An Attempt at Analysing the Consistency Problems in the UML from a Classical Algebraic Viewpoint [*]

E. Astesiano and G. Reggio

DISI, Università di Genova - Italy

Abstract. In this paper, after introducing the problem and a brief survey of the current approaches, we look at the consistency problems in the UML in terms of the well-known machinery of classical algebraic specifications. Thus, first we review how the various kinds of consistency problems were formulated in that setting. Then, and this is the first contribution of our note, we try to reduce, as much as possible, the UML problems to that frame. That analysis, we believe, is rather clarifying in itself and allows us to better understand what is new and what instead could be treated in terms of that machinery. We conclude with some directions for handling those problems, basically with constrained modelling methods that reduce and help precisely individuate the sources of possible inconsistencies.

1 Introduction

Few years after its introduction, the UML has become, for good and bad, a "lingua franca" for an object-oriented support to software development. Among the many problems raised by its use, the consistency of UML artifacts/documents has emerged as crucial and challenging. Indeed, it has attracted in recent years, especially after the year 2000, a significant amount of work; that interest is also witnessed by the organization of a workshop within the conference ≪UML≫ 2002 on "Consistency Problems in UML-based Software Development" [1].

Within that context "consistency" bears roughly the same meaning of logical consistency; indeed the UML artifacts/documents are organized in "models" that correspond to logical specifications. However the nature of those artifacts is, at least apparently, so different from the traditional specifications used in the formal method community that even the definition of consistency is somewhat controversial. Among the sources of difficulty we can mention: the nature of the notation, that is visual and not directly defined adopting inductive techniques; the UML multiview approach, which describes a system, at some level of abstraction, as a collection of sub-descriptions dealing with possibly overlapping aspects; the use of UML artifacts throughout a software development process

[*] Work supported by the Italian National Project SAHARA (Architetture software per infrastrutture di rete ad accesso eterogeneo).

typically consisting of many phases, in which the same kind of document may have different meanings. With Engels et al. [18], we can say that *"Altogether the consistency conditions depend on the diagrams [constructs] involved, the development process employed, and the current stage of the development."*

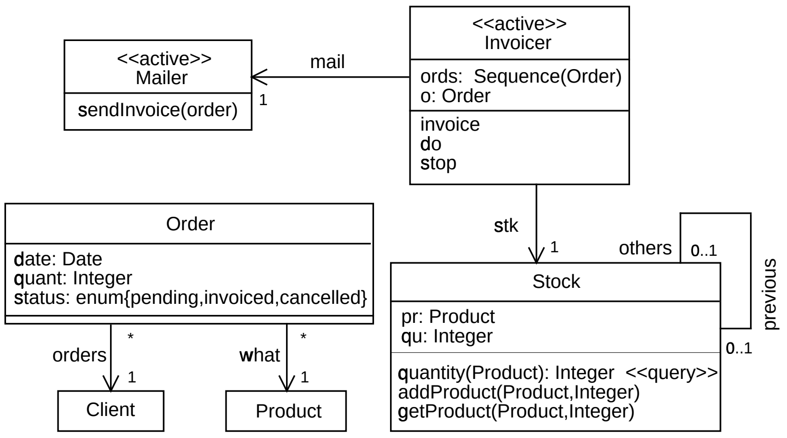
Confronted with the consistency problems in the UML, first during some work on UML semantics done within the CoFI Initiative [30] and now in the development of a UML-based method [5, 6], because of our long acquaintance with algebraic development techniques, we have been tempted to look at them trying to borrow the classical frame of logical-algebraic specifications, also for understanding and isolating the possible novelties. Indeed, in that framework one could define concepts and problems in a rather rigorous way, though of course that does not mean to solve all consistency problems, many of them being of uncomputable nature. However, among the many related valuable research attempts at dealing with consistency in UML (consider, e.g., [1] also for further references) that view is not explicit; though it appears to be underlying some treatment, notably in [18], where some rather clean informal definitions have a clear logical origin. The analysis in our paper will help understand why most of the related papers departs from the logical-algebraic approach; indeed the relationship is not obvious at all.

The purpose of this paper is first of all informative and exploratory, to establish a link between the work and terminology in the UML world and in the logical-algebraic setting. To achieve that purpose we first present extensively the origin and the nature of the consistency problems in the UML in Sect. 3 and briefly review the current terminology and research directions in Sect. 4. Then in Sect. 5, avoiding many technicalities, in a simple language adopting some conceptual object-oriented notation (a subset of UML), we recall the basic setting of logical-algebraic specifications; and, in Sect. 6, we propose a view of the UML setting in terms of the classical concepts of the logical-algebraic setting. On the basis of the new setting, we then propose, in Sect. 7, a first attempt at exploiting and learning from the outlined correspondence to deal in practice with consistency issues in the UML framework. We will use in the paper a simple UML model as running example, presented in Sect. 2.

Here, by UML we intend UML 1.3 as presented by the official standard specification version 1.3, [37]; while [34] is the reference manual (also if for previous version 1.1) and [21] is a short introduction with examples. A warning for those acquainted with the current literature on the UML consistency: we do not intend here to pursue the so called translational approach, by which one converts the UML or parts of it to some semantically well-defined formalism and then handles consistency by translation. Our attempt is to stay at the UML level trying to individuate at that level the correspondence with a logical-algebraic setting. For a classical algebraic approach we mention, e.g., the one in [38], for a much more updated and comprehensive view see [3], and the CoFI initiative [30].

