\square oi(A')$, otherwise, B" would not be constrained by B' on its domain.
- From B' $>_e$ B $\square$ $f(i;o)\square A'$ $\square$ $f(i;o)\square A$, it follows that $f(i)\square oi(A')$, otherwise, B' would not cover the domain of B.

These observation are in contradiction, therefore, (i) holds.

Exploiting and simplifying the definition of $>_e$ and using the above notation, B'' $>_e$ B' and B' $>_e$ B can be rewritten as:

(ii) ∀t,A',A''.((t,A')□B' □ (t,A'')□B'' □ ∀f,i.(f(i)□oi(A') □ f(i)□oi(A'') □ $A''_{f(i)}$ ∏ $A'_{f(i)}$))) □
∀t,A,A'. ((t,A)□B □ (t,A')□B' □ ∀f,i.(f(i)□oi(A) □ f(i)□oi(A') □ $A'_{f(i)}$ ∏ $A_{f(i)}$)))

Together with (i), B'' $>_e$ B follows immediately. Note that from this result, it follows that $>_e$ is also transitive for deterministic and constant deterministic behaviors.

III. $>_e$ is not transitive in general.

To prove that $>_e$ is not transitive in general, we give a counterexample. Define B, B', and B'' as follows:

B = { (<>,{{},{f(i;o)}}), (<f(i;o)>,{{f(i;o)}}), (<f(i;o) f(i;o)>, {{}}) }
B' = { (<>,{{},{f(i;o')}}), (<f(i;o')>,{{}}) }
B'' = { (<>,{{f(i;o)}}), (<f(i;o)>,{{}}) }

B'' $>_e$ B' and B' $>_e$ B hold, but B'' $>_e$ B does not, because after trace <f(i;o)>, B'' blocks for f(i), but B accepts f(i).

h) Let B, B' be deterministic behaviors. Then B' $<_r$ B iff B = B'.

<u>Proof:</u>

Since a deterministic behavior is characterized by the set of its traces, it suffices to show B' $<_r$ B iff traces(B) = traces(B').

Notation: Let A be a set of actions. oi(A) $=_{Df}$ {f(i) | ∃o. f(i;o)□A}.

I. B' $<_r$ B □ traces(B') ∏ traces(B)

Follows immediately from a) and the definition of $<_r$.

II. B' $<_r$ B □ traces(B) ∏ traces(B')

Assume that this does not hold. Then there must be a (non-empty) trace t = t'^<a> such that t'□traces(B)↔traces(B'), t□traces(B), and t□traces(B'), where a=f(i;o). Since B and B' are deterministic behaviors, the trace uniquely determines the set of offered actions. Let A, A' be the set of offered actions of B, B' after t', respectively. From t□traces(B), we can conclude that a□A and f(i)□oi(A). From B' $<_r$ B, we have oi(A) ∏ oi(A'), therefore, f(i)□oi(A'). This implies that there is o' such that f(i,o')□A'. However, o□o', because f(i;o)□A'. Since traces(B')∏traces(B), f(i;o')□A must hold. It follows that B has an undetermined output for f(i) after t', which contradicts our assumption that B is a deterministic behavior.

III. traces(B) = traces(B') □ B' $<_c$ B

Assume that this does not hold. Then there must be a (non-empty) trace t□traces(B) and sets A, A' such that (t,A)□B, (t,A')□B', and ¬(A'∏A). Therefore, we have an action a such that a□A' and a□A. Since B' is a behavior, t^<a>□traces(B'). B is a deterministic behavior, therefore, the set A of offered actions after t is uniquely determined. If a□A, then t^<a>□traces(B). It follows that traces(B) □ traces(B'), which contradicts our assumption.

IV. traces(B) = traces(B') □ B' $>_d$ B

For all t□traces(B), there are uniquely determined sets A, A' of offered actions after t for B, B', respectively. Since B, B' are deterministic and traces(B) = traces(B'), A = A' holds for all traces t□traces(B) and the corresponding sets A, A' of offered actions, which implies B' $>_d$ B.

