Consistency of the Static and Dynamic Components of Object-Oriented Specifications

Anna Formica
IASI-CNR, Rome, Italy
formica@iasi.rm.cnr.it

Heinz Frank
University of Klagenfurt, Klagenfurt, Austria
heinz@ifi.uni-klu.ac.at

Abstract

Object-Oriented modeling and design methodologies have been receiving a significant attention since they allow a quick and easy-to-gasp overview about a complex model. However, in the literature there are no formal frameworks that allow designers to verify the consistency (absence of contradictions) of both the static and dynamic components of the specified models, that are often assumed to be consistent. In this paper, a unifying formal framework is proposed that allows the consistency checking of both the static and dynamic components of a simplified Object-Oriented model.

Keywords: Object-Oriented specifications, consistency, integrity constraints, static model, dynamic model.

1 Introduction

"Object-Oriented modeling and design is a new way of thinking about problems using models organized around real-world concepts" [28]. Nowadays, various Object-Oriented (OO) modeling and design methodologies have been consolidating, such as, for instance OMT [28], OOD [5], OOSE [25] and, in particular, UML [31], that is very popular at the moment.

These methodologies have been receiving most of the attention since they offer significant modeling facilities by using diagrammatic notations. The use of diagrams, that can be rooted in the early beginning of conceptual modeling (the

ER model [12]), is becoming a consolidated methodology since it allows a quick and easy-to-gasp overview about a complex model. UML, for instance, offers a set of nine different diagram types which allow one to view a system from different perspectives.

However, visual modeling languages lack the capability of expressing finer details about the system, therefore they have been enriched with textual languages [32]. In UML, for instance, the Object Constraint Language (OCL) has been defined, that is a textual language for describing constraints within OO models. It has been specifically designed to complete diagrams with formal statements concerning restrictions on the values allowed for the object instances of the specified model.

When using an OO modeling methodology two main dimensions have to be considered: the $structure$ of the objects, that is represented by the $static$ model ($object$ model or $class$ model), and the $behavior$ of the objects, that is represented by the $dynamic$ model (or $behavioral$ model).

In the past, research was mainly concentrated on understanding the static model of the objects, that is, the attributes, relationships, and integrity constraints the objects have to satisfy, whereas the object behavior was either ignored or supposed to be defined by the signature of methods only (see for instance [2, 9, 10, 16, 26]). Languages similar to OCL were used to express conditions (integrity constraints) on the static model (as for instance, "nobody must earn more than his/her boss"). Successively, such languages were used to express conditions on the behavioral model too (as for instance, "in order to transfer more than a certain amount of money a further signature is required"). Currently, $statecharts$, introduced by David Harel [23], or variants of statecharts [1], are often used to express the object behavior by using constraint languages similar to OCL (see for instance [18]). However, nowadays that behavioral models are spreading, there is no significant work in the literature about formal frameworks that allow, for instance, the consistency (i.e., absence of contradictions) of the specified models to be checked. In fact, very often such models are simply assumed to be consistent (see for instance [18, 22]).

In this work, both dimensions of OO conceptual modeling are considered. In particular, in the paper a unifying formal framework has been defined that allows the consistency checking of both the static and dynamic components of a simplified OO model. The contribution of the paper consists in: (i) the definition of the semantics of the dynamic component of an Object-Oriented specification in terms of the semantics of the associated static component, (ii) the notion of *consistent behavior* of an OO specification, and (iii) the formal characterization of it. Notice that, in OO modeling, object behavior is distinguished in *intra-object* and *inter-object* behaviors, i.e., the behavior of objects that are instances of the same type,

and the behavior of objects instances of different types, respectively. In this paper we focus on intra-object behavior.

The paper is organized as follows. In Section 2, the static and dynamic models adopted in this paper are formally introduced. In Section 3, the formal characterization (i.e., the necessary and sufficient conditions) of the consistency of the static and dynamic components of an OO specification is presented. Finally, the conclusion and future work follow. Below the related work is given.

1.1 Related work

In the literature, static and behavioral models are generally assumed to be consistent. However, it is interesting to briefly recall some of the existing proposals concerning the integrity constraint satisfiability (consistency) checking, as investigated within the fields of databases and logic programming.

In [2, 7, 13, 16] we find various methodologies for the verification of the consistency of static data models, including ISA hierarchies, or disjointness constraints, or cardinality constraints. However, constraints involving comparison operators are not addressed that, vice versa, are on the basis of the dynamic model proposed in this paper.

The satisfiability of integrity constraints involving comparison operators has been addressed, for instance, in [3, 17, 15], within the field of Object-Oriented databases (OODB), and widely investigated in the context of deductive databases [6], by relying on theorem prover techniques. However, since the formalism adopted in [3] is very expressive (it includes union, complement, quantified sets, valueset-types etc..), the consistency checking of recursive schemas enriched with explicit integrity constraints is undecidable. Similarly, in the context of deductive databases, and also in [17], since a schema is a set of first order logic formulas, the methods proposed by the authors are semidecidable. Finally, in [15], a characterization of finite satisfiability of OODB integrity constraints involving comparison operators has been addressed. However, in that paper a different class of constraint expressions has been considered that, for instance, does not allow comparisons with constants to be expressed.

Leibniz is a system for logic programming, based on *logic decomposition* techniques [30]. It compiles fast solution algorithms for checking the satisfiability of a given set of boolean formulas in conjunctive normal form, in which the variables range on predefined, finite domains. Indeed, in order to adopt this method in our data model, a preliminary step should be performed since we do not assume finite domains, i.e., we do not require objects to take values on finite ranges of values.

Finally, in the literature, many OO specification approaches have been proposed, related to system functions, behavior, communication, and decomposition (the reader that is interested in a comparison among the different proposals may

refer to [33]). In this paper, regarding the static model, we followed essentially the UML approach. In particular, the static model has been formalized by using an OO specification language based on the notion of a type, that can be seen as a textual form of the UML class diagrams. Furthermore, the constraint expression language adopted in this paper is similar to OCL, that is the textual language used to describe constraints in UML [32]. Regarding the behavioral model, we followed the statecharts approach proposed by Harel [22, 23, 24], that is recalled in Subsection 2.2.

2 The static and dynamic models

2.1 The static model

In this subsection, a specification language of a simplified OO model is presented. The language, that is based on the notion of a $type$, is compliant with the $ODMG$ standard [10], and has a kernel common to the type-expression specification language O2 [4].

A type has a $name$, a $tuple$ and a $constraint\ expression$. The tuple is given by a set of *typed properties* (*tp*) enclosed in square brackets. A constraint expression (*c_expr*) is a disjunction of conjunctions ($disjunctive\ normal\ form$) of expressions of the form: "$p\ \theta\ K$" ($single\ expression$, indicated as s_expr for short), where p is the $name$ of a property, θ stands for a comparison operator, such as "=", "$\geq$, ">", $\neq$, etc., and K is a constant. For instance, consider the example below where two types are defined, whose $names$ are vehicle, and employee, respectively. In particular, one c_expr is given, associated with employee, establishing a lower bound for the consultant ($consult$) and manager (mgr) salaries.

