プログラミング-言語・基礎・実践-　16－6
(1994. 3. 9)

分散システム仕様の動的進化に対する枠組
オブジェクトふるまいの専門化と必要条件仕様記述

アイサム. A. ハミド
東北芸術工科大学
情報デザイン学科
山形市上桜田200番地

あらまし　本論では、実行可能な仕様記述法の開発と動的変更を考慮した形式的記述の技術に論点を置く。大規模なオブジェクト指向仕様記述を進化させるために２つのレベルのモデルを導入する。第１のレベルは、型（クラス）の動的変更を扱う。一方、第２のレベルはモジュールの変更を扱う。両方のレベルでその変更後、仕様記述の一貫性を保証するために、構造的かつ振る舞い的制約の集合を定義する。型とモジュールの動的変更を可能にするために、我々は変更の操作を支援するメタオブジェクトを用いるリフレクティブなオブジェクト指向仕様記述言語を開発した。この言語において型とモジュールはオブジェクトである。

和文キーワード　オブジェクト指向仕様記述、ソフトウェア進化、型変更、モジュール互換性、リフレクション、動的変更

Dynamic Evolution of Distributed Systems Specifications
Specialization of Object Behaviors and Requirement Specifications

Issam A. Hamid
Tohoku University of Art & Design
Department of Information Design
200 Kamisakurada, Yamagata city, JAPAN

Abstract　Given two behavior descriptions, the question whether one is a specialization of the other is important in many situations. Sometimes it must be checked whether the behavior of an implementation is a specialization of the specification. In this paper, we are concerned with formal description techniques that allow for the development and the dynamic modification of executable specifications. A two-level model for the evolution of large object-oriented specifications is introduced. The first level deals with the dynamic modifications of types (classes), while the second level deals with modifications of modules. We define a set of structural and behavioral constraints to ensure the specification consistency after its modification at both levels. To allow for dynamic modification of types and modules, we develope a reflective object-oriented specification language which use meta-objects to support the modification operations.

英文 key words　object-oriented specifications, software evolution, modules compatibility, reflection, dynamic modifications.

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 refinement. We may say, for instance, that the detailed design description is a "specialization" of the functional specification. Another kind of specialization becomes important if a product must be adapted to several different user requirements. In that case, a more general "generic" system may be designed, and the design, as well as the corresponding implementation, may be adapted to each different user requirements. In this case, each of these "specific" designs and implementations may be considered as a "specialization" of the generic case. We are concerned in this paper with these two kinds of "specializations".

In the context of object-oriented programming languages, two kinds of "specializations" have been considered[Amer 87b]: (1) code-sharing, usually called "inheritance", which is based on concept 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 is based on the concept that certain properties, in particular the interface, of the subclass remain valid for the object instances of a subclass. In this paper, we only consider the second aspect, since our attention is more directed towards specification languages, which define properties of object instances and classes.

Another important concept related to "specialization" is sometimes called "conformance", namely in the sense that a class C' conforms with a class C if an object instance of class C can be replaced, within the overall system, by an instance of the class C' without invalidating certain important system properties. Certain papers explore this question in the context where the important property is type checking [Blac 87][Card 88]. Other papers address the comparison of object behaviors and their influence on the overall system behavior [Cusa 89][Amer 89].

The purpose of this paper is to show that (1) a unified concept of "specialization" can be used in all the above contexts, and that (2) this concept corresponds to different notions of "specialization" which have been introduced in the context of different specification languages. These notions include 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.

(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. Our notion of specialization corresponds to 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.

In the following sections, we show that the above concepts of specialization may be considered special cases of the reduction relation. It is to be noted that the notion of reduction includes 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 two aspects will be developed separately in this paper. They intuitively ressemble the notions of "safeness" and "liveness" in[Alpe 87], although there are certain differences, as discussed in Sec. 5.1.

2. Objects

We consider that a real system consists of a number of objects (also called object instances). Each object has a behavior, in the following also called "temporal behavior", which characterizes its interactions with other objects within the system during its lifetime. There may be a large number of object instances that have the same behavior. Therefore we will discuss in the following mainly behaviors and the comparison between different behaviors.