2 UML Model Example

The UML model presented in this section describes an abstract design of a system for handling the order invoicing in a company. This model includes a UML class diagram plus associated constraints introducing the elements used to model the invoicing system (Fig. 1), some methods defining some operations of the passive class Stock (Fig. 2), a statechart defining the behaviour of the instances of the active class Invoicer (Fig. 3), and a UML sequence diagram, describing how three objects cooperate to successfully invoicing an order (Fig. 4).



Class invariants

context S: Stock inv: S.qu >= 0

context I: Invoicer inv: I.stk.previous = {}

Operation pre/postconditions

getProduct(p:Product,q:Integer)

pre: self.quantity(p) >= q

post: self.quantity(p) = self.quantity@pre(p) - q

addProduct(p:Product,q:Integer)

post: self.quantity(p) = self.quantity@pre(p) + q

Fig. 1. UML Class Diagram with Constraints

3 Consistency in the UML: the Problem

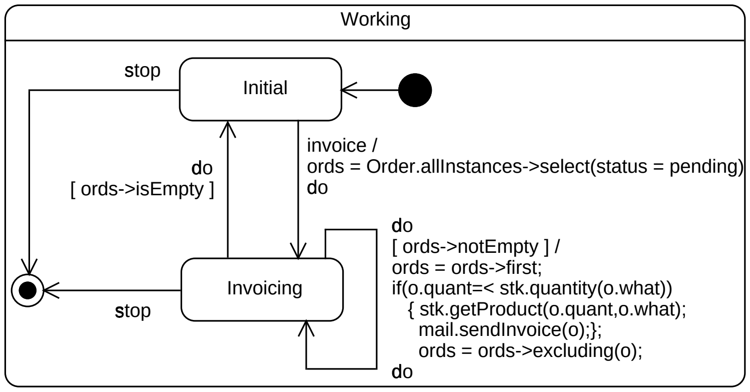
Consistency is a heavily overloaded word in the computing field, and used in a particular way in the UML community, thus we first try to clarify what it means.

A first informal approximation is in the following statement from [18]:

"During the development process artifacts representing different aspects of the system are produced. The artifacts should be properly related to each other in order to form a consistent *description of the developed system."*

```
getProduct(p:Product,q:Integer) method:
   if (self.pr  = p) {self.qu  = self.qu - q}
   elseif (self.others  <> {}) {self.others. getProduct(p,q)}
   else {null}
quantity(p:Product):Integer method:
   if (self.pr  = p) {return self.qu}
   elseif (self.others  <> {}) {return self.others.quantity(p)}
   else {return 0}
```

Fig. 2. Methods Associated with Operations of the Passive Class Stock



Working
stop
Initial
invoice /
ords = Order.allInstances->select(status = pending)
do
do
[ords->isEmpty]
do
[ords->notEmpty] /
ords = ords->first;
if(o.quant=< stk.quantity(o.what))
{ stk.getProduct(o.quant,o.what);
mail.sendInvoice(o);};
ords = ords->excluding(o);
do
Invoicing
stop

Fig. 3. Statechart Defining the Behaviour of the Active Class Invoicer

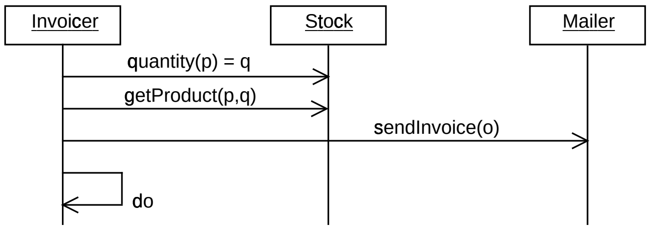


Fig. 4. Sequence Diagram Showing a Successful Order Invoice

There are two main reasons to have many different artifacts describing the same system:

multiview description techniques: at some level of abstraction a system is described as a collection of sub-descriptions dealing with different, possibly overlapping, aspects;

phased development process: the system is developed throughout different phases and iterations, each one producing a new more refined description of the system.

In the words of Engels et al. [18] *"Altogether the consistency conditions depend on the diagrams [constructs] involved, the development process employed, and the current stage of the development."*

A terminological clarification: in the world of the object-oriented software engineering a system description is called a *model* (e.g., UML models), whereas in the world of the formal methods, a system description is called a *specification* (e.g., logical-algebraic specifications). We will use the terminology consistent with the current use in any case, but the reader should recall that *model* and *specification* are essentially synonyms.

Notice that a single artifact produced using some notation may be unable to correspond to some coherent description of a system (for example, because it is ill-formed); in the UML world the word *inconsistent* is used also in these cases. However, we do not use the word inconsistent, as it is often done also in the SE literature (see, e.g., [18] where CSP processes ending in deadlock are considered to be inconsistent), to qualify descriptions which are well-defined, but describe senseless or useless systems (think for example, of an imperative program with a nonterminating loop or of a process ending in deadlock).

In the next two subsections we will illustrate in more detail how inconsistency may arise in connection with the two reasons mentioned above (3.1 and 3.2); while in subsection 3.3 we point out some aspects of the nature and the current status of the visual notation used, which may complicate and influence also the treatment of the consistency of a single diagram.

3.1 Multiview Descriptions

We speak of a multiview description of a system (or of a software artifact) whenever it consists of a collection of sub-descriptions (*views*) dealing with different, possibly overlapping, aspects of the system. For example, we can have the following views:

static view: the types of the entities building the system;
behaviour view: the behaviour of the various entities building the system;
interaction view: how the entities building the system interact among them;
...

The UML model of Sect. 2 shows a simple concrete case of a multiview description. Indeed, Fig. 1 is the static view, Fig. 2 and 3 are two behaviour views, and Fig. 4 is an interaction view.