Example 2.1
vehicle $\dot{\preceq}$ [maker:string, owner:employee, color:(red, green, blue),
production_date:integer]
employee $\dot{\preceq}$ [name:string, salary:integer, drives:vehicle, status:(depend, consult, mgr),
boss:employee],
(salary $>$ 2000 $\wedge$ status = consult) $\vee$
(salary $>$ 4000 $\wedge$ status = mgr)

□

A property, that is identified by a $name$, can be typed by using: (i) type $names$ (as, in Example 2.1, vehicle.owner), establishing an explicit link (or association) between two types; (ii) $atomic$-$types$, e.g., *integer* or *string* (for instance, in the example, vehicle.maker); (iii) $valueset$-$types$, that are specified between ordinary parenthesis by the interval extremes (in the case of integer or real intervals), or by

enumeration (as, for instance, vehicle.color). In a tuple, multiple occurrences of the same property $names$ are not allowed. Furthermore, we assume that properties are single-valued, that is, an object, instance of a type, has to take exactly one value in correspondence with each property.

Notice that, in this paper, inheritance is not addressed. In particular, types are supposed to have both the static (typed properties and constraint expressions) and behavioral (that will be addressed in the next subsections) components explicitly given.

A c_expr is *well-formed w.r.t.* (a type whose $name$ is) τ if all the properties defining the constraint expression are properties of τ. For instance, the c_expr of the previous example is well-formed w.r.t. employee, whereas it is not well-formed w.r.t. vehicle. Below, the notion of an OO $specification$ is introduced.

Definition 2.1 [OO specification] A finite set of types is an OO *specification* iff:

- every type $name$ is $uniquely$ defined (i.e., the same $name$ is not associated with more than one tuple);
- there are no dangling type $names$ (i.e., every type $name$ is defined);
- every c_expr associated with a type τ is well-formed w.r.t. τ.

□

The set of types given in Example 2.1 is an OO specification. Notice that in a specification, besides the above requirements, the constraint expressions are supposed to be correctly typed, e.g., in Example 2.1, the constraint salary $>$ red associated with employee would be rejected at a pre-processing stage, by using a type-checker.

As already mentioned, in our approach the c_expr's are in $disjunctive\ normal\ form$, i.e., they are disjunctions of conjunctions of s_expr's. Of course, by applying the standard replacement rules for logical operators [21], any expression that is a conjunction of c_expr's, or its negation, will be considered a c_expr as well.

2.1.1 Semantics of an OO specification

The formal semantics of an OO specification will be given according to the formal semantics of Description Logics, as defined in [8].

Given an OO specification $\mathcal{S}$, let $\mathcal{T}$ be the set of type $names$, $atomic$-$types$, and $valueset$-$types$ of $\mathcal{S}$ (corresponding to the $atomic\ concepts$ in [8]), and let $\mathcal{P}$ be the set of property $names$ of $\mathcal{S}$ (corresponding to the $atomic\ roles$ in [8]).

An $interpretation$ $\mathcal{I} = (\Delta^{\mathcal{I}}, .^{\mathcal{I}})$ over $\mathcal{S}$ consists of a non-empty and finite set $\Delta^{\mathcal{I}}$, that is the $domain$ of $\mathcal{I}$, and a function $.^{\mathcal{I}}$, that is the $interpretation$ $function$ of $\mathcal{I}$, that maps every type $\tau \in \mathcal{T}$ to a subset $(\tau)^{\mathcal{I}}$ of $\Delta^{\mathcal{I}}$ (the set of $instances$ of τ) and every property $p \in \mathcal{P}$ to a subset $(p)^{\mathcal{I}}$ of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. Then, a type:

$\tau \dot{\preceq}$ *tuple*, *c_expr*

is an $inclusion\ assertion$ as defined in [8] (i.e., it specifies only necessary conditions for an object to be an instance of the type τ), for which the interpretation function is defined as follows ($\#S$ denotes the cardinality of the set S and a_expr denotes an $and\ expression$, i.e., a conjunction of s_expr's):

- $(tuple,\ c_expr)^{\mathcal{I}} = (tuple)^{\mathcal{I}} \cap (c_expr)^{\mathcal{I}}$
- $(tuple)^{\mathcal{I}} = \bigcap_j\ ([p_j : type_j])^{\mathcal{I}} =$
  $\bigcap_j\ (\{x \in \Delta^{\mathcal{I}} \mid \#\ \{y\text{: } < x, y > \in (p_j)^{\mathcal{I}} \text{ and } y \in (type_j)^{\mathcal{I}}\} = 1\})$
  that corresponds to the Description Logics construct $(\exists^{=1} p_j.type_j)^{\mathcal{I}}$
- $(c_expr)^{\mathcal{I}} = (a_expr_i \vee a_expr_j)^{\mathcal{I}} = (a_expr_i)^{\mathcal{I}} \cup (a_expr_j)^{\mathcal{I}}$
- $(a_expr)^{\mathcal{I}} = (s_expr_i \wedge s_expr_j)^{\mathcal{I}} = (s_expr_i)^{\mathcal{I}} \cap (s_expr_j)^{\mathcal{I}}$
- $(s_expr)^{\mathcal{I}} = (p\ \theta\ K)^{\mathcal{I}} =$
  $\{x \in \Delta^{\mathcal{I}} \mid \forall\ y\text{: } < x, y > \in (p)^{\mathcal{I}} \Rightarrow y\ \theta\ K\}$
  where θ is one of the comparison operators "$\geq$", "$>$", "$=$", "$\neq$", etc..

A *model* of $\mathcal{S}$ is an interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, .^{\mathcal{I}})$ that $satisfies$ all the inclusion assertions in $\mathcal{S}$, i.e., for any inclusion assertion defined as above, the following holds:

$(\tau)^{\mathcal{I}} \subseteq (tuple,\ c_expr)^{\mathcal{I}}$.

A type $\tau \in \mathcal{T}$ is *consistent* ($satisfiable$ according to [8]) in $\mathcal{S}$ if $\mathcal{S}$ admits a model for which $(\tau)^{\mathcal{I}} \neq \emptyset$.

Finally, $\mathcal{S}$ is $consistent$ if it admits a model for which each type is consistent in $\mathcal{S}$.

In the following, a few definitions involving constraint expressions that do not necessarily belong to the given specification are presented.

Consider a consistent specification $\mathcal{S}$, an inclusion assertion of $\mathcal{S}$ as defined above, and a constraint expression, say c_expr_i, that is well-formed w.r.t. τ. Then c_expr_i is $consistent\ w.r.t.\ \tau$ iff there exists at least one model $\mathcal{I}$ of $\mathcal{S}$ for which:

$(\tau)^{\mathcal{I}} \cap (c_expr_i)^{\mathcal{I}} \neq \emptyset$.