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. Sometimes, the requirement could be the statement that the behavior of the object instances must be equivalent to a given behavior. However, more freedom is often allowed for the implementation. In such cases, it seems convenient to define the requirement in terms of a set of behaviors where the actual behavior of the specified object instances must belong to this set. We use the term "class" to denote a set of behaviors.

In general, the behavior offered by an object depends on its past interactions. We also say that the past interactions determine its "state" and that its state determines its future behavior. Those objects for which the offered behavior does not depend on its past interactions are called "constant" objects, and their behavior "constant behaviors". They form an important class and are discussed in this sections. Object behaviors involving state changes are discussed in Sec. 5.

Note: In some typed object-oriented languages one says that each object instance has a type (which defines its behavior). In this paper, we do not use the term "type"; however, it seems that our term "behavior" has a very similar meaning. Our term "class" is simply a set of behaviors and may be used to represent a set of related behaviors, such as the different Integer values, or seemingly unrelated behaviors, such as an Apple and a Car.

2.1. Constant objects

Definition (offered behavior): At each instance in time, the behavior of an object is characterized by its offered behavior (in the following also simply called behavior, if there is no ambiguity). The offered behavior of an object is a set of actions. An action is of the form f<i:o>, where f is the name of an operation, i is an object (called effective input parameter), and o is an object (called the result of the operation). If a behavior contains more than one element f<i,o> for a given operation f and given input i (and different results o), we say that the operation has an undeterminate result for this input parameter. If there is no such element, we say that the operation blocks for this input parameter.

Definition (trace): A trace is a sequence of executed actions.

Definition (constant behavior): The temporal behavior of an object is called **constant** iff the offered behavior does not depend on its trace.

Definition: The alphabet of an object O is the set of operation names that are explicitely used in the definition of the object behavior.

Example: The objects T and F (representing a simplified version of the boolean constants True and False, respectively) are constants and have the alphabet {not, and} (if we assume that the operations "or", "imply", etc. are not used). The object T is a constant, and its behavior (in this constant state, which we also denote by T) is

B(T) = { not< :F>, and<F:F>, and<T:T> }

which means that "not(True)" is False, "and" of True and False is False, and "and" of True and True is also True.

Note: In the above example, we have defined explicitely the result of all operations for all possible input parameter values. This is possible if the domain of the input parameters is finite. However, this is often not the case. For instance, we may consider the integer constants as objects with the alphabet {+,-,*,/}. In this case, first of all, the number of objects to be considered is not finite (there is an infinite number of integers), and secondly, for each integer constant, the input parameter considered by the operations may, in turn, be any integer constant. In such cases, the usual algebraic notation involving axioms about the results of operations may be used to define the semantics of such a class of objects.

2.2. Specializing behaviors: discussion

We assume that we have to compare two behaviors B and B'. We say that B' is a specialization of B if the behavior B' is more "precisely" defined than B. For instance, B may leave certain aspects undefined, so-called "don't care" situations; if B' defines these aspects we say that B' is a specialization of B. In the subsequent subsection we define two relations which define more precisely our notion of "specialization". The following discussion and examples are given as an introduction to those definitions.

In the following we define different versions of boolean constants "True" which have different behaviors. The first two constants T1 and T2 below are generalizations of the constant T discussed above. We consider that the alphabet of all this different constants is {not, and}, and we write B(Ti) to denote the behavior of the object Ti.

T1 is a constant "True" accepting arbitrary input parameters: B(T1) = B(T) union

{ not<i:o> } union {and< :o> } union {and<i:o>}

where the i are arbitrary objects, except instances of the True and False constants, and the o are arbitrary objects. The second term, for instance, means that the operation "not" with an input parameter different from True and False (which is not allowed according to the usual definition of the "not" operator) yields a "don't care" result.

T2 is a constant "True" accepting arbitrary operations: B(T2) = B(T1) union {x<i,o> | x not in alpha(T), and any i and o}

where the second term means that any operation not explicitely defined (not included in the alphabet) is accepted by the object and yields a "don't care" result.

Even more "general" objects may be defined, such as the following object which behaves like the True constant, except that the result for "and" is undeterminate (although the result still remains within the class of boolean constants).

T3 is a constant "True" with undeterminate result for the "and" with False: B(T3) = B(T2) union {and<F:T>}

A more realistic example of an undeterminate behavior is a sorting module which is defined as follows: It accepts as input a sequence of pairs and provides as output a sequence which contains the same pairs, but sorted by ascending values of the first elements of the pairs. This specification does not determine the order of pairs in the output which have the same value for their first element.

