A. Formica, H. Frank

CONSISTENCY OF THE STATIC AND DYNAMIC COMPONENTS OF OBJECT-ORIENTED SPECIFICATIONS

R. 532 Ottobre 2000

Anna Formica — Istituto di Analisi dei Sistemi ed Informatica del CNR, viale Manzoni 30 - 00185 Roma, Italy. Email: formica@iasi.rm.cnr.it.

Heinz Frank — University of Klagenfurt, Universitätsstr. 65-67, 9020 Klagenfurt, Austria. Email: heinz@ifi.uni-klu.ac.at.

ISSN: 1128-3378
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, often, are 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 an Object-Oriented specification.
1. Introduction

"Object-Oriented modeling and design is a new way of thinking about problems using models organized around real-world concepts" [27]. Nowadays, various Object-Oriented (OO) modeling and design methodologies have been consolidating, such as, for instance OMT [27], OOD [5], OOSE [25] and, in particular, UML [30], 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 [31]. 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's 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's behavior was either ignored or it was 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's 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, the contribution of this 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. Therefore, a unifying framework has been defined that allows the consistency of both the static and dynamic components of an OO specification to be checked.

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

As already mentioned, in the literature, static and behavioral models are generally assumed to be consistent. However, since in this paper the consistency checking of these models is treated in terms of consistency checking of a set of integrity constraints, it is interesting to briefly recall some of the existing proposals concerning the integrity constraint satisfiability (consistency), 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 is addressed. However, in that paper a different class of constraint expressions is 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 [29]. 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, this method cannot be adopted in our data model since we do not preliminary define finite ranges of values that the properties of types can assume.

2. The static and dynamic models

2.1. The static model

In this paper, a simple OO specification language that is based on the notion of a type is presented. Such a language 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 definition and a constraint expression. The definition is given by a set of typed properties (tp) enclosed in square brackets (tuple). A constraint expression (cExpr) is a disjunction of conjunctions (disjunctive normal form) of expressions of the form: "p θ K" (single expression, indicated as sExpr for short), where p is the name of a property, θ stands for a comparison operator such as "="", "≥", ">", "", 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 cExpr is given, associated with employee, establishing a lower bound for the consultant (consult) and manager (mgr) salaries.

Example 2.1.

vehicle ≤ [maker:string,
        owner:employee, color:(red, green, blue), production_date:integer]
employee ≤ [name:string,
salary:integer, drives:vehicle, status:(depend, consult, mgr), boss:employee],
(salary > 2000 \land status = consult) \lor
(salary > 4000 \land status = mgr)

Notice that, a property is identified by a *name* and 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, in 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 can take only one value in correspondence with each property.

Notice that, in this paper, inheritance is not addressed. In particular, types will be 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) \( \tau \) if all the properties defining it are properties of \( \tau \). For instance, the *c.expr* associated with the employee type above 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 definition);
- there are no dangling type names (i.e., every type name is defined);
- every *c.expr* associated with a type \( \tau \) is well-formed w.r.t. \( \tau \).

The set of types given in Example 2.1 is a simple 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 \( \text{salary > red} \) 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 \( S \), let \( T \) be the set of types of \( S \), consisting of type *names*, *atomic-types*, and *valueset-types* (that correspond to the atomic concepts in [8]), and let \( P \) be the set of property *names* of \( S \) (corresponding to the atomic *roles* in [8]).
An interpretation \( \mathcal{I} = (\Delta^\mathcal{I}, \mathcal{I}) \) over \( S \) consists of a non-empty 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 \( \tau \)) and every property \( p \in \mathcal{P} \) to a subset \( p^\mathcal{I} \) of \( \Delta^\mathcal{I} \times \Delta^\mathcal{I} \). Then, a type-expression:

\[
\tau \subseteq \text{type-definition, } c.expr
\]
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 \( \tau \)), 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):