Under the same assumptions above, consider the constraint expressions c_expr_k and c_expr_h well-formed w.r.t. τ. Then c_expr_k and c_expr_h are:

- *equivalent w.r.t. τ tuple* iff for any model $\mathcal{I}$ of $\mathcal{S}$ the following holds:
  $(c_expr_k)^{\mathcal{I}} \cap (tuple)^{\mathcal{I}} = (c_expr_h)^{\mathcal{I}} \cap (tuple)^{\mathcal{I}}$
  where c_expr_k and c_expr_h are also supposed to be consistent w.r.t. τ;

- *disjoint w.r.t. τ tuple* iff for any model $\mathcal{I}$ of $\mathcal{S}$ the following holds:
  $(c_expr_k)^{\mathcal{I}} \cap (c_expr_h)^{\mathcal{I}} \cap (tuple)^{\mathcal{I}} = \emptyset$.

2.2 The dynamic model

The behavior of a type is defined by a *statechart* [22, 23, 24]. A statechart is associated with a type and consists of *states*, *events* and *transitions*. A state is composed of a *name* (which identifies it) and a condition that the objects, instances of the associated type, have to satisfy to be in that state [31]. This condition is referred to as the *range* of the state. A *transition* is a relationship among states and is triggered by an event. A transition, which is identified by a *label*, indicates that an object, which is in a state (called *source state*) will enter another state (called *target state*) when the event occurs and some specified condition (called the *guard* of the transition) holds [31]. Therefore, at the end of the transition the object will be in the target state of the transition. An *event* is identified by a *name* and may trigger one or more transitions.

Example 2.2 Consider a type book defined as follows:

book $\dot{\preceq}$ [isbn:string, title:string, signed:bool, age:integer, registered:bool,
reserved:bool, archived:bool, status:(new, preparation, in library,
borrowed, in text book collection, in archive)]

and suppose that books which are in the library can be borrowed, if they are not reserved. This simplified behavior for the type book is represented in Figure 1 by means of states, events and transitions. In particular, in Figure 1 two states are represented, whose *names* are book on stock and book on loan, respectively. The ranges of these states are described in Table 1, by using the syntax presented in the previous subsection. Furthermore, in Figure 1, t3 is the label of a transition between the source state book on stock and the target state book on loan, that is triggered by the event lending. The guard of the transition is reserved = false.

book on stock	status = in library
book on loan	status = borrowed

Table 1: Ranges of the states of Figure 1

□

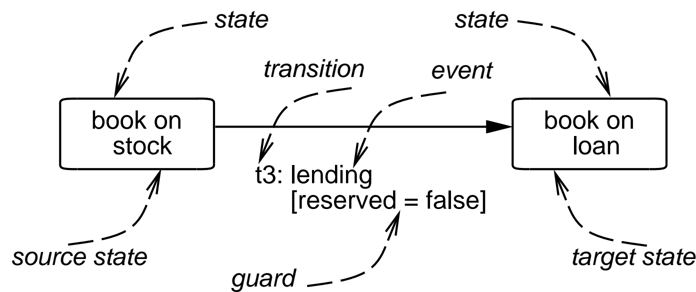

Figure 1: A simplified behavior associated with the book type

A more elaborated behavior associated with the type book is shown below, within a more complex example concerning a university library.

The Library

Consider the domain of a university library and the type book defined above. The behavior of book objects is now shown in Figure 2. Two main activities are necessary for the administration of new books before they can be placed into the library: the book registration and the book processing. The book registration is responsible for recording new books. For this purpose, all the information related to the book is stored in the book catalogue and, successively, a registration number is given to the book. In the book processing state, the book is described with some keywords and, then, a signature is added.

After the administration process, books are placed into the library, where they can be borrowed. Books, which are necessary for a lecture are given in a special place called text book collection. Nobody is allowed to borrow books from the text book collection. If a book has to be placed into the text book collection but, at that moment, is borrowed by anyone else, it can be reserved. Reserved books cannot be borrowed by anyone but are placed into the text book collection immediately after they are returned by the borrower. Books which are borrowed or in the text book collection, and are not reserved, can be returned to the library. Borrowed books, that are reserved, can be returned to the text book collection only. Books, older than 10 years can be given into the archive. Such books can be borrowed too, but they can be returned to the archive only.

In the next sections this example will be used to better clarify our proposal. For a deeper understanding about extensions of the statechart language we refer the reader to [18].

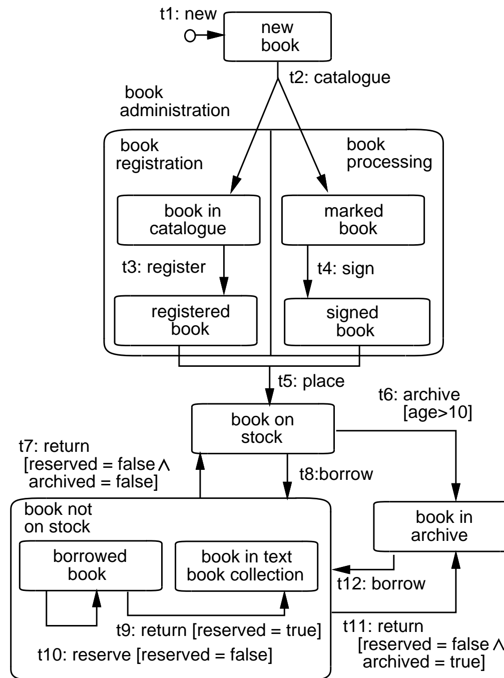

Figure 2: A more elaborated behavior associated with the book type

2.2.1 States

A state in a statechart associated with a type τ denotes a subset of all possible object instances of τ. A state can be seen, at an intensional level, as a predicate that is associated with a given type, whereas, at an extensional level, it can be considered as the set of all possible objects which fulfill such a predicate.

To make statecharts more readable and to avoid combinatorial explosion of nodes and arcs, state hierarchies have been introduced. According to Harel's definition [24], we distinguish between *OR-states*, *AND-states* and *basic states*. OR-states have substates which are related to each other by "exclusive-or", i.e., an object can be in only one substate of an OR-state at any time. AND-states have substates which are "and" related, i.e., an object that is in an AND-state is also in all substates of the AND-state. In AND-states, it is worth distinguishing *component states* from *computational substates*. The former are syntactical substates, as for instance, in the case of book administration of Figure 2, book registration and book processing, whereas the latter are semantical substates, as for instance, again in the case of book administration, book in catalogue and signed book. In other words, computational substates correspond to the legal combinations of substates of component states.

Basic states are the states at the bottom of a state hierarchy, i.e., they do not

have substates. The states at the highest level of a statechart, i.e., without parent states, are called *root states*.

Example 2.3 In Figure 2 book administration is an AND-state with the substates book registration and book processing. These two substates, together with the book not on stock state, are OR-states. All the other states are basic states. Root states are new book, book administration, book on stock, book not on stock and book in archive. □

The ranges of the basic states have to be given by the designer, whereas the ranges of the structured states (AND-states and OR-states) are defined according to the ranges of their substates as follows.

Definition 2.2 [**The Range function**] The $Range$ function is inductively defined on the set of states of a statechart as follows:

- each basic state is associated with a c_expr called its $range$;
- each OR-state S is associated with the disjunction of the $ranges$ of all component states of S;
- each AND-state S is associated with the conjunction of the $ranges$ of all component states of S.