Another important concept is blocking. We say that the offered behavior of an object blocks for an operation x with input parameter i if there is no element of the form x<i:o> (for arbitrary o) in the behavior. For example, the following version of "True" blocks for the operation "and" with parameter F:T4 is a constant "True" blocking for "and" with False: B(T4) = B(T2) minus {and<F:F>}.

2.3. Specializing behaviors: Definitions

Definition (constraining): Given two behaviors B' and B, we say that B' is constrained by B (written B' <c B) iff each x<i:o> in B' is also included in B.

Definition (domain of operation): The (non-blocking) domain of an operation x for a given behavior B is the set of all i for which B contains at least one x<i:o> (with an arbitrary o).

Definition (domain of behavior): The domain D of a behavior B, written Dom(B), is the set of those pairs (x,i), of operations x and input parameters i, such that B contains at least one x<i:o> (with an arbitrary o). We say that an operation x with input i blocks iff the pair (x,i) is not in the domain.

Definition (domain coverage): Given two behaviors B' and B, we say that B' covers the domain of B (written B' >d B) iff the domain of B is included in the domain of B', i.e., iff Dom(B) included in Dom(B').

Note: Any set of pairs (x,i) which has an empty intersection with Dom(B) may be considered a "refusal set" (in the sense of CSP [Hoar 85]) since all the pairs (x,i) in such a set, block with the given behavior.

Examples: The object behavior that contains no action blocks for all operations and is called "Stop". The other extreme is the behavior that allows "don't care" results for all actions. We call this behavior "Arbitrary". Figure 1 shows the relations between the different kinds of "True" behaviors defined above.

When writing down the definition of a behavior, one usually only wants to consider certain operations which are of particular interest. We assume in the following that these operations are those that are included in the alphabet of the behavior. For the other possible operations, some "default" behavior is assumed. It is important that these default assumptions correspond naturally to the semantic relations of specialization.

In the context of several well-known description methods (e.g. CSP, LOTOS, finite state machines, Petri nets) it is often assumed that actions that are not explicitely defined are not possible. For instance, the environment trying an action which is not defined would be blocked in CSP [Hoar 85] or LOTOS [Loto 89]. We call this convention "blocking by default". As an example, we consider the behavior denoted by the expression "B = a; b; stop c; stop" in LOTOS, which offers (initially) the operations a and c and blocks for all other operations that the environment may wish to execute. Note that the alphabet of this expression is {a, b, c}.

In the following part of this paper, we take another default approach by assuming that operations that are not in the alphabet are "dont' care", that is, they are possible. However, their results are undeterminate, and in the case of objects with state changes, the behavior in the next state would be completely "don't care" as well (see Section 4.2). We call this convention "undefined by default". In the case of the expression "B = a; b; stop c; stop" the denoted semantics is different than in the case of LOTOS, since the "undefined by default" convention denotes a behavior which offers a and c, and blocks for b. And for all other operations, which are not included in the alphabet, the denoted behavior makes an offer with undeterminate result and "don't care" next behavior.

3. Classes of objects

In this section we consider object classes which are characterized by a set of behaviors. While most examples relate to sets of constant behaviors, the definitions given in this section are also valid for the more complex temporal behaviors discussed in Section 4.

Definition (class): A class is a set of behaviors.

Notation: In the following, we use the convention that variables representing sets of behaviors are written in bold.

3.1. Examples of Class Constructions

In practice, one is usually interested in particular classes of behaviors. In the following we discuss the most common approaches to defining useful behavior classes.

Class construction by enumeration of behaviors: A simple method of describing a class is by enumerating the behaviors which are included in the class. For example, using the convention "undefined by default", we may define the class Boolean to be the set of object behaviors T2 and F2 (as defined in Section 2.2).

Class 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 class definition. The approach followed for algebraic specifications of abstract data types descibes a class by defining 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 operations "succ" and "pred"..

Note: The above approaches to class definition correspond to the standard view in programming langugages that a class (also called "type") is a set of constants (also called "values").