- \((\text{type-definition, } c.expr)^\mathcal{I} = (\text{type-definition})^\mathcal{I} \cap (c.expr)^\mathcal{I}\)
- \((\text{type-definition})^\mathcal{I} = \bigcap_j ([p_j : \text{type}_j])^\mathcal{I} = \bigcap_j \{x \in \Delta^\mathcal{I} \mid \# \{y : y > p_j \text{ and } y \in (\text{type}_j)^\mathcal{I} = 1 \}\}
  that corresponds to the Description Logics construct \((\exists p_j . \text{type}_j)^\mathcal{I}\);
- \((c.expr)^\mathcal{I} = (a.expr_1 \lor a.expr_j)^\mathcal{I} = (a.expr_1)^\mathcal{I} \cup (a.expr_j)^\mathcal{I}\)
- \((a.expr)^\mathcal{I} = (s.expr_1 \land s.expr_j)^\mathcal{I} = (s.expr_1)^\mathcal{I} \cap (s.expr_j)^\mathcal{I}\)
- \((s.expr)^\mathcal{I} = (p \theta K)^\mathcal{I} = \{x \in \Delta^\mathcal{I} \mid \forall y : y > p \Rightarrow y \theta K\}
  where \( \theta \) is one of the comparison operators \( \geq, >, =, \neq \), etc..

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

\((\tau)^\mathcal{I} \subseteq (\text{type-definition, } c.expr)^\mathcal{I} \).

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

Finally, \( S \) is \textit{consistent} if it admits a model for which each type is consistent in \( S \).

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

Consider a consistent specification \( S \), an inclusion assertion of \( S \) as defined above, and a constraint expression, say \( c.expr_1 \), that is well-formed w.r.t. \( \tau \). Then \( c.expr_1 \) is \textit{consistent w.r.t. \( \tau \)} iff there exists at least one model \( \mathcal{I} \) of \( S \) for which:

\((\tau)^\mathcal{I} \cap (c.expr_1)^\mathcal{I} \neq \emptyset \).

Under the same assumptions above, consider the constraint expressions \( c.expr_k \) and \( c.expr_h \) well-formed w.r.t. \( \tau \). Then \( c.expr_k \) and \( c.expr_h \) are:

- \textit{equivalent w.r.t. \( \tau \) type-definition} iff for any model \( \mathcal{I} \) of \( S \) the following holds:
  \((c.expr_k)^\mathcal{I} \cap (\text{type-definition})^\mathcal{I} = (c.expr_h)^\mathcal{I} \cap (\text{type-definition})^\mathcal{I} \)
  where \( c.expr_k \) and \( c.expr_h \) are also supposed to be consistent w.r.t. \( \tau \);
- \textit{disjoint w.r.t. \( \tau \) type-definition} iff for any model \( \mathcal{I} \) of \( S \) the following holds:
  \((c.expr_k)^\mathcal{I} \cap (c.expr_h)^\mathcal{I} \cap (\text{type-definition})^\mathcal{I} = \emptyset \).
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 [30]. We call this condition 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 [30]. 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:

```plaintext
book ≜ [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.

![Figure 1: A simplified behavior associated with the book type](image)

<table>
<thead>
<tr>
<th>state</th>
<th>transition</th>
<th>event</th>
</tr>
</thead>
<tbody>
<tr>
<td>book on stock</td>
<td></td>
<td></td>
</tr>
<tr>
<td>book on loan</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>source state</th>
<th>guard</th>
<th>target state</th>
</tr>
</thead>
<tbody>
<tr>
<td>t3: lending [reserved = false]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Ranges of the states of Figure 1

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, first 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, first 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 out of stock (i.e., books which are borrowed or in the text book collection) are returned to the library, if they are not reserved. Books, older than 10 years can be given into the archive. Such books can be borrowed too, but can only be returned into the archive.

In the next sections the example will be used to discuss the statechart language in a more detailed way. For a deeper understanding about extensions of the statechart language we refer the reader to [18].
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 subclasses which are related to each other by "exclusive-or", i.e., an object can only be in one subclass of an OR-state at any time. AND-states have subclasses which are "and" related, i.e., an object, that is in an AND-state is also in all subclasses of the AND-state. Basic states are the states at the bottom of a state hierarchy, i.e., they do not have subclasses. 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 subclasses book registration and book processing. These two subclasses, together with the state book not on stock, 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 must 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 subclasses as follows.

Definition 2.2. [Range of a state] The range of a state S is defined as:

- a c.Expr, if S is a basic state;
- the disjunction of the ranges of all the subclasses of S, if S is an OR-state;
- the conjunction of the ranges of all the subclasses of S, if S is an AND-state.

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 subclasses borrowed book and book in text book collection, resulting therefore:

status = borrowed \lor (\text{status} = \text{in text book collection} \land \text{reserved} = \text{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 of which is obtained by the disjunction of the ranges of its component states.

2.2.2. Events and transitions

An event is an incident whose goal is to change 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 state) when the event occurs and some specified condition (called the guard of transition) holds.

**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 text book collection, triggered by the event return. The guard of the transition is \( \text{reserved} = \text{true} \). The event return also triggers transitions t7 and t11.

Notice that a transition may have more than one source (or target) state (see, for instance, transition t5 having the source states registered book and signed book). Therefore, below we will speak about source and target states of a transition.

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. Therefore, the preconditions of a transition are defined as the conjunction of the ranges of the source states and the guard of the transition.

After a transition has been applied, the object has to satisfy the ranges of its target states. Therefore, the postconditions of a transition must imply the ranges of its target states.