Notice that a description/model/specification split in many different views is not just any description/model/specification modularly decomposed or structured; indeed in the second case the structure/decomposition may follows the structure of the described system; think, e.g., of a description of a distributed system split into the description of the composing processes.

Multiview models have nice advantages but also some problematic aspects. First of all, splitting a model of a system in several views allows to decompose it in chunks having a sensible size, and this is quite relevant in the case of visual models. Moreover, each single view focuses on a different aspect, and this is useful to analyze and to understand the various features of the modelled system. The splitting of a model into views allows also to split the work of producing such model among different persons and/or along the time. The less nice aspect of a multiview model is that the consistency of the overlapping submodels, corresponding to the various views, has to be guaranteed. The example of Sect. 2 shows many cases where the views are overlapping, and so where possible inconsistencies may arise, e.g.,

- the description of the class Invoicer in the static view of Fig. 1 (operations, attributes, constraints) must be coherent with the behaviour of the same class as presented in Fig. 3;
- the behaviour of class Invoicer depicted in Fig. 3 must be coherent with the role played by its instances in the collaboration depicted in the interaction view of Fig. 4;
- the method associated with the operation getProduct in Fig. 2 must be coherent with the pre/postconditions associated with the same operation in Fig. 1.

Another problematic aspect is that when a multiview model is refined to a less abstract one, each submodel cannot be refined independently; think for example, of refining an object of class Invoicer into several cooperating objects, in such case three views of the model must be refined in a coherent way.

3.2 Phased Development Process

In the software engineering world it is now well established that the development process of a software artifact should be organized in phases and that each phase is organized in various activities, where some of them may be iterated several times. Furthermore, many models[1] of such process have been proposed, each one characterized by its own phases and tasks to be done in each of them. Among the most important, of very different nature, from general guidelines to formal standards, we have

- the basic old one waterfall, characterized by four phases: capture and specification of the requirements, design, coding and maintenance;

[1] Here we have another occurrence of the word *model*, not to be confused with the UML models.

– the official standard of the German public administration V-Model;
– the most known one based on the UML RUP [31], quite heavy;
– the new Agile methods family (such as Extreme-Programming[2]).

Whatever software process development model we consider, the various phases and iterations require to produce different artifacts describing the system under development, which should be coherent among them; coherent means that they cannot present contradictory statements about an aspect of the system. In general such artifacts are related by particular relationships, such as realization, refinement, implementation, ...

Taking as example the trivial waterfall model, then the design must *realize* the requirements expressed by the requirement specification, and the code should be a correct *implementation* of the design specification.

We have also that in the UML world the models used in different phases may use different UML constructs (recall that UML has been defined by putting together many different notations). For example, we may have use case diagrams, sequence and collaboration diagrams for the requirements, and class diagrams plus statecharts and method definitions for the design.

3.3 The UML Notation

In the UML community the word *consistency* is used also w.r.t. a single diagram, with the general idea of the diagram being statically well-formed and having one precise meaning. These aspects are now quite standard and non-problematic for usual programming and specification languages (see, e.g., Java or SDL), and also in the case of a quite refined syntax and sophisticated semantics. Consider, for example, CASL [2] a logical-algebraic specification language that offers a powerful combination of ISO-Latin characters, subtyping, mix-fix operation syntax, and overloading to write quite readable textual specifications, and a sophisticated formal semantics for partiality, subsorting and architectural specifications.

But, UML is a visual language, and so its syntax is not (and cannot be) given by the standard techniques, such as BNF and abstract syntax trees or terms built by combinators. Thus even syntactic correctness does pose new and relevant problems. The chosen technique is to use *metamodelling* to present the abstract syntax plus the associated visual notation (concrete syntax) given apart for each abstract construct. Presenting the abstract syntax (the chapter of the official UML specification concerning the abstract syntax is strangely titled *Semantics*) in a metamodelling style means to give a (meta) class diagram, where the classes (or better metaclasses) correspond to the various constructs, their attributes and the relationships among them (including aggregation/composition) to the elements characterizing such constructs; and specialization has the obvious meaning. The conditions corresponding to the properties guaranteeing the static correctness (well-formedness) are given as constraints attached to such (meta)

[2] `http://www.ExtremeProgramming.org/`

class diagram. However, due to its dimension, the (meta) class diagram defining UML is split in several parts, each one concerning some particular construct (e.g., state machines) and presented in a separate chapter of the UML official specification [37] together with the relative constraints. Thus, the constraints consider always a unique construct (e.g., state machines: there is a unique initial state, there is no transition leaving a final state), and there are no constraints considering a whole model made by several constructs. So nothing is said about the mutual relationships among the constructs building a model (e.g., a call event appearing on a transition of a state machine must be built by an operation of the context class).

We present in Fig. 5 a simplified fragment of the UML metamodel showing some of the constructs needed to define a class diagram.