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

Class construction by functional range: The set of behaviors for which the results of the operation x are confined to the class **R** is defined as { B I x<i:o> contained in B implies o contained in R}. This set of behaviors can also be defined as the set of behaviors that are constrained by the behavior Range(x, R) (formally {B I B <c Range(x, R)}), where Range(x, R) is the behavior which includes for all inputs i and all outputs o included in R the action x<i:o> (and which is "don't care" for all other operations).

Class construction by functional domain: The set of behaviors including an operation with a given name x and covering a given domain of input parameter behaviors D, written "Domain(x, D)", is defined as { B I for each i contained in D, there exists o such that x<i:o> contained in B }. This set of behaviors can also be defined as the set of behaviors that cover the domain of the behavior Domain(x, D) (formally {B I B >d Domain(x, D)}), where Domain(x, D) is the behavior which includes for all outputs o and all inputs i included in D the action x<i:o> (and which is "don't care" for all other operations).

Note: The above definition implies that a behavior of this class will never block for the operation x with an input parameter in D.

Class construction by functional signature: We call a **functional signature** x<D,R> the definition of an operation name x, the class D of explicitely foreseen input object behaviors for the input parameter, and the class R of the foreseen results. For a given functional signature, we define the class of offered behaviors that **satisfy the functional signature** to be the class **FunSig**(x<D, R>) = {B I B <c Range(x, R) and B >d Domain(x,D)}, which is the intersection of the above two classes.

Note: The class of mathematical functions, named x, from domain D to range R is a subset of **FunSig**(x<D,R>); namely those elements of this set which are determinate for all elements of the domain. The concept of "partial functions", which have no defined result for certain elements of the domain, may be interpreted in several different manners. Using the "undefined be default" approach, we could say that for the undefined cases all results would be possible. The approach "blocking by default" would imply that the function blocks for those cases. In other occasions, a special constant written Ö may be introduced in order to represent an "undefined" result.

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

Class 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 class of behaviors that **satisfy the signature**, written "**Sig(s)**", to be the intersection of all the FunSig(x<D,R>) where x<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 class of behaviors which is a subclass of the original one. This can be stated as follows.

Proposition (object-oriented subtyping): (Note: This shows a relation between signatures and corresponding classes of behaviors.) If the object signature s' is obtained from a signature s by the addition of the functional signature x<D,R>, then **Sig**(s') is subclass of **Sig**(s).

3.2. Comparison of Classes

As discussed in 2, offered behaviors may be compared by relations such as constrainment (corresponding to the notion of subset of possible actions) and domain coverage. These concepts may also be used to compare classes of offered behaviors. In addition, we may compare classes in terms of the set of behaviors they contain (as in the theorems above). We call this comparison "subtyping". Based on this latter view, we may also construct new classes, using such operators as **choice** (union) among several disjoint classes, and **multiple inheritance** (intersection of behaviors) of several given classes.

Definition (subclass): Given two classes (i.e. sets) B and B' of behaviors, we say that B' is a subclass of B iff B' is subset of B .

Definition (constraint relation): Given two classes B and B' of offered behaviors, we say that B' is **constrained** by B (written B' <c B) iff for each B' [B' there exists B [B such that B' <c B.

Note: It is clear that subtying implies constrainment.

Definition (domain coverage): Given two classes B and B' of offered behaviors, we say that B' covers the domain of B (written B' >d B) iff for each B' [B' there exists B [B such that B' >d B .

Note: If a class of behaviors B' covers the domain of a class B then an element of B' may only block for a given operation and input parameter if there is an element of B that may block for the same situation.

Examples: We call **Arbitrary** the class which contains one behavior, namely Arbitrary. We call **Stop** the class which constains one behavior, namely Stop. We call **Empty** the class containing no behavior. We note that all classes of behaviors constrain the class **Empty**, and are constrained by the class **Arbitrary**. And the class **Stop** is constrained by all classes, except by Empty. We also note that **Arbitrary** and **Empty** cover the domain of all classes. Another example is the class Chaos which contains all subsets of the behavior Arbitrary. We note that all classes are constraint by **Chaos** and cover the domain of **Chaos**. An overview of these relationships is given in Fig. 2.

4. Object behavior with state changes

4.1. Deterministic behavior

Definition: We say that an object behavior is **deterministic** if after each trace t of executed actions, the object is in a determined state with offered behavior B. (Note: This does not preclude a nondeterminate result).