Similarly to the ranges of states, as a specification language for guards, pre- and postconditions, \( c.expr \)'s will be used. In Table 3, the postconditions of transitions of the example of Figure 2 are shown.

**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 \( \text{reserved} = \text{true} \), resulting therefore:

\[ \text{status} = \text{borrowed} \land \text{reserved} = \text{true} \]

The preconditions of transition t5 correspond to the conjunction of the ranges of its source states

<table>
<thead>
<tr>
<th>new book</th>
<th>status = new</th>
</tr>
</thead>
<tbody>
<tr>
<td>book in catalogue</td>
<td>status = preparation \land \text{registered} = \text{false} \land \text{signed} = \text{false} \lor \text{registered} = \text{false} \land \text{signed} = \text{true}</td>
</tr>
<tr>
<td>marked book</td>
<td>status = preparation \land \text{signed} = \text{false} \land \text{registered} = \text{false} \lor \text{registered} = \text{false} \land \text{signed} = \text{true}</td>
</tr>
<tr>
<td>registered book</td>
<td>status = preparation \land \text{registered} = \text{true} \land \text{signed} = \text{false} \lor \text{registered} = \text{true} \land \text{signed} = \text{true}</td>
</tr>
<tr>
<td>signed book</td>
<td>status = preparation \land \text{signed} = \text{true} \land \text{registered} = \text{false} \lor \text{registered} = \text{true} \land \text{signed} = \text{true}</td>
</tr>
<tr>
<td>book on stock</td>
<td>status = \text{in library}</td>
</tr>
<tr>
<td>borrowed book</td>
<td>status = \text{borrowed}</td>
</tr>
<tr>
<td>book in text book</td>
<td>status = \text{in text book collection} \land \text{reserved} = \text{false}</td>
</tr>
<tr>
<td>collection</td>
<td></td>
</tr>
<tr>
<td>book in archive</td>
<td>status = \text{in archive} \land \text{age} &gt; 10</td>
</tr>
</tbody>
</table>

Table 2: Ranges of the basic states of Figure 2
registered book and signed book. The postconditions of t5 are:
\[\text{status} = \text{in library} \land \text{reserved} = \text{false},\]
that imply the range of the state book on stock.

2.2.3. The behavior of a type

In this paper, states and transitions will be indicated by using the following notation. \(S\) denotes the range of the state (whose name is) \(S\), whereas \(t.Prec\), \(t.Post\), and \(t.Guard\) denote the preconditions, the postconditions, and the guard of the transition (whose label is) \(t\), respectively. Such conditions can be combined by disjunction and conjunction. For instance in our example, the preconditions of transition t9 are the conjunction of borrowed book.Range and t9.Guard.

If \(S\) is an OR-state or an AND-state, then \(S.Substates\) denotes the set of the (names of the) direct subtates of \(S\). In the example of Figure 2, book administration.Substates contains the states book registration and book processing.

If \(t\) is a transition, \(t.Target\) and \(t.Source\) denote the sets of the target states and source states of \(t\), respectively. In the example, for instance, t5.Source contains the states registered book and signed book.

\textbf{Definition 2.3. [Behavior of a type]} The behavior of a type \(\tau\), indicated as \(B_\tau\), is a statechart whose ranges of the basic states, postconditions and guards of transitions are \texttt{c.expr}'s well-formed w.r.t. \(\tau\).

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.

\textbf{Definition 2.4. [Equivalent states]} The states \(S_1\) and \(S_2\) of the behavior \(B_\tau\) of the type \(\tau\) are equivalent iff their ranges are equivalent \texttt{c.expr}'s w.r.t. \(\tau\) type-definition.
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, resulting in:

\[
\text{status} = \text{preparation} \wedge \text{registered} = \text{false} \wedge \text{signed} = \text{false} \lor \\
\text{status} = \text{preparation} \wedge \text{registered} = \text{false} \wedge \text{signed} = \text{true} \lor \\
\text{status} = \text{preparation} \wedge \text{registered} = \text{true} \wedge \text{signed} = \text{false} \lor \\
\text{status} = \text{preparation} \wedge \text{registered} = \text{true} \wedge \text{signed} = \text{true}
\]

(see Table 2). □

Definition 2.5. [Disjoint states] The states \( S_1 \) and \( S_2 \) of the behavior \( B_r \) of the type \( \tau \) are disjoint iff their ranges are disjoint \( c_{expr} \)'s w.r.t. \( \tau \) type-definition. □

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 attribute status. □

Definition 2.6. [Orthogonal states] The states \( S_1 \) and \( S_2 \) of the behavior \( B_r \) of the type \( \tau \) are orthogonal iff \( S_1 \) and \( S_2 \) are OR-states which are equivalent, and \( \forall s' \in S_2. \text{Substates} \) there exists at least one model \( \mathcal{I} = (\Delta^\mathcal{I}, \sigma^\mathcal{I}) \) such that:

\[
(s.\text{Range})^\mathcal{I} \cap (s'.\text{Range})^\mathcal{I} \cap (\tau)^\mathcal{I} \neq \emptyset.
\]

Essentially, orthogonality means that for an object, which is in a substate of an OR-state \( S_1 \), it is possible to be in anyone of the substates of \( S_2 \) at the same time. 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 of the ranges of the states marked book and registered book is not empty, since an object with the attribute values: \( \text{status} = \text{preparation}, \text{signed} = \text{false}, \text{registered} = \text{true} \), satisfies the ranges of both these states (see Table 2). □

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

Definition 2.7. [Consistent behavior of a type] Given a consistent OO specification \( S \) and a type \( \tau \) of \( S \), the behavior \( B_r \) of the type \( \tau \) is consistent iff the following conditions hold:

1. for each state \( S \) of \( B_r \), there exists at least one model \( \mathcal{I} \) of \( S \) s.t.:
   \[
   (S.\text{Range})^\mathcal{I} \cap (\tau)^\mathcal{I} \neq \emptyset;
   \]
2. for each pair of root states \( S_1, S_2 \) of \( B_r \):
   \( S_1, S_2 \) are disjoint;
3. for each OR-state \( S \) of \( B_r \):
   for each pair of substates \( s_1, s_2 \in S.\text{Substates} \):
   \( s_1, s_2 \) are disjoint;
4. for each AND-state \( S \) of \( B_r \):
   for each pair of substates \( s_1, s_2 \in S.\text{Substates} \):
   \( s_1 \) and \( s_2 \) are orthogonal;
5. for each transition \( t \) of \( B_\tau \), let \( S_i \in t.\text{Target States} \), for \( i = 1 \ldots n \). Then, for each \( t \):

(a) there exists at least one model \( I \) of \( S \) s.t.:

\[
(t.\text{Pre}C)^I \cap (\tau)^I \neq \emptyset,
\]

(b) there exists at least one model \( I \) of \( S \) s.t.:

\[
\bigcap_i (S_i.\text{Range})^I \cap (t.\text{Post}C)^I \cap (\tau)^I \neq \emptyset,
\]

(c) \( t.\text{Post}C \), \( \text{not}(S_1.\text{Range} \land \ldots \land S_n.\text{Range}) \) are disjoint.

\[\square\]

Let us briefly illustrate these conditions. Condition 2.7(1) deals with the fact that any object that is in a given state must also satisfy the type \( \tau \) and the constraints of \( \tau \).

**Example 2.10.** Suppose we add the constraint \( \text{age} < 10 \) 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, that is:

\[
\text{status} = \text{in archive} \land \text{age} > 10.
\]

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 2.7(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 2.7(3)).

Condition 2.7(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 2.7(5) concerns the pre- and postconditions of the transitions. In particular, the condition 2.7(5a) requires the consistency of the preconditions of a transition with respect to the associated type. Conditions 2.7(5b,5c) are used to express that the postconditions of a transition must imply the ranges of its target states.

**Example 2.11.** Suppose again that \( \text{age} < 10 \) is a constraint associated with the type book. Then the preconditions of transition \( t_6 \) are not consistent with the type book, due to the guard \( \text{age} > 10 \). Suppose we now delete \( \text{age} > 10 \) from the postconditions of transition \( t_6 \) in Table 3. Then, both the conditions 2.7(5b,5c) are violated because the postconditions of \( t_6 \) do not imply the range of the target state, that is:

\[
\text{status} = \text{in archive} \land \text{age} > 10.
\]

By generalizing Definition 2.3, we can speak about the **behavior of an OO specification**, since it is a set of types. Therefore we can introduce the notion of **consistent behavior of an OO specification** defined as follows.

**Definition 2.8.** [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 the types (i.e., their structures and \( e_{expr}'s \)) 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 consistency checking 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 \( \sigma \) be the type of the property \( p \) in \( \tau \). Then, the Interval (Int) of the s_expr is defined as follows:

- if \( \sigma \) is the integer or the real type:
  \[
  \text{Int}(s_{expr}) = (-\infty, K) \quad \text{if} \quad \theta = "<"; \\
  \text{Int}(s_{expr}) = (-\infty, K] \quad \text{if} \quad \theta = "\leq"; \\
  \text{...(and so on in all the other cases)}
  \]