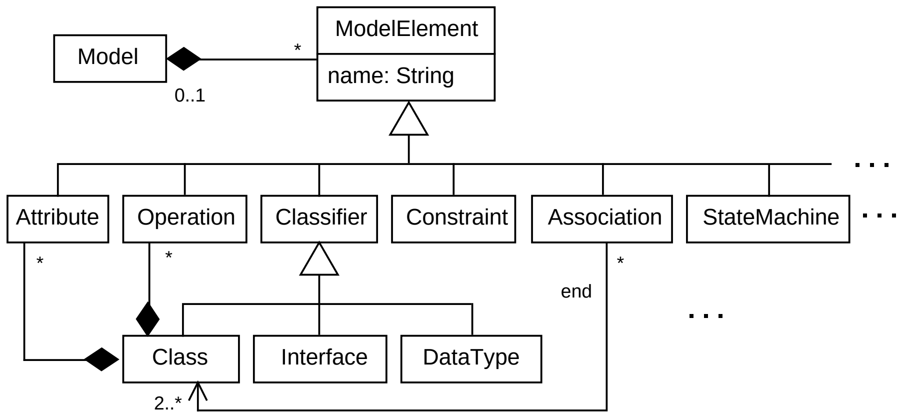


Fig. 5. UML Metamodel Simplified Fragment Concerning Class Diagram

The concrete syntax of UML, which is in the official standard [37] called *notation*, correspond to give many different visual diagrams each one corresponding to a subset of a model. Notice that, however, the relationship between the abstract and the concrete syntax is quite weak; for example the class diagram does not appear as a metaclass in the abstract syntax but only in the notation, and a unique model element (collaboration) corresponds to two different visual diagrams: sequence diagram and collaboration diagram.

UML aims to being the *unified* notations to model all the different aspects of a software system during the various development phases, and thus it offers an *abundance of constructs* originated in many different notations. Indeed, UML combines quite standard ways to present classes and associations, with more specific notations for the behavioural aspects (state machine, sequence diagram, collaboration diagram, activity diagram), with special notation for the use cases (use case diagram), with diagram for the physical/deployment view of the systems, and with constructs to structure the models (packages). We have, as nicely said in [18], a *“multitude of UML diagrams”*.

There is another aspect of UML that makes the treatment of its syntax problematic. Indeed, UML is not just a language to be used as it is in any case; instead to support its use in particular domains, or following particular development methods, or using particular applicative technologies UML offers already in itself mechanisms to define its own specializations (UML profiles). For example,

- stereotypes allow to define variants of the existing constructs,
- tagged values allow to add parts to existing constructs,
- additional constraints allow define restrictions on the subset of the UML which may be used (e.g., no visible attributes are allowed).

For what concerns the semantics (once called *dynamic* as opposed to *static*) of a UML model the situation is still worse. First of all the semantics is defined informally and in several points it is ambiguous, incomplete and even contradictory. Moreover, the existence of a variable semantics for some constructs is explicitly stated (*semantic variation points*); e.g., the policy for handling the event queues for the state machines is not fixed. Such feature allows to tune the notation to various possibly very different usages.

Finally, the same construct may be used for different purposes, even at different phases; for example, a state machine may describe the behaviour of an active or of a passive class, of an operation, of a use case, or of the whole system. In all these cases the semantics is slightly different.

4 Current Research Directions

Because of the problems outlined in the preceding section, a considerable amount of work has been devoted recently to the consistency issues in the UML. In particular, Engels et al. in a series of papers ([17–20]) have discussed the terminology, some technical and methodological aspects and references to other work. Moreover, a special workshop has been organized within the ≪UML≫ 2002 Conference devoted to “Consistency Problems in UML-based Software Development” (see [1]). An interesting discussion of the general issue of consistency in a more general setting than UML can be found in Derricik et al. [14].

As can be seen in the UML-related literature, usually the current terminology adopts a rather informal style, much different from the one used in the logical-algebraic field. Here too we use that style, to be contrasted later on, at least in part, with the one we propose on the basis of the analogy with the ADT viewpoint.

First we have to mention two orthogonal classifications of consistency, namely “Horizontal versus Vertical” and “Syntactic versus Semantic”.

Horizontal consistency, also named intra-consistency or intra-model consistency, is *“related to consistency among diagrams within a given model”* [1], typically within a development phase.

Vertical consistency, also named inter-consistency or inter-model consistency, is *“concerning consistency between different models”*, typically at different development phases [1]; the qualification “vertical” is referred to the process of refining models and requires the refined model be consistent with the one it refines [19].

Syntactic consistency "ensures that a specification conforms to the abstract syntax specified by the metamodel ... this requires that the overall model be well-formed" [19].

Semantic consistency

- *"with respect to horizontal consistency, requires models of different viewpoints to be semantically compatible with regards to the aspects of the system which are described in the submodels"* [18].
- *"with respect to vertical consistency, semantic consistency requires that a refined model be semantically consistent with the one it refines"* [18].

Finally and noteworthy, in [20] also *evolution consistency* is mentioned and addressed, namely consistency between different versions of the same submodel.

With reference to the aforementioned classification, let us briefly review some research directions.

As for static consistency, there are various attempts, but still a lack of methods for defining static semantics analogous/comparable to the methods centered around term structure and induction principles. As we have seen in the previous section, the static semantics is dealt with using metamodelling (class diagrams plus constraints). Thus the current work on static semantics is concerned with checking metamodelling constraints within a rule checking system, with various techniques. For example the xlinkit system, a generic tool for managing the consistency of distributed documents in XML format, is used and expanded to check consistency of XML documents in XMI format (Finkelstein et al. [23]). In [36] Surrouille and Caplat propose a transformation of OCL constraints into operational rules handled within a knowledge system. Sometimes, as it is typical of a good amount of UML-related work, there is not a a clear cut between syntax and semantics; in this line among other things the NEPTUNE project (Bodeveix et al. [10]) on one side extends OCL (also with temporal operators) and then provides tools for checking well-formedness of OCL rules at the application level and also satisfaction of OCL constraints defined at the metalevel.

As for semantic consistency, we can roughly distinguish three approaches.

In the transformational approach all viewpoints are translated to a common underlying semantic framework and deal with consistency there. The common underlying semantic framework is provided by

- an integration of Transition Systems, Algebraic Specifications and Transformation Rules (Grosse-Rhode [22]);
- Generalized Labelled Transition Systems (Reggio et al. [33], Bhaduri and Venkatesh [9]);
- RDS (Reactive System Design Support) (Lano et al. [26]);
- High Level Petri nets (Baresi and Pezzè [7]);
- Automata (Lilius [27]).