Notation: We define a deterministic object behavior in terms of a set D of tuples <t,B> where t is a trace of actions and B is the offered behavior of the object after the execution of the trace t. It is noted that the set D of tuples must satisfy the following **consistency condition**: If <t,B> is included in D and x<i:o> is included in B, then there is a <t',B'> included in D where t' is equal to t concatenated with the action x<i:o>. Therefore, the "B" in the tuplets of D are redundant, and it is sufficient to consider only the set of possible traces.

Definition: Given two object behaviors D and D', we say that D' is constrained by D (written D' <c D) iff for each <t,B'> included in D' there is a <t,B> included in D with B' <c B.

Definition: Given two object behaviors D and D', we say that D' **covers the domain** of D (written D'>d D) iff for any **trace** t,<t,B> included in D and <t,B'> included in D' implies B'>d B.

Note: The statement saying that D' is **constrained** by D is equivalent to saying that the traces of D' are a subset of the traces of D. However, the statement saying that D' covers the domain of D does not imply that the traces of D' include those of D, since the traces of D' may have different results as output compared to those of D.

4.2. Non-deterministic behaviors

Definition: An object behavior is non-deterministic if there exists a trace t such that after the execution of the actions of t, the object may be in one of several states, each characterized by a different offered behavior.

Notation: We define a non-deterministic behavior in terms of a set N of tuplets <t,B> where t is a trace of actions and B is the set of potential offered behaviors that may apply after the execution of the trace t.

We can adapt the relations for comparing behaviors described above as follows to the case of non-deterministic object behaviors.

Definition: Given two object behaviors N and N', we say that N' is constrained by N (written N' <c N) iff for each <t,P'> included in N' there is a <t,P> included in N with P' <c P, where the latter relation is the same as defined for classes of offered behaviors in Section 3.2.

Definition: Given two object behaviors N and N', we say that N' covers the domain of N (written N' >d N) iff for any trace t, <t,P> included in N and <t,P'> included in N' implies P' >d P, where the latter relation is the same as defined for classes of behaviors in Sections 3.2.

Note: If we ignore the input parameters and results of operations then the statement saying that N' covers the domain of N is equivalent to saying that N' conforms to N, as defined in [Brin 86].

Note: For the classes of constant behavior Chaos, Arbitrary, Stop and Empty, we have corresponding temporal object behaviors with the same names. We note that the behaviors Stop and Empty allow only for the empty trace (no possible action). The temporal behaviors Arbitrary and Stop are deterministic (for instance Arbitrary admits, after any trace of executed actions, the set of all actions), while the temporal behavior Chaos is non-deterministic and may, after any sequence of executed actions, block or offer arbitrary sets of actions (see also [Hoar 85]).

Note: The definitions for constrainment and domain coverage given in the previous sections of this paper are special cases of the definition given here for non-deterministic objects.

Definition (reduction): Given two object behaviors N and N', we say that N' is a reduction of N (written N' red N) iff N' is constrained by N and N' covers the domain of N.

Note: If we ignore the input parameters and results of operations, then the above definition of "reduction" is the same as the one given in [Brin 86].

5. Specifications and Conformance Relations

5.1. Specification of requirements

In the context of temporal logic, it is customary to distinguish between safeness and liveness.[Alpe 87] Safeness states that "nothing bad can happen", while liveness states that "certain good things will eventually happen". Using the constrainment and domain coverage relations introduced above, we can state the following requirements for the behavior IA of an implementation of a module A :

(a) IA shall be constrained by a given behavior Bc.

(b) IA shall cover the domain of a given behavior Bd.

In many situations, the two behaviors Bc and Bd may be the same. In general, however, these two behaviors may be distinct. The distinction between these two aspects (constainment and domain coverage) is related to the predicates MAY and MUST described by Larson which are used to distinguish between transitions that may be implemented and those that must be implemented in a given transition system. There is also much similarity between constrainment and safeness. However, in the case of non-deterministic behaviors, the constrainment relation is finer than trace semantics [Boch 91c] while the latter corresponds to safeness. On the other hand, the domain coverage requirement has some similarity with liveness, in the sense that it states the absense of certain deadlocks. We therefore adopt the following definition for the specification of a module.