  where ordinary parentheses and square brackets denote open and closed intervals, respectively;

- if \( \sigma \) is the string (bool) type, there are two possible cases. If \( \theta = "," \), \( \text{Int}(s_{expr}) \) is the \( K \) (boolean) constant; otherwise, if \( \theta = ",\neq" \), \( \text{Int}(s_{expr}) \) is the complement of \( K \) in the set of all possible strings (the opposite boolean constant);

- if \( \sigma \) is a valueset-type, \( \text{Int}(s_{expr}) \) behaves as in the previous cases, with the further intersection with the valueset.

\[ \square \]

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):

\[
\text{employee} \leq [\text{salary:integer}, \text{name:string}, \text{status:depend, consult, mgr}, \text{boss:employee}],
\]

\[
\begin{align*}
\text{(salary > 2000 \^ status \neq mgr) (1)} & \vee \\
\text{(salary > 4000 \^ status \neq depend \^ status \neq consult) (2)}
\end{align*}
\]

Then:

\[
\text{Int}(\text{status} \neq \text{consult}) = \{\text{depend, mgr}\}
\]

and:

\[
\text{Int}(\text{salary} > 3000) = (3000, +\infty).
\]

\[ \square \]

Definition 3.2. [Conflicting Expression set] Given a property \( p \) and an a_expr, the Conflicting Expression (ConfEx) set identifies all the s_expr's of the a_expr containing the p property:

\[
\text{ConfEx}(p, a_{expr}) = \{s_{expr}\_h = p \theta K \mid a_{expr} = s_{expr}\_h \ldots \}
\]

\[ \square \]

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:

\[
\{\text{status} \neq \text{depend, status} \neq \text{consult}\}.
\]

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

\[
\text{Dom}(p,a\_expr) = \bigcap_j \{ \text{Int}(s\_expr_j) \mid s\_expr_j \in \text{ConfEx}(p,a\_expr) \}
\]

\( \Box \)

**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:

\[
\text{Int}(\text{status} \neq \text{depend}) = \{ \text{mgr, consult} \}
\]

\[
\text{Int}(\text{status} \neq \text{consult}) = \{ \text{mgr, depend} \}
\]

the Domain set is:

\[
\{ \text{mgr} \}.
\]

\( \Box \)

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 \( \tau \), we assume that \( c\_expr \), stands for the constraint expression defining the type \( \tau \).

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

\[
c\_expr = a\_expr_1 \lor \ldots \lor a\_expr_n
\]

\[
c\_expr = a\_expr_{\tau,1} \lor \ldots \lor a\_expr_{\tau,m}
\]

then, consistency holds iff at least one \( a\_expr_i \) of \( c\_expr \) and one \( a\_expr_{\tau,j} \) of \( c\_expr \) exist such that their conjunction is consistent w.r.t. \( \tau \). This is checked by using the AndCheck procedure. Of course, if the type \( \tau \) does not contain any constraint expression, only \( a\_expr_i \) must be consistent w.r.t. \( \tau \).

**Procedure 1 The Consistent procedure**

**input:** a type \( \tau \), possibly including \( c\_expr = a\_expr_{\tau,1} \lor \ldots \lor a\_expr_{\tau,m} \)

and a \( c\_expr \) well-formed w.r.t. \( \tau \):

\[
c\_expr = a\_expr_1 \lor \ldots \lor a\_expr_n
\]

**output:** \( \text{true} \), if the \( c\_expr \) is consistent w.r.t. \( \tau \); \( \text{false} \), otherwise

\[
\text{Consistent}(\tau,c\_expr) \leftarrow \text{false}
\]

if \( \exists\ a\_expr_i\ a\_expr_{\tau,j} \) s.t.:

\[
\text{AndCheck}(\tau,a\_expr_i \land a\_expr_{\tau,j}) = \text{true}
\]

\[
\text{Consistent}(\tau,c\_expr) \leftarrow \text{true}
\]

**end if**

**Example 3.4.** Consider the type employee of Example 3.1, and the following \( c\_expr \) (that is indeed an \( a\_expr \)):

\[
\text{salary} > 5000 \land \text{status} = \text{mgr}.
\]

In order to check if it is consistent w.r.t. employee, one of the following two \( a\_expr \)'s:

\[
(\text{salary} > 2000 \land \text{status} \neq \text{mgr} \land \text{salary} > 5000 \land \text{status} = \text{mgr})
\]

\[
(\text{salary} > 4000 \land \text{status} \neq \text{depend} \land \text{status} \neq \text{consult} \land
\]

...
must be consistent w.r.t. \( \tau \). This check is performed by the AndCheck procedure informally illustrated below.

Given a type \( \tau \) and any \( a_{expr} \) well-formed w.r.t. \( \tau \), the AndCheck procedure returns \textit{true}, if the \( a_{expr} \) is consistent w.r.t. \( \tau \); \textit{false}, otherwise. Essentially, the AndCheck procedure reports consistency if for each property \( p_i \) of the type \( \tau \) 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. \( \tau \) since the property \( p_i \) cannot be instantiated.

**Example 3.5.** Consider Example 3.4. Only the second \( a_{expr} \) is consistent w.r.t. \( \tau \), since:

\[
\begin{align*}
\text{Dom}(\text{salary}, a_{expr}) &= (5000, +\infty), \\