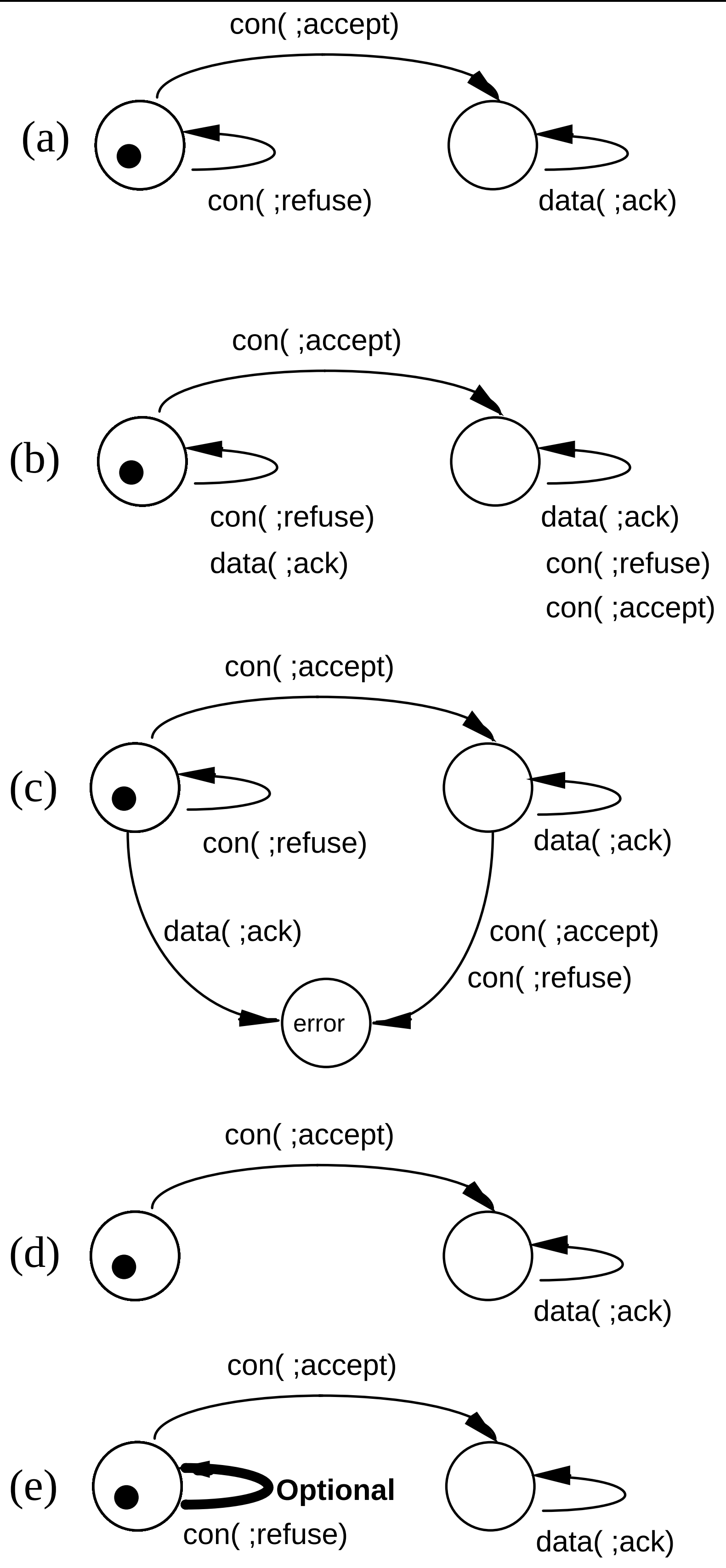

Figure 1: Different versions of a communication service access point

References