A related approach provides a common semantic model, but to derive consistency checks in the original specification; this is the case of the work of Davies and Crichton ([13]) that derives sets of allowable traces in an object diagram via the CSP semantic model and of Jurjens ([25]) that proposes a refinement for UML diagrams from a common ASM semantic model.

Another approach is by interpreting and testing; for example in [17] Engels et al. propose DMM, Dynamic Meta Modelling, a graphical version of SOS rules used to interpret UML models and to generate tests.

A most important point stressed in particular by Engels et al. [19, 17] is the following principle: in cases such as UML-based development, it is mandatory to provide a *methodological framework* for dealing with consistency issues. Thus we need development methods dealing with consistency, typically by restricting the use of notation to be consistent; for example this is included in the work of the Autofocus group (TUM [35]) and of Huzar et al. [24]. At the end of this paper we will show how consistency is addressed from that methodological viewpoint in our own work [5, 6]. In order to understand the complexity of the issue let us report the steps of a general frame for consistency management proposed by Engels at al. [19, 17]

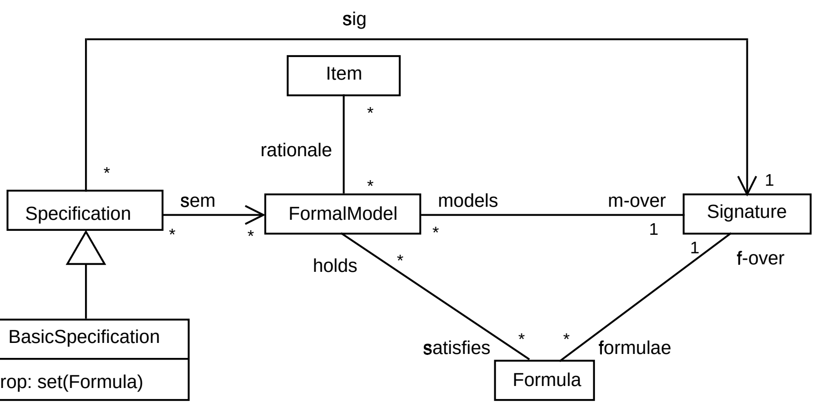
1. Identification of conceptual model, aspects, notation
2. Identification of consistency problems
3. Choice of a (common) semantic domain
4. Partial mapping of aspects leading to consistency problems
5. Specification of consistency conditions
6. Location and analysis of potential inconsistencies

5 The Logical-Algebraic Specification Case

5.1 The Logical-Algebraic Framework

The logical-algebraic framework may be presented in a very refined way using the categorical language and the concept of institution, see, e.g., [38, 3, 30]. Here, to make smoother the transition to UML and also to be understandable by the wider audience of UML specialists, we present it following the metamodelling style typical of the UML and the concept of specification method of [4]. Of course that means to remain at a more informal level and not to be concerned with more sophisticated issues, like the consistency with the change of signatures. In Fig. 6 we summarize the essential structure with the comments below, and [*exemplify them for the case of first-order specification of partial abstract data types, see [3]*]. Notice below the different terminology from the one of the UML: the UML models roughly correspond to the specifications here; while the (formal) models here correspond to the basic semantic structures.

- The Item are the specified elements [*abstract data structures/types*].
- The FormalModel are formal structures corresponding to the specified items [*many-sorted partial algebras*].



General

context F: Formula inv: F.holds.m-over = F.f-over
context FM: FormalModel inv: FM.satisfies.f-over = FM.m-over
context S: Signature inv: S.models.includesAll(S.formulae.holds)

Well-Formedness Rules

context BSP: BasicSpecification inv: BSP.prop.f-over = BSP.sig

Semantics

context SP: Specification inv: SP.sig.models.includesAll(SP.sem)
context BSP: BasicSpecification inv: BSP.sem = BSP.prop.holds

Fig. 6. Logical-Algebraic Specification Framework: Basic Specifications

– The association rationale describes how the formal models give an abstract and precise representation of the specified items [*it is easy to see how many-sorted algebras formally models data structure: carriers = sets of values, algebra functions = data operations and predicates*]; see [4] for more significative examples of rationales.
– The formal models are classified by their static structure, that in the logical-algebraic case is usually defined by a Signature; in general a signature is a list of the *constituent features* of the formal models [*many-sorted first-order signatures; in this case the constituent features are sorts, operations and predicates*].

  The association, whose association ends[3] are m-over and models links the formal models with the signatures describing their structure [*each many-sorted algebra is built over a many-sorted signature*].
– A Formula is a description of a property of interest about the formal models. More precisely, a formula describes a property that is sensible only for the formal models having a given structure represented by a signature; thus each formula is built over a signature (the association with ends f-over and formu-

[3] In the UML it is possibly to name differently the two ends of an association, each association end will be used to navigate in the direction towards the class to which it is placed near.

lae links signatures with the formulae built over them) [*first-order formulae over a many-sorted signature*].
- The association, whose ends are holds and satisfies, defines when a formula holds on a formal model/a formal model satisfies a formula. Clearly, this relationship is sensible only when the linked formal models and formulae are built over the same signature, see the constraints in Fig. 6 [*the usual interpretation of first-order formulae*].
- A Specification is characterized by a signature (sig) and by a set of formal models, its semantics (sem); such models are all over the signature of the specification (constraint SP.sig.models.includesAll(SP.sem)).
- A BasicSpecification is a specification that consists exactly of a signature and of a set of formulae over it, and determines the set of formal models satisfying all such formulae, constraint BSP.sem = BSP.prop.holds. The well-formedness constraint BSP.prop.f-over = BSP.sig requires that the formulae of a basic specification are built over its signature.