\text{Dom}(\text{status}, a_{expr}) &= \{\text{mgr}\}.
\end{align*}
\]

Let us briefly illustrate the Equivalent procedure. Such a procedure allows us to determine if two \( c_{expr} \)'s well-formed w.r.t. \( \tau \) are equivalent w.r.t. \( \tau \) type-definition. In particular, two constraint expressions \( c_{expr_1} \) and \( c_{expr_2} \) are equivalent if for each \( a_{expr_i} \in c_{expr_1} \) there exists an equivalent \( a_{expr_j} \in c_{expr_2} \) and vice versa. The equivalence of the \( a_{expr_i} \) and \( a_{expr_j} \) is checked by the EquaCheck procedure. In particular, \( a_{expr_i} \) and \( a_{expr_j} \) are equivalent if, for all the properties \( p \) of \( \tau \), 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:

\[ a_{expr_1} = \text{status} \neq \text{mgr} \wedge \text{status} \neq \text{consult} \]

and:

\[ a_{expr_2} = \text{status} = \text{mgr}. \]

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

\[
\text{Dom}(\text{status}, a_{expr_1}) = \text{Dom}(\text{status}, a_{expr_2}) = \{\text{mgr}\}
\]

and for all the other properties of employee, for both the expressions the Domain set is empty (see also Example 3.3).

### 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 characterization of the consistency of the static component is obtained by applying the Consistent procedure to all the constraint expressions of the types of the specification.

**Theorem 3.1.** [Characterization of the consistency of the static component of an OO specification] The static component of an OO specification \( S \) is consistent iff for each type \( \tau \) of \( S \) containing constraints:

\[ \text{Consistent}(\tau, c_{expr_\tau}) = \text{true}. \]

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

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

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

By iterating the above steps for each type of \( S \), it is possible to construct at least one model of \( S \) for which each type is consistent in \( S \).

The following lemma is a generalization of the previous theorem to the case of any \( c_{\text{expr}} \) well-formed w.r.t. \( \tau \).

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

Proof. Similar to Theorem 3.1.

Below, two further lemmata follow, that are related to the equivalence of a pair of \( a_{\text{expr}} \)'s and \( c_{\text{expr}} \)'s, respectively.

**Lemma 3.3. [Characterization of the equivalence of \( a_{\text{expr}} \)'s]** Given a consistent specification \( S \), consider a type \( \tau \) of \( S \) and two \( a_{\text{expr}} \)'s, \( a_{\text{expr}_h}, a_{\text{expr}_k} \), each of which is well-formed and consistent w.r.t. \( \tau \). Then \( a_{\text{expr}_h} \) and \( a_{\text{expr}_k} \) are equivalent w.r.t. \( \tau \) type-definition iff:
\[
\text{EquiCheck}(\tau, a_{\text{expr}_h}, a_{\text{expr}_k}) = \text{true}.
\]

Proof. \( \Rightarrow \) By contradiction. Assume that \( \text{EquiCheck}(\tau, a_{\text{expr}_h}, a_{\text{expr}_k}) = \text{false} \). Then, there exists at least one property \( p_i \) s.t. \( \text{Dom}(p_i, a_{\text{expr}_h}) \neq \text{Dom}(p_i, a_{\text{expr}_k}) \), and suppose that \( y \in \text{Dom}(p_i, a_{\text{expr}_h}) \) and \( y \notin \text{Dom}(p_i, a_{\text{expr}_k}) \). Therefore, it is possible to define a model \( \mathcal{I} = (\Delta^\mathcal{I}, \mathcal{I}) \) s.t. there exists an element \( x \in (\tau)^\mathcal{I} \) and \( <x, y> \in (p_i)^\mathcal{I} \) (in fact, according to Definition 3.1, if \( y \notin \text{Dom}(p_i, a_{\text{expr}_h}) \), and \( \sigma \) is the type of the property \( p_i \) in \( \tau \), then necessarily \( y \in (\sigma)^\mathcal{I} \). Then, \( x \in (a_{\text{expr}_h})^\mathcal{I} \cap (\tau)^\mathcal{I} \) (\( \neq \emptyset \) from the hypothesis) that does not belong to \( (a_{\text{expr}_k})^\mathcal{I} \cap (\tau)^\mathcal{I} \) (\( \neq \emptyset \) from the hypothesis), and vice versa.

\( \Leftarrow \) Suppose \( \text{EquiCheck}(\tau, a_{\text{expr}_1}, a_{\text{expr}_2}) = \text{true} \). Consider an element, say \( x \), such that \( x \in (a_{\text{expr}_1})^\mathcal{I} \cap (\tau)^\mathcal{I} \). Since for each property \( p_i \) of \( \tau \), \( \text{Dom}(p_i, a_{\text{expr}_1}) = \text{Dom}(p_i, a_{\text{expr}_2}) \), it

\(^1\)Notice that it is not possible to get an infinite loop because constraint expressions on 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.
is easy to see that $x \in (a_{expr})^\tau \cap (\tau)^\tau$ too (and vice versa starting from $x \in (a_{expr})^\tau \cap (\tau)^\tau$).