□

Example 2.4 The ranges of the basic states of the example of Figure 2 are shown in Table 2. According to the above definition, the range of the OR-state book not on stock is defined as the disjunction of the ranges of its substates borrowed book and book in text book collection, i.e.:
status = borrowed $\vee$ (status = in text book collection $\wedge$ reserved = false).
The range of the AND-state book administration is defined as the conjunction of the ranges of the OR-states book registration and book processing, each obtained as disjunction of the ranges of its component states. □

In the following, the range of a state S will be indicated as $S.Range$.

2.2.2 Events and transitions

An event is an incident whose goal is the modification of the state of an object. An event, which is identified by a *name*, is set off explicitly, and triggers one or more transitions. A transition, which is identified by a *label*, indicates that an object, that is in a given state (called *source state*), will enter another state (called *target*

new book	status = new
book in catalogue	status = preparation $\wedge$ registered = false
marked book	status = preparation $\wedge$ signed = false
registered book	status = preparation $\wedge$ registered = true
signed book	status = preparation $\wedge$ signed = true
book on stock	status = in library
borrowed book	status = borrowed
book in text book collection	status = in text book collection $\wedge$ reserved = false
book in archive	status = in archive $\wedge$ age > 10

Table 2: Ranges of the basic states of Figure 2

state) when the event occurs and some specified condition (called the *guard* of transition) holds.

In the following, similarly to the notation used for the ranges of states, $t.Guard$ will denote the guard of transition t.

Example 2.5 In the example of Figure 2, t9 is the label of a transition between the source state borrowed book and the target state book in text book collection, triggered by the event return. The guard of the transition is reserved = true. The event return also triggers transitions t7 and t11. □

Notice that transitions may have computational substates as source (or target) states. For instance, the combination of registered book and signed book states is the source state of transition t5.

In dynamic modeling, events and transitions represent (partial) specifications of the methods associated with the types. The model defines which conditions ($preconditions$) an object has to fulfill in order to be able to react to an event, and which conditions ($postconditions$) an object satisfies after the state change. If an event is set off and the preconditions hold, an object is transferred to a new state. Notice that the preconditions of a transition can be derived from the defined model, whereas the postconditions are suitably defined by the designer. In particular, a state change can be performed if the object is in the source states of the transition (that means it satisfies the ranges of the source states) and, furthermore, it satisfies the guard of the transition. In the following let $t.Target_States$ and $t.Source_States$ be the sets of the target and source states of transition t, respectively (for instance, t5.Source_States contains the states registered book and signed book). Then, the preconditions of a transition are defined as follows.

Definition 2.3 [**Preconditions of a transition**] The preconditions of a transition t, indicated as $t.PreC$, are defined as:

$t.PreC = t.Guard \wedge s_1.Range \wedge \ \ldots \ \wedge s_n.Range$
where $s_i \in t.Source_States$, $i = 1 \ldots n$. □

After a transition has been applied, the object has to satisfy the ranges of its target states and its postconditions, which must explicitly be given by the designer. Therefore, the $postconditions$ of a transition t, indicated as $t.PostC$, *must imply the ranges of its target states*.

Similarly to ranges of states, c_expr's will be used to specify guards, pre- and postconditions of transitions. In Table 3, the postconditions of the transitions of the example of Figure 2 are shown.

t1: new	status = new
t2: catalogue	status = preparation ∧ registered = false ∧ signed = false
t3: register	status = preparation ∧ registered = true
t4: sign	status = preparation ∧ signed = true
t5: place	status = in library ∧ reserved = false
t6: archive	status = in archive ∧ age > 10
t7: return	status = in library ∧ reserved = false ∧ archived = false
t8: borrow	status = borrowed ∧ reserved = false ∧ archived = false ∨ status = in text book collection ∧ reserved = false ∧ archived = false
t9: return	status = in text book collection ∧ reserved = false
t10: reserve	status = borrowed ∧ reserved = true
t11: return	status = in archive ∧ age > 10 ∧ reserved = false ∧ archived = true
t12: borrow	status = borrowed ∧ reserved = false ∧ archived = true ∨ status = in text book collection ∧ reserved = false ∧ archived = true

Table 3: Postconditions of the transitions of Figure 2

Example 2.6 In the example of Figure 2, the preconditions of transition t9 are given by the conjunction of the range of the state borrowed book and its guard reserved = true, i.e.:

status = borrowed ∧ reserved = true.

The preconditions of transition t5 correspond to the conjunction of the ranges of the states registered book and signed book. The postconditions of t5, that are:

status = in library ∧ reserved = false,

imply the range of the book on stock state. □

2.2.3 The behavior of a type

The behavior of a type is defined as follows.

Definition 2.4 [**Behavior of a type**] The behavior of a type τ, indicated as B_τ, is a statechart whose ranges of the basic states, postconditions and guards of transitions are c_expr's well-formed w.r.t. τ. □

Based upon the ranges of the states we can define relationships between states, namely *equivalent* states, *disjoint* states and *orthogonal* states. These relationships will allow us to define the notion of consistent behavior of an OO specification.

Definition 2.5 [**Equivalent states**] The states S_1 and S_2 of the behavior B_τ of the type τ are *equivalent* iff their ranges are equivalent c_expr's w.r.t. τ tuple. □

Example 2.7 In the example of Figure 2 the states book registration and book processing are equivalent since their ranges coincide. In fact, each of these states is an OR-state, whose range is given by the disjunction of the ranges of its substates, that is, status = preparation (see Table 2). □

Definition 2.6 [**Disjoint states**] The states S_1 and S_2 of the behavior B_τ of the type τ are *disjoint* iff their ranges are disjoint c_expr's w.r.t. τ tuple. □

Example 2.8 In the example of Figure 2 the states new book and book on stock are disjoint states, since they require different values for the status attribute. □

If S is an OR-state or an AND-state, in the following $S.Substates$ denotes the set of the direct substates of S. In the example of Figure 2, book administration.Substates contains the states book registration and book processing.

Definition 2.7 [**Orthogonal states**] The states S_1 and S_2 of the behavior B_τ of the type τ are *orthogonal* iff S_1 and S_2 are OR-states which are equivalent, and $\forall s \in S_1.Substates$ and $\forall s' \in S_2.Substates$ there exists at least one model $\mathcal{I} = (\Delta^\mathcal{I}, .^\mathcal{I})$ s.t. both $s.Range$ and $s'.Range$ are consistent w.r.t. τ, i.e.:
$(s.Range)^\mathcal{I} \cap (s'.Range)^\mathcal{I} \cap (\tau)^\mathcal{I} \neq \emptyset$. □

Essentially, orthogonality means that for an object, which is in a substate s of an OR-state S_1, it is possible to be in anyone of the substates s' of S_2 at the same time. In other words, the ranges of the states s and s' have not to be contradictory. This allows the definition of parallelism by using AND-states [23].

Example 2.9 In the example of Figure 2 the states book registration and book processing are orthogonal states. In fact, they are OR-states and their ranges are equivalent (as shown above). Furthermore, the range of each of the substates of book registration has a non-empty intersection with each of the ranges of the substates of book processing (and vice versa). For example, the intersection between the ranges of the states marked book and registered book is not empty, since an object with the attribute values status = preparation, signed = false, and registered = true, satisfies the ranges of both these states (see Table 2). □

In line with Harel's definitions [23], we formally define the notion of $consistent$ behavior of a type.