Typically, an algebraic specification language offers together with construct to present basic specifications several ways to structure complex specifications, each one given by a combinator which builds new specifications from existing ones. The most common combinators are

- sum or union ("SP1 + SP2" is the specification having all the constituent features and all the properties of SP1 and of SP2);
- reveal ("reveal SIG in SP" is the specification SP where only the constituent features present in the signature SIG are made visible/revealed);
- rename ("rename SP by ISOMORPH" is the specification SP where its constituent features, sorts, operations and predicates, are renamed as described by the signature isomorphism ISOMORPH).

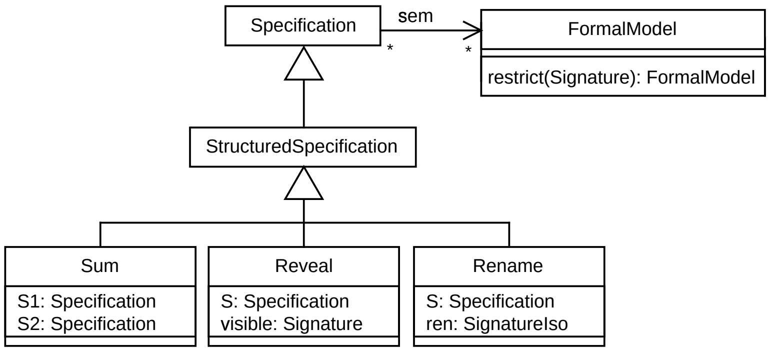
In Fig. 7 we give a fragment of the extension of the class diagram of Fig. 6 to include structured specifications.

5.2 Consistency in Logical-Algebraic Specifications

In the logical-algebraic setting the consistency problems are defined along the following lines, where we use some current terminology found in the literature about UML.

First of all we define the so-called *syntactic consistency* that is called in the formal methods/programming language community *syntactic/static correctness* or *static semantics*.

- A basic logical-algebraic specification is *syntactically consistent* whenever it consists of a set of well-formed formulae over its signature, determine by the association f-over of Fig. 6 (see the constraint BSP.prop.f-over = BSP.sig in the same figure).



General
context SP: Reveal inv: SP.sig= SP.visible
. . . . . .
Well-Formedness Rules
context SP: Reveal inv: SP.visible.isSub(SP.S.sig)
. . . . . .
Semantics
context SP: Reveal inv: SP.sem= SP.S.sem.restrict(SP.visible)
. . . . . .

Fig. 7. Logical-Algebraic Specification Framework: Structured Specifications Fragment

Usually in the algebraic setting the syntactically consistent basic specifications are defined in an inductive/constructive way, that is by defining directly by induction the set of all the correct formulae over a signature, and of all the correct (basic) specifications, instead of qualifying which elements in a larger set correspond to correct formulae, see, e.g., [38, 8] for inductive definition of first-order formulae. The syntactic consistency of structured specifications is handled in a similar way; here by means of some constraints on a class diagram, see, e.g., Fig. 7 for an example of such well-formedness rules; whereas in the algebraic setting again the correct structured specifications are defined in an inductive/ constructive way [8].

For the semantic consistency, also in the logical-algebraic world we have the distinction between consistency of one specification (or intra-specification consistency) and vertical consistency (or inter-specifications consistency) of one specification w.r.t. another one.

– A logical-algebraic specification SP is *(horizontally) semantically consistent* (standard terminology *consistent*) iff its semantics, defined by the association sem of Fig. 6, is not empty (SP.sem <> {}).

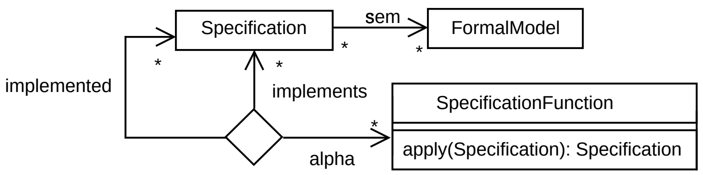
Horizontal semantic inconsistencies in the logical-algebraic case are due to the fact that a specification includes some formula that implies the negation of another of its formulae (including the case of a formula that implies its negation).

Notice that a semantically consistent specification is not always a sensible specification of some data structure. Consider, for example, the case of a specification whose all models are isomorphic to the trivial algebra (the one whose carries have exactly one element and the interpretation of operations and predicates is the obvious one); this is consistent, but *in most of the cases* it is not what the specifier intended. In the case of partial algebras, you can have also specifications where the unique model is the algebra whose carriers are empty sets. Unfortunately, it is not possible to define in general this kind of specifications, they depend on the particular setting, and cannot fully banned (sometimes they are really wanted, e.g., the specification of a token requires a unique sort with just one element). However, from the methodological point of view, it may be useful to define the class of such specifications, which we name *pseudo-inconsistent*, and perhaps to introduce techniques to detect them.

The problem of vertical consistency, concerning two algebraic specifications, one *implementing* the other, is handled in a very careful way. In this setting, we speak of vertical consistency relative to a given relationship between the structures of the two specifications due to Sannella and Wirsing (see [38]), given as a function mapping specifications into specifications. We summarize in Fig. 8 this notion of implementation, as a ternary association, where the end alpha shows the way a specification is implemented by the other one.

The presence of a semantics (association sem) allows to precisely define the notion of vertical consistency by the constraint in Fig. 8 stating that

"if SP is implemented by SP1, then the semantics of SP (a class of formal models) includes the semantics (another class of formal models) of α(SP1)"."