\[\Box\]

**Lemma 3.4.** [Characterization of the equivalence of $c_{expr}$'s] Given a consistent specification $S$, consider a type $\tau$ of $S$ and two $c_{expr}$'s, $c_{expr_h}$, $c_{expr_k}$, each of which is well-formed and consistent w.r.t. $\tau$. Then $c_{expr_h}$ and $c_{expr_k}$ are equivalent w.r.t. $\tau$ type definition iff:

$$\text{Equivalent}(\tau,c_{expr_h},c_{expr_k}) = true.$$  

**Proof.** Trivial.  

Finally, by using the above lemmata, we are able to present the characterization of the consistency of the dynamic component of an OO specification. In particular, first the consistency of the behavior of a type is presented. Then this result is trivially extended to the consistency of the dynamic component of an entire specification.

**Theorem 3.5.** [Characterization of the consistency of the behavior of a type] Given a consistent specification $S$ and a type $\tau$ of $S$, the behavior $B_\tau$ of the type $\tau$ is consistent iff the following conditions hold:

1. for each state $S$ of $B_\tau$:
   $$\text{Consistent}(\tau,S.\text{Range}) = true;$$

2. for each pair of root states $S_1$, $S_2$ of $B_\tau$:
   $$\text{Consistent}(\tau,S_1.\text{Range} \land S_2.\text{Range}) = false;$$

3. for each OR-state $S$ of $B_\tau$:
   for each pair of subtrees $s_1$, $s_2 \in S.\text{Substates}$:
   $$\text{Consistent}(\tau,s_1.\text{Range} \land s_2.\text{Range}) = false;$$

4. for each AND-state $S$ of $B_\tau$:
   for each pair of subtrees $s_1$, $s_2 \in S.\text{Substates}$:
   $s_1$ and $s_2$ are OR-states s.t.:
   $$\text{Equivalent}(\tau,s_1.\text{Range} \land s_2.\text{Range}) = true;$$
   and for each pair of subtrees $s_1''$, $s_2''$, s.t.:
   $s_1'' \in s_1.\text{Substates}$ and $s_2'' \in s_2.\text{Substates}$:
   $$\text{Consistent}(\tau,s_1''.\text{Range} \land s_2''.\text{Range}) = true;$$

5. for each transition $t$ of $B_\tau$:
   (a) $\text{Consistent}(\tau,t.\text{PreC}) = true$;
   (b) $\text{Consistent}(\tau,S_1.\text{Range} \land \ldots \land S_n.\text{Range} \land t.\text{PostC}) = true$;
   (c) $\text{Consistent}(\tau,\text{not}(S_1.\text{Range} \land \ldots \land S_n.\text{Range}) \land t.\text{PostC}) = false$,

where $S_i \in t.\text{Target States}$, for $i = 1..n$.

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

$\Box$
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:
\[ \text{Consistent(book, book on stock. Range} \land \text{book in archive. Range)} = false. \]

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

\[ \text{Dom(status, status = library} \land \text{status = archive} \land \text{age} > \text{10)} = \emptyset. \]

Now, consider for instance transition t2. Condition 5(a) holds since the preconditions of t2 are
not disjoint with the type book, i.e.:
\[ \text{Consistent(book, t2. PreC)} = \text{Consistent(book, status = new)} = true. \]

With regard to the postconditions of t2, it results that they are not disjoint with the ranges of
the target states (see condition 5(b)), since:
\[ \text{Consistent(book, book in catalogue. Range} \land \text{marked book. Range} \land \text{t2. PostC)} = true. \]

In fact, it is easy to see that, once transformed in disjunctive form, in the above expression there
exists the \( a_{expr} \):

\[ \text{status = preparation} \land \text{registered = false} \land \text{signed = false} \]

for which the \text{AuditCheck} procedure holds. Furthermore, the condition 5(c) holds since for each
\( a_{expr} \) there exists 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 above theorem as follows:

Corollary 3.6. [Characterization of the consistency of the dynamic component an
OO specification] Given a consistent OO specification \( S \), the dynamic component of \( S \) is
consistent iff, for each type \( \tau \) of \( S \), the behavior \( B_\tau \) of the type \( \tau \) satisfies the conditions of
Theorem 3.5. \( \square \)

4. Conclusion

In this paper, a language for OO specifications has been proposed, and a method for the con-
sistency 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 dif-
ferent research fields such as, for instance, 	extit{schema transformations} or 	extit{schema integration}, whose
main goal is to support the analysis and design phases at best, as for instance \([18, 19, 20, 28]\).

The specification language on which we focused in this paper is very simple. In future work, we
mean to extend it, 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 goes beyond the scope of this paper.
References