Definition (specification): A specification for an object A is a pair (Bc,Bd) of two behaviors. We say that an implementation IA is **conform** to the specification (Bc,Bd) iff the **behavior** B of the implementation satisfies the following two conditions:

(a) Constrainment condition: B <c Bc

(b) Domain coverage condition: B >d Bd

Note: In the case that the two specified behaviors are equal (Bc = Bd = B) then an implementation conforms to the specification iff its behavior is a reduction of B.

In the context of strongly typed programming languages, the signature of a function indicates the domain of the input parameters for which the function must be defined and the range of the 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 often not completely checked). Therefore, the type declaration of a function is a specification, where the given behaviors Bc and Bd are the behaviors Range and Domain, respectively, as introduced in Section 3. The type checking of object-oriented programs [Blac 87] follows the same principles.

5.2. Conformance relation between specifications

The idea of conformance is the following [Blac 87] [Cusa 89] : We assume that a system is designed as a composition of several "modules", and that the specification of module A is SA. We say that another module specification SA' **conforms** to SA in the context of this system iff an implementation conforming to SA' may be used for the module A without affecting the behavior of the overall system. In general, this conformance relation is dependent on the environment in which the module is supposed to operate [Gotz 91]. In the following we ignore this dependency and consider only the case of an environment with an arbitrary behavior.

Definition (conformance between specifications): Given two specifications (Bc, Bd) and

(Bc', Bd'), we say that (Bc', Bd') **conforms** to (Bc, Bd) iff Bc' <c Bc and Bd' >d Bd.

Proposition: If (Bc', Bd') conforms to (Bc, Bd) then any behavior B conforming to (Bc', Bd') also conforms to (Bc, Bd).

5.3. Multiple inheritance

In the design of complex systems, it may occur that one identifies a module A which should satisfy several requirements, for instance the ideal professor in a university should be a good teacher, a good researcher and a good public relations manager. These different requirements may be specified separately in terms of different specifications. The specification of module A is then the "conjunction" of the three requirements. Sometimes the different requirements may be contradictory. In that case no behavior exists that satisfies them all.

At the level of object signatures, the issues of multiple inheritance (also called subtyping relations) have been discussed in the literature on object-oriented programming languages. However, these issues are much less explored at the level of temporal behavior. Ignoring the requirements on domain coverage, it can be shown that, in the context of non-deterministic transition systems, the most general behavior satisfying the constrainment requirements of several specifications can be obtained by parallel composition of the behaviors Bs of every specification.

6. Conclusions

A unified approach can be used for the comparison of object behaviors from the context of predefined constants, such as Integers or Booleans, through user-defined functions and object signatures, to the consideration of temporal behaviors which may involve state dependencies and non-determinism. The approach considers two relations for the comparison of object behaviors and classes: (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 unified approach provides a framework for the definition of requirement specifications, the conformance between implementations and specifications, as well as the comparison between specifications which is important for managing the multiple inheritance subtyping lattice of object-oriented specifications and for questions of replacement and reutilisation.

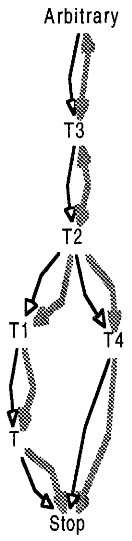

Fig. 1: Relationship between the behaviors of "True" constants of varrious types.

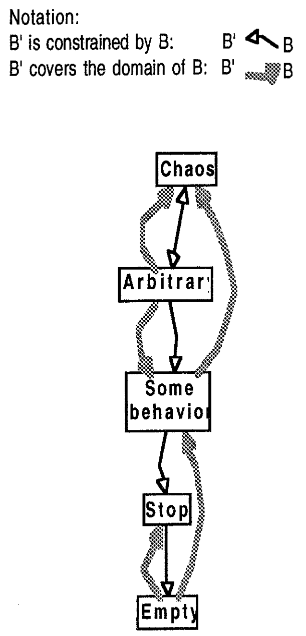

Fig. 2: Relationship between various classes of contant behaviors.

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.

[**Card 88**] 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.

[**Gotz 91**] R. Gotzhein, On Conformance in the Context of Open Systems, 12th Int. Conf. on Dist. Computing Systems, Japan June 92.

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

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

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