Definition 2.8 [Consistent behavior of a type] Given a consistent OO specification $\mathcal{S}$ and a type τ of $\mathcal{S}$, the behavior B_τ of the type τ is *consistent* iff the following conditions hold:

1. for each state S of B_τ, there exists at least one model $\mathcal{I}$ of $\mathcal{S}$ s.t. $S.Range$ is consistent w.r.t. τ, i.e.:
   $(S.Range)^{\mathcal{I}} \cap (\tau)^{\mathcal{I}} \neq \emptyset$;

2. for each pair of root states S_1, S_2 of B_τ:
   S_1, S_2 are disjoint;

3. for each OR-state S of B_τ:
   for each pair of substates s_1, $s_2 \in S.Substates$:
   s_1, s_2 are disjoint;

4. for each AND-state S of B_τ:
   for each pair of substates s_1, $s_2 \in S.Substates$:
   s_1 and s_2 are orthogonal;

5. given a transition t of B_τ, assume that $t.Target_States = \{S_i\}$, $i = 1 \ldots n$. Then, for each transition t:

   (a) there exists at least one model $\mathcal{I}$ of $\mathcal{S}$ s.t. $t.PreC$ is consistent w.r.t. τ, i.e.:
   $(t.PreC)^{\mathcal{I}} \cap (\tau)^{\mathcal{I}} \neq \emptyset$,

   (b) there exists at least one model $\mathcal{I}$ of $\mathcal{S}$ s.t. all $S_i.Range$, $i = 1 \ldots n$, and $t.PostC$ are consistent w.r.t. τ:
   $\bigcap_i (S_i.Range)^{\mathcal{I}} \cap (t.PostC)^{\mathcal{I}} \cap (\tau)^{\mathcal{I}} \neq \emptyset$,

   (c) $t.PostC, not(S_1.Range \wedge ... \wedge S_n.Range)$ are c_expr's disjoint w.r.t. τ tuple.

□

Let us briefly illustrate these conditions. Condition 2.8(1) deals with the fact that any object that is in a given state must also satisfy the type τ and the constraints of τ.

Example 2.10 Suppose we add the constraint age $<$ 10 to the type book of Example 2.2. Then, the behavior of this type becomes inconsistent. In fact, no object could satisfy this constraint and the range of the state book in archive, i.e.:
status = in archive $\wedge$ age $>$ 10. □

According to Harel [23], an object cannot be in more than one root state at the same time. Therefore, all root states must be disjoint (condition 2.8(2)). OR-states have substates that are related to each other by exclusive-or, therefore the substates of an OR-state must be disjoint (condition 2.8(3)).

Condition 2.8(4) deals with AND-states. AND-states have substates which are and-related. Therefore, an object, that is in one direct substate of an AND-state must be in all (direct) substates of the AND-state.

Condition 2.8(5) concerns the pre- and postconditions of transitions. In particular, the condition 2.8(5a) requires the consistency of the preconditions of a transition with respect to the associated type. Conditions 2.8(5b,5c) are used to express that the postconditions of a transition have to imply the ranges of its target states.

Example 2.11 Suppose again that age $<$ 10 is a constraint associated with the type book. Then the preconditions of transition t6 are not consistent with the type book, due to the guard age $>$ 10. Suppose we now delete age $>$ 10 from the postconditions of transition t6 in Table 3. Then, both the conditions 2.8(5b,5c) are violated because the postconditions of t6 do not imply the range of the target state:

status = in archive $\wedge$ age $>$10. □

By generalizing Definition 2.4, since an OO specification is defined as a set of types, we can speak about the *behavior of an OO specification*. Therefore, we can introduce the notion of *consistent behavior of an OO specification* as follows.

Definition 2.9 [Consistent behavior of an OO specification] Given a consistent OO specification $\mathcal{S}$, the behavior of $\mathcal{S}$ is $consistent$ iff the behavior of each of the types of $\mathcal{S}$ is consistent. □

In the remaining of this paper, given an OO specification, the set of types (i.e., their structures and c_expr's) and the behavior of the specification will be referred to as the $static$ and $dynamic$ *components of the specification*, respectively.

3 Consistency of the static and dynamic components of an OO specification

In this section, a method for the consistency checking of both the static and dynamic components of an OO specification is presented. In particular, in the first subsections the definitions and the procedures on which the approach is based are presented and, successively, in the last subsection, the formal characterization of the proposed method is illustrated.

3.1 Formal definitions

Below the notion of *Interval* of an s_expr is presented. In particular, given an s_expr associated with a type τ, the $Interval$ of the s_expr is the set of admissible values that can be associated with an instance of the type τ through the property p defining the s_expr.

Definition 3.1 **[*Interval* of an s_expr]** Given a type τ, consider an s_expr well-formed w.r.t. τ, $s_expr = p\ \theta\ K$, and let σ be the type of the property p in τ. Then, the *Interval* (*Int*) of the s_expr is defined as follows:

- if σ is one of the types $integer$ or $real$, $Int(s_expr)$ is the interval of integers or reals, respectively, that is defined as:
  $Int(s_expr) = (-\infty, K)$ if θ is "<";
  $Int(s_expr) = (-\infty, K]$ if θ is "$\leq$";
  $Int(s_expr) = [K, K]$ if θ is "=";
  ....(and so on in all the other cases)
  where ordinary parentheses and square brackets denote open and closed intervals, respectively;

- if σ is the $string$ (*bool*) type, there are two possible cases. If θ is "=", $Int(s_expr)$ is the singleton containing the K (boolean) constant; otherwise, if θ is "$\neq$", $Int(s_expr)$ is the complement set of K in the set of all possible strings (the singleton containing the opposite boolean constant);

- if σ is a $valueset$-$type$, $Int(s_expr)$ behaves as in the previous cases, with the further intersection with the denoted set of values.

□

Example 3.1 Consider a type employee defined as follows (notice that the expressions are numbered only for better referencing them in the next examples):

employee $\dot{\preceq}$ [salary:integer, name:string, status:(depend, consult, mgr), boss:employee],
(salary $>$ 2000 $\wedge$ status $\neq$ mgr) (1) $\vee$
(salary $>$ 4000 $\wedge$ status $\neq$ depend $\wedge$ status $\neq$ consult) (2)

Then:
Int(status $\neq$ consult) = {depend, mgr}
and:
Int(salary $>$ 3000) = (3000, +∞). □

Definition 3.2 **[*Conflicting Expression* set]** Given a property p and an a_expr (a conjunction of s_expr's), the *Conflicting Expression* (*ConfEx*) set identifies all the s_expr's of the a_expr containing the p property:
$ConfEx(p, a_expr) = \{s_expr_h = p\ \theta\ K \mid a_expr = ...\ s_expr_h\ ...\}$ □

Example 3.2 Consider the property status of the employee type of Example 3.1 and the a_expr (2). Then their conflicting expression set is given by the following set of s_expr's:

$\{\mathsf{status} \neq \mathsf{depend}, \mathsf{status} \neq \mathsf{consult}\}$. □

Definition 3.3 [*Domain* set] Given a type τ, a property p of τ, and an a_expr well-formed w.r.t. τ, the *Domain* (*Dom*) set identifies the interval of admissible values for the property p according to the given a_expr, i.e.:

$$Dom(p, a_expr) = \bigcap_j \{Int(s_expr_j) \mid s_expr_j \in ConfEx(p, a_expr)\}$$ □

Example 3.3 Consider again the property status and the a_expr (2) of Example 3.1. Starting from the conflicting expression set seen above, since:

$Int(\mathsf{status} \neq \mathsf{depend}) = \{\mathsf{mgr}, \mathsf{consult}\}$
$Int(\mathsf{status} \neq \mathsf{consult}) = \{\mathsf{mgr}, \mathsf{depend}\}$

the Domain set is $\{\mathsf{mgr}\}$. □

3.2 The consistency checking procedures

In this subsection, the procedures on which the consistency checking method is based are presented. In the following, given a type τ, we assume that c_expr_τ stands for the constraint expression associated with the type τ.