Semantics

context SP: Specification inv: SP.sem.includesAll(SP.alpha.apply(SP.implemented).sem)

Fig. 8. Logical-Algebraic Specification Framework: Implementation

When the specification function is a composition of a renaming, an extension with derived operations and predicates, a reveal and an extension with axioms, we have the so called implementation by rename-extend-restrict-identify of [15, 16]; which corresponds, within the framework of abstract data types, to the Hoare's idea of implementation of concrete data types.

6 The UML Framework

Here we try to provide a framework for the UML corresponding to the previous one for algebraic specifications. After some preliminary considerations we give a schematic view of the possible correspondence. Our presentation is sketchy in the sense that we concentrate on the basic underlying ideas; indeed a too detailed treatment here would be senseless, since the UML is evolving; the version considered here 1.3 [37][4] will be replaced by a rather different one, UML 2.0.

Here we consider UML at the level of the abstract syntax, which is as it is presented by the metamodel in the *Semantic* chapter of the official standard [37]; reference to the corresponding visual diagrams (as presented in the *Notation* chapter of [37]) will be added to help relate what we present with the current view of UML.

Recall that in the following we use the terminology of Sect. 5.1, along the schema of Fig. 6.

The UML constructs to structure models (specifications) is the package, thus UML models without packages (and their variants as subsystems) may be considered as basic specifications, following the terminology of the previous section.

In this section we will explore the analogy with the algebraic specifications and argue that the role of basic specifications can be played in the UML by the notion of *UML basic model.*

UML Item UML models are meant to describe real-world systems, as software systems, information systems, business organizations; and thus these are the elements of Item.

UML Signature In the UML, neither in the metamodel, nor in the associated notation, there is an obvious construct that may play the role of the signature in the algebraic specification case. However, if we look carefully at the various (model) elements building a UML model we may discover that many of such model elements just state which *entities* will be used in the UML model to describe the system (giving their name and their kind/type), for example classes, operations and attributes. We call *structural* such model elements.

Now, a UML model made only of structural model elements may be considered a kind of signature, since it defines the structure of the modelled system; such models will be called *signature diagrams.*

– A *structural model element* may be
  * a classifier of the following kinds: class (distinguished in active and passive), datatype, interface, use case, actor and signal; clearly it will be equipped with its own particular features (e.g., attributes, operations and signal receptions for classes), but without any form of semantic constraints (e.g., the isQuery annotation for an operation, or the specification part for a signal reception);
  * an association, but without any semantic attribute (e.g., multiplicity and changeability)[5];

[4] To be precise the last version is UML 1.5, but there are no big differences with 1.3.

[5] We do not include aggregation/composition in the signature diagrams, because essentially they are just a normal associations plus some constraints concerning the

  * a generalization relationship, but without any predefined constraint (only the subtype aspect matters at the structural level).

- A *signature diagram* is a UML model built only of structural model elements satisfying some well-formed constraints, as
  * all classes have different names,
  * all attributes of a class have different names,
  * all operations of a class have different names,
  * the type of an attribute is either an OCL type, or the name of a class appearing in the diagram,
  * ...

Here for simplicity we have expressed the well-formedness constraints on signature diagrams by using the natural language, but they may be expressed precisely as OCL formulae.

We report in Fig. 9 the signature diagram for our running example of UML model of Sect. 2. To help make clear the difference with the class diagram of Fig. 1 we have shadowed the parts not belonging to the signature diagram.

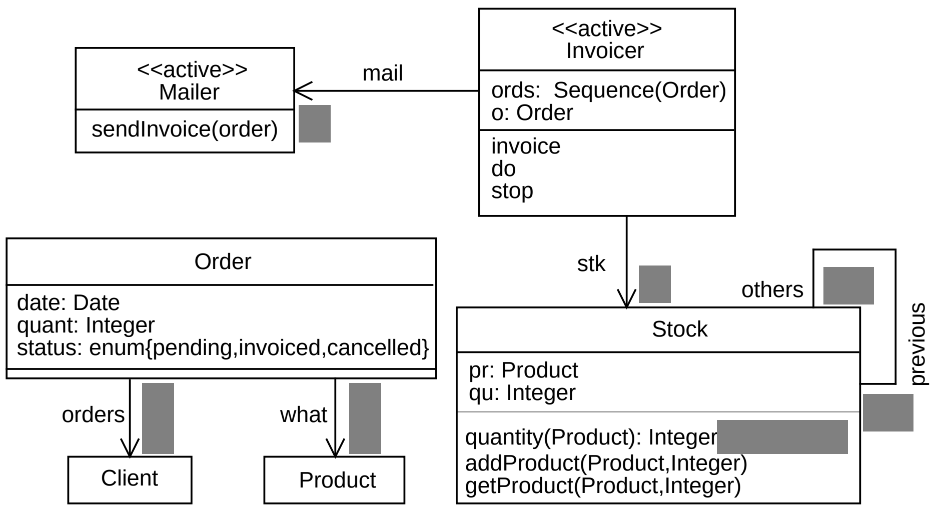


Fig. 9. The signature diagram of the example UML model

Notice that our signature diagram may be visualized as a particular class diagram, and that any development method based on the UML requires to provide at least a static view of the modelled system, which is a class diagram. Thus our proposal is not peculiar, but it just makes explicit the underlying splitting between the static/structural part of a UML model and the dynamic/behavioural/semantic part.

creation/termination of the aggregated/composed objects and those of their subparts.