Given a type τ and a c_expr which is well-formed w.r.t. τ, the $Consistent$ procedure presented below checks if the c_expr is consistent w.r.t. τ. We recall that this notion corresponds to the possibility of defining at least one object of type τ (therefore, also satisfying c_expr_τ, if present) satisfying the c_expr. In particular, suppose that:

$c_expr = a_expr_1 \ \vee \ldots \ \vee a_expr_n$
$c_expr_\tau = a_expr_{\tau,1} \ \vee \ldots \ \vee a_expr_{\tau,m}$

then, consistency holds iff at least one a_expr_i of c_expr and one $a_expr_{\tau,j}$ of c_expr_τ exist such that their conjunction is consistent w.r.t. τ. This is checked by using the $AndCheck$ procedure. Of course, if the type τ does not contain any constraint expression, only a_expr_i must be consistent w.r.t. τ.

Example 3.4 Consider the type employee of Example 3.1, and the following c_expr (that is indeed an a_expr):

salary $>$ 5000 $\wedge$ status = mgr.

In order to check if it is consistent w.r.t. employee, one of the following two a_expr's:

(salary $>$ 2000 $\wedge$ status $\neq$ mgr $\wedge$ salary $>$ 5000 $\wedge$ status = mgr)

(salary $>$ 4000 $\wedge$ status $\neq$ depend $\wedge$ status $\neq$ consult $\wedge$

Procedure 1 The $Consistent$ procedure

input: a type τ, possibly including $c_expr_\tau = a_expr_{\tau,1} \ \vee \ldots \ \vee a_expr_{\tau,m}$
and a c_expr well-formed w.r.t. τ:
$c_expr = a_expr_1 \ \vee \ldots \ \vee a_expr_n$
output: *true*, if the c_expr is consistent w.r.t. τ; *false*, otherwise

$Consistent(\tau, c_expr) \leftarrow false$
if $\exists$ a_expr_i, $a_expr_{\tau,j}$ s.t.:
$AndCheck(\tau, a_expr_i \ \wedge a_expr_{\tau,j})$ = true **then**
$Consistent(\tau, c_expr) \leftarrow true$
end if

salary $>$ 5000 $\wedge$ status = mgr)

must be consistent w.r.t. τ. This check is performed by the $AndCheck$ procedure informally illustrated below. □

Given a type τ and any a_expr well-formed w.r.t. τ, the $AndCheck$ procedure returns *true*, if the a_expr is consistent w.r.t. τ; *false*, otherwise. Essentially, the $AndCheck$ procedure reports consistency iff for each property p_i of the type τ occurring at least once in the a_expr (i.e., $|ConfEx(p_i, a_expr)| \geq 1$) the Domain set $Dom(p_i, a_expr)$ is non-empty. Otherwise the a_expr is not consistent w.r.t. τ since the property p_i cannot be instantiated.

Example 3.5 Consider Example 3.4. Only the second a_expr, say a_expr_2, is consistent w.r.t. τ, since:
Dom(salary, a_expr_2) = (5000, +∞),
Dom(status, a_expr_2) = {mgr}. □

Now, let us briefly illustrate the $Equivalent$ procedure. Such a procedure allows us to determine if two c_expr's well-formed w.r.t. τ are equivalent w.r.t. τ tuple. In particular, two constraint expressions c_expr_h and c_expr_k are equivalent iff for each $a_expr_i \in c_expr_h$ there exists an equivalent $a_expr_j \in c_expr_k$ and vice versa. The equivalence of the a_expr_i and a_expr_j is checked by the $EquiCheck$ procedure. In particular, a_expr_i and a_expr_j are equivalent iff, for all the properties p of τ, the Domain set on the pairs (p,a_expr_i) and (p,a_expr_j) is the same.

Example 3.6 Consider the type employee of Example 3.1 and the following two a_expr's:
a_expr_1 = status $\neq$ depend $\wedge$ status $\neq$ consult

and:

$a_expr_2 =$ status = mgr.

They are equivalent w.r.t. employee tuple because in correspondence with the status property, we have:

Dom(status, a_expr_1) = *Dom*(status, a_expr_2) = {mgr}

and for all the other properties of employee both the expressions have an empty Domain set. □

3.3 Formal characterization of consistent specifications

In this subsection, the consistency of both the static and dynamic components of an OO specification is formally characterized. Below, we start by focusing on the static component. The consistency of the static component is checked by applying the $Consistent$ procedure to all the types, and their associated constraint expressions, that are defined in the specification.

Theorem 3.1 [Characterization of the consistency of the static component of an OO specification] The static component of an OO specification $\mathcal{S}$ is consistent iff for each type τ of $\mathcal{S}$ containing constraint expressions:

$Consistent(\tau, c_expr_\tau)$ = true.

Proof. $\Rightarrow$ Trivial (by contradiction).

$\Leftarrow$ By construction. Suppose that for each type τ of $\mathcal{S}$, $Consistent(\tau, c_expr_\tau) = true$. Then for each τ there exists at least one a_expr of c_expr_τ, say $a_expr_{\tau,k}$, s.t., for each property p_i of τ, $Dom(p_i, a_expr_{\tau,k})$ is non-empty. Therefore, it is possible to define at least one model $\mathcal{I} = (\Delta^\mathcal{I}, .^\mathcal{I})$ s.t., for each τ, there exists an element $x_\tau \in (\tau)^\mathcal{I}$ defined as follows. For each property p_i of τ, let y_{p_i} be an element of $\Delta^\mathcal{I}$ s.t. $< x_\tau, y_{p_i} > \in (p_i)^\mathcal{I}$, and:

- if p_i is typed with an $atomic\text{-}type$ or a $valueset\text{-}type$ and occurs in $a_expr_{\tau,k}$, then $y_{p_i} \in Dom(p_i, a_expr_{\tau,k})$, that is non-empty (the case for which p_i does not occur in $a_expr_{\tau,k}$ is trivial);
- if p_i is typed with any type $name$, say γ, such that $(\gamma)^\mathcal{I}$ already contains one element (for instance, in the case $\gamma = \tau$), then y_{p_i} can be any element that is already present in $(\gamma)^\mathcal{I}$ (as for instance x_τ). Otherwise, the value y_{p_i} can be constructed by iterating the above steps.[1]

[1]Notice that it is not possible to get an infinite loop because constraint expressions on relationships (i.e., properties typed with type $names$) cannot be enforced. Therefore, in the construction process, recursive properties can always be instantiated with already defined elements of the interpretation domain.

□

The following lemma is a generalization of the previous theorem to the case of any c_expr well-formed w.r.t. τ.

Lemma 3.2 [Characterization of the consistency of any c_expr w.r.t. a type] Consider a consistent specification $\mathcal{S}$, a type τ of $\mathcal{S}$ and any c_expr well-formed w.r.t. τ. Then the c_expr is consistent w.r.t. τ iff:
$Consistent(\tau, c_expr)$ = true.

Proof. Similar to Theorem 3.1. □

Below, two further lemmata follow, that are related to the equivalence of a pair of a_expr's and c_expr's, respectively.

Lemma 3.3 [Characterization of the equivalence of a_expr's] Given a consistent specification $\mathcal{S}$, consider a type τ of $\mathcal{S}$ and two a_expr's, a_expr_h, a_expr_k, each well-formed and consistent w.r.t. τ. Then a_expr_h and a_expr_k are equivalent w.r.t. τ tuple iff:
$EquiCheck(\tau, a_expr_h, a_expr_k)$ = true.

Proof. $\Rightarrow$ By contradiction. Assume that $EquiCheck(\tau, a_expr_h, a_expr_k)$ = false. Then, there exists at least one property p_i s.t. $Dom(p_i, a_expr_h) \neq Dom(p_i, a_expr_k)$, and suppose that $y \in Dom(p_i, a_expr_h)$ and $y \notin Dom(p_i, a_expr_k)$. Therefore, it is possible to define a model $\mathcal{I} = (\Delta^{\mathcal{I}}, .^{\mathcal{I}})$ s.t. there exists an element $x \in \Delta^{\mathcal{I}}$, $< x, y > \in (p_i)^{\mathcal{I}}$, $x \in (a_expr_h)^{\mathcal{I}} \cap (\tau)^{\mathcal{I}}$ and $x \notin (a_expr_k)^{\mathcal{I}} \cap (\tau)^{\mathcal{I}}$ (notice that according to Definition 3.1, if $y \in Dom(p_i, a_expr_h)$, and σ is the type of the property p_i in τ, then necessarily $y \in (\sigma)^{\mathcal{I}}$).

$\Leftarrow$ Suppose $EquiCheck(\tau, a_expr_h, a_expr_k)$ = true. Consider an element, say x, such that $x \in (a_expr_h)^{\mathcal{I}} \cap (\tau)^{\mathcal{I}}$. Since for each property p_i of τ, $Dom(p_i, a_expr_h)$ = $Dom(p_i, a_expr_k)$, then $x \in (a_expr_k)^{\mathcal{I}} \cap (\tau)^{\mathcal{I}}$ too. □

Lemma 3.4 [Characterization of the equivalence of c_expr's] Given a consistent specification $\mathcal{S}$, consider a type τ of $\mathcal{S}$ and two c_expr's, c_expr_h, c_expr_k, each well-formed and consistent w.r.t. τ. Then c_expr_h and c_expr_k are equivalent w.r.t. τ tuple iff:
$Equivalent(\tau, c_expr_h, c_expr_k)$ = true.

Proof. Trivial.

□

Finally, by using the above lemmata, the characterization of the consistency of the dynamic component of an OO specification can be formally presented. In particular, first the consistency of the behavior of a type is introduced and, successively, such a result is extended to the dynamic component of an entire specification.

Theorem 3.5 [Characterization of the consistency of the behavior of a type] Given a consistent specification $\mathcal{S}$ and a type τ of $\mathcal{S}$, the behavior B_τ of τ is *consistent* iff the following conditions hold:

1. for each state S of B_τ:
   $Consistent(\tau,S.Range) = true$;
2. for each pair of root states S_1, S_2 of B_τ:
   $Consistent(\tau,S_1.Range \wedge S_2.Range) = false$;
3. for each OR-state S of B_τ:
   for each pair of substates s_1, $s_2 \in S.Substates$:
   $Consistent(\tau,s_1.Range \wedge s_2.Range) = false$;
4. for each AND-state S of B_τ:
   for each pair of substates s_1, $s_2 \in S.Substates$:
   s_1 and s_2 are OR-states s.t.:
   $Equivalent(\tau,s_1.Range \wedge s_2.Range) = true$;
   and for each pair of substates s_1'', s_2'', s.t.:
   $s_1'' \in s_1.Substates$ and $s_2'' \in s_2.Substates$:
   $Consistent(\tau,s_1''.Range \wedge s_2''.Range) = true$;
5. for each transition t of B_τ:
   (a) $Consistent(\tau,t.PreC) = true$;
   (b) $Consistent(\tau,S_1.Range \wedge \ldots \wedge S_n.Range \wedge t.PostC) = true$;
   (c) $Consistent(\tau,not(S_1.Range \wedge \ldots \wedge S_n.Range) \wedge t.PostC) = false$,

   where $S_i \in t.Target_States$, for $i = 1...n$.

Proof. The thesis follows directly from Lemmata 3.2,3.3,3.4.

□

Example 3.7 According to the previous theorem, it is easy to see that the behavior of the type book of Figure 2 is consistent. Just to show a few examples consider, for instance, the root states book on stock and book in archive. They are disjoint since:

$Consistent$(book, book on stock.Range $\wedge$ book in archive.Range) $= false$.

In fact, in correspondence with the status property, we have:

Dom(status, status = library $\wedge$ status = archive $\wedge$ age $>$ 10) = $\emptyset$.

Now, consider for instance transition t2. Condition 5(a) holds since:

$Consistent$(book, t2.PreC) = $Consistent$(book, status = new) = true.

Regarding the postconditions of t2, the following holds (see condition 5(b)):

$Consistent$(book, book in catalogue.Range $\wedge$ marked book.Range $\wedge$ t2.PostC) = true.

Furthermore, condition 5(c) holds since there exists at least one of the properties status, registered, or signed for which the Domain set is empty. Therefore, the postconditions of t2 imply the ranges of the target states. □

Of course, the consistency of the behavior of an OO specification is obtained by generalizing the theorem above as follows:

Corollary 3.6 [Characterization of the consistency of the dynamic component of an OO specification] Given a consistent OO specification $\mathcal{S}$, the dynamic component of $\mathcal{S}$ is consistent iff, for each type τ of $\mathcal{S}$, the behavior B_τ of τ satisfies the conditions of Theorem 3.5. □

The computational complexity of the proposed method can be evaluated similarly to the complexity of the satisfiability problem of Boolean expressions [27]. Such a problem can be solved in exponential time in the size of a given expression, by an exhaustive algorithm that tries all possible combinations of truth values for the variable appearing in the expression.

4 Conclusion

In this paper, a formal framework for the consistency checking of both the static and dynamic components of an OO specification has been presented. The consistency checking method proposed in this paper could be employed in different research fields such as, for instance, *schema transformations* or *schema integration*, whose main goal is to support the analysis and design phases at best (see, for instance, [18, 19, 20, 29]).

In future work, we will analyze possible extensions of the OO specification language presented in this paper, for instance, by including constraint expressions comparing not only attribute values with constants, but also attribute values among

them. However, since the more expressive the language the harder the reasoning with the language expressions, a deep preliminary analysis about the trade-off between the expressive power of the language and the possibility of reasoning with it is required. Such an activity, i.e., the identification of fragments of formal logic that allow decidable reasoning methods to be defined, is one of the main challenges of conceptual modeling that is beyond the scope of this paper.

References

[1] M. von der Beeck; "A comparison of statecharts variants"; in L. de Roever and J. Vytopil, (Eds.), *Formal Techniques in Real-Time and Fault-Tolerant Systems*, *Lecture Notes in Computer Science* 863, pp.128-148, Springer-Verlag, New York, 1994.

[2] C.Beeri, A.Formica, M.Missikoff; "Inheritance Hierarchy Design in Object-Oriented Databases"; *Data & Knowledge Engineering* (DKE), Vol.30, No.3, pp.191-216, July 1999.

[3] D.Beneventano, S.Bergamaschi, S.Lodi, C.Sartori; "Consistency Checking in Complex Object Database Schemata with Integrity Constraints"; *IEEE Transactions on Knowledge and Data Engineering*, Vol.10, No.4, July/August 1998.

[4] F.Bancilhon, C.Delobel, P.Kanellakis (Ed.s); "Building an Object-Oriented Database System: The Story of O2", *Morgan Kaufman*, 1992.

[5] G.Booch; "Object-Oriented Design with Applications"; *Benjamin Cummings*, 1991.

[6] F.Bry, N.Eisinger, H.Schutz, S.Torge; "SIC: Satisfiability Checking for Integrity Constraints"; *Proc. of Deductive Databases and Logic Programming (DDLP'98), workshop at JICSLP*, 1998.

[7] D.Calvanese, M.Lenzerini; "Making Object-Oriented Schemes More Expressive"; *Proc. of the 13th Int. Symp. on Principles of Database Systems '94* (PODS'94); Minneapolis, USA, May 1994.

[8] D.Calvanese, M.Lenzerini, D. Nardi; "Description Logics for Conceptual Data Modeling"; in *Logics for Databases and Information Systems*, *Kluwer Academic Publisher*, Jan Chomicki and Günter Saake (Eds.), pp.229-263, 1998.

[9] G. Castagna; "Covariance and contravariance: conflict without a cause"; *ACM Transactions on Programming Languages and Systems*, 17(3), pp.431-477, March 1995.

[10] R.G.G.Cattel et Al.; "The Object Data Standard: ODMG 3.0"; *Academic Press*, 1999.

[11] C.Chang, R.Lee; "Symbolic Logic and Mechanical Theorem Proving"; *Academic Press*, London 1987.

[12] P.Chen; "The Entity-Relationship Model - Toward a Unified View of Data"; *ACM Transaction on Database Systems*, pp.9-36, March 1976.

[13] N.Coburn, G.E.Weddell; "Path Constraints for Graph-Based Data Models: Towards a Unified Theory of Typing Constraints, Equations, and Functional Dependencies"; *Proc. of Int. Conf. on Deductive and Object-Oriented Databases '91* (DOOD'91), Munich, Germany, 1991; Lecture Notes in Computer Science (LNCS) 566, Springer-Verlag.

[14] D.G.Firesmith; "The inheritance of state models"; *Report on Object Analysis and Design (ROAD)*, 2(6), pp.13-15, March 1996.

[15] A.Formica; "Finite Satisfiability of Integrity Constraints in Object-Oriented Database Schemas"; To appear in *IEEE Transactions on Knowledge & Data Engineering*; 2001.

[16] A.Formica, H.D.Groger, M.Missikoff; "An Efficient Method For Checking Object-Oriented Database Schema Correctness"; *ACM Transactions on Database Systems* (TODS), 23 (3), pp.333-369, 1998.

[17] A.Formica, M.Missikoff, R.Terenzi; "Constraint Satisfiability in Object-Oriented Databases"; *East-West Database Workshop, Klagenfurt 1994*; Workshops in Computing, J.Eder and L.A.Kalinichenko (Eds.), pp.48-60, London, Springer-Verlag, 1995.

[18] H.Frank, J.Eder; "Equivalence transformations on statecharts"; in S.K.Chang, (Ed.), *Twelfth International Conference on Software Engineering and Knowledge Engineering - SEKE'2000*; pp.150-158, KSI, July 2000.

[19] H.Frank, J.Eder; "Integration of statecharts"; in M.Halper, (Ed.), *Third IFCIS International Conference on Cooperative Information Systems (CoopIS98)*, pp.364-372; IEEE Computer Society, August 1998.

[20] H.Frank, J.Eder; ”Towards an automatic integration of statecharts”; in J.Akoka, M.Bouzeghoub, I.Comyn-Wattiau, and E.Metais, (Eds.), *Conceptual Modeling - ER’99*, pp.430-444. Springer Verlag, LNCS 1728, November 1999.

[21] M.R.Genesereth, N.J.Nilsson; ”Logical Foundations of Artificial Intelligence”; *Morgan Kaufmann*; Los Altos, CA, 1987.

[22] D.Harel; ”Statecharts: A visual formalism for complex systems”; *Science of Computer Programming*, 8, pp.231-274, 1987.

[23] D.Harel; ”On visual formalisms”; *Communications of the ACM*, 31(5), pp.514-530, May 1988.

[24] D.Harel, A.Naamad; ”The statemate semantics of statecharts”; *ACM Transactions on Software Engineering and Methodology*, 5(4), pp.293-333, October 1996.

[25] I.Jacobson, M.Christerson, P.Jonsson, G.Overgaard; ”Object Oriented Software Engineering: a use case driven approach”; *Addison-Wesley*, 1992.

[26] A.Kemper, G.Moerkotte; ”Object-Oriented Database Management: Applications in Engineering and Computer Science”; *Prentice Hall*, 1994.

[27] C.H.Papadimitriou; ”Computational Complexity”; *Addison Wesley*, 1995.

[28] J.Rumbaugh, M.Blaha, W.Premerlani, F.Eddy, and W.Lorensen; ”Object-Oriented Modeling and Design”; *Prentice Hall International, Inc*, 1991.

[29] C.Türker, G.Saake; ”Deriving relationships between integrity constraints for schema comparison”; in W.Litwin, T.Morzy, and G.Vossen, (Eds.), *Advances in databases and information systems (ADBIS’98)*, pp.188-199; Springer, LNCS 1475, 1998.

[30] K.Truemper; ”Effective Logic Computation”; *Wiley-Interscience Pub.*, New York, 1998.

[31] Rational Software et.al.; ”Unified Modeling Language” (UML) version 1.3; *http://www.rational.com/uml*, June 1999.

[32] J.Warmer, A.Kleppe; ”OCL: The constraint language of the UML”; *The Journal of Object-Oriented Programming (JOOP)*, 12(2), pp.10-13, May 1999.

[33] R.Wieringa; ''A Survey of Structured and Object-Oriented Software Specification Methods and Techniques''; *ACM Computing Surveys*, 30 (4), pp. 459-527, 1998.