It is possible to define our signature diagrams in a very precise way at the metamodel level; we just need to introduce a new metaclass corresponding to the signature diagrams, to slightly modify the metaclasses corresponding to the structural model elements (e.g., by dropping the multiplicity from the associations) and to redefine the metaclass corresponding to a complete model (now it is an aggregate of one signature diagram and of several semantic model elements).

UML FormalModel There is not a standard choice for the formal models for the UML, because no official formal semantics is available. Many, mostly partial, proposals may be found in the literature; the important point is that the chosen formal models should be structures able to accommodate all the aspects supported by the UML models.

Just for having one at hand to be used as reference for explanatory purposes, we mention our proposal of [32, 33], where we advocated, according to a line of work developed within the CoFI initiative [30], the use of what we call UML-formal systems. They are sufficiently general to be used also for UML 2.0 we believe.

- A *generalized labelled transition system* is a 4-uple

  $$(\mathit{STATE}, \mathit{LABEL}, \mathit{INFO}, \rightarrow),$$

  where STATE, LABEL and INFO are sets and

  $$\rightarrow \subseteq \mathit{INFO} \times \mathit{STATE} \times \mathit{LABEL} \times \mathit{STATE}.$$

  A transition $(i, s, l, s') \in \rightarrow$, represented as $i : s \xrightarrow{l} s'$, describes a possibility/capability of the modelled system to pass from a situation represented by s into another one represented by s', where l describes the interaction with the external world during such transition, and i some additional information on the transition (e.g., moves of the subcomponents).

- The *UML-formal systems* are a particular class of generalized labelled transition system (see [33]).

We have then to match each signature diagram with the corresponding UML-formal systems. This can be done quite simply, because the UML-formal systems of [33] include a description of the static structure of the UML model to which they give semantics. For example, they define the names available in such model and classify them in class names, attribute names, operation names and so on.

UML Formula Surprisingly or not, the UML constructs playing the role of formulae in our setting will be particular model elements (the non-structural ones) that state properties of the *entities* used in the modelled system. We name them *semantic model elements*, and list them below.

constraints including the implicit ones (i.e., those embedded in the definition of structural model elements, as association multiplicities and signal reception specifications);

methods in the UML official standard [37] they are considered as class features, but they just define the semantics of operations; thus they have no structural effect, but restrict the formal models to those where the interpretation of some operation matches the one described by the method itself;

state machines visually presented as statecharts, they fix the behaviour of the instances of classes, or of an operation, or of a use case; thus they restrict the formal models to those where the behaviour of such elements is the one described by the state machine.

collaborations visually presented as sequence or collaboration diagrams, impose restrictions on the possible interactions (exchange of messages, i.e., operation calling) among some objects used to model a system, and so they constrain the behavior of such objects.

activity graphs visually presented as activity diagrams, impose restrictions on the causal relationships among facts happening in the associated entity (an operation, the whole system, a part of the system, a use case, ...) and so they constrain the behavior of the mentioned entities.

Then, we need to define when a semantic model element is built (well-formed) over a signature diagram (association f-over of Fig. 6).

As we have pointed out repeatedly considering the current status of the UML, it is not worthwhile to present here in detail the well-formedness conditions for all constructs. Still it is worthwhile to illustrate that kind of statement w.r.t. some basic points and significant examples. Thus we single out the well-formedness of a state machine, as a paradigmatic case of UML semantic model element.

A state machine SM is well-formed over a signature diagram ΣD iff it fulfills the following rules

- Well-Formedness Rules from UML official specification (not depending on ΣD); e.g., *"A final state cannot have any outgoing transitions."*
- Well-Formedness Rules depending on ΣD, mainly about elements of the SM containing expressions/actions; e.g., for each transition of SM
  * the (OCL) expression of the guard is well-formed w.r.t. ΣD, the context class and the parameter list of the event, and has type Bool;
  * the event is well-formed w.r.t. ΣD and the context class, which depends on the kind of event:

    call event the operation is an operation of the context class,

    change event the (OCL) expression is well-formed w.r.t. ΣD and the context class, and has type Bool;

    ...
  * the action of the effect is well-formed w.r.t. ΣD, the context class and the parameter list of the event.

Notice that to check whether an OCL expression or an action is well-formed w.r.t. ΣD, a class and a possibly empty list of typed free variables, and to find the type of a correct expression are the basic ingredients to define not only when a state machine is well-formed, but also for the other semantic model element.

We want to point out that it is not advisable to follow the inductive/constructive style of the algebraic specifications to define the well-formed UML models. The reason is that UML is a visual notation whose constructs are in large part variants of graphs, and thus they cannot be naturally defined by means of combinators/constructors, which just support tree-like structuring. [6]

For what concerns the association holds/satisfies, [33] defines when a UML-formal system is in agreement with an OCL constraint and a state machine. We have made also some feasibility studies for what concerns collaborations and activity graphs, whose results confirm that it is possible to define when a UML formal system is in accord with one of these UML constructs.

The *formulae* in our running UML model are obviously the diagrams of Fig. 3 and 4, the methods of Fig. 2, the explicit constraints in Fig. 1 and the implicit ones in the class diagram in the same figure (as the association multiplicity).

UML BasicModel The basic specifications in the UML framework are the UML models without packages, which we name *UML basic model*; indeed, the package is the basic and unique UML construct to structure a model. A UML basic model may be rearranged to explicitly present a signature diagram and a set of semantic model elements. The semantics of a UML basic model (association sem) is, then, defined as for the algebraic specifications, that is the set of the UML formal systems over its signature diagram that *satisfies* all its semantic model elements.

Notice that a UML model without packages defined following the UML meta-model of [37] may be automatically transformed into the corresponding basic model, as already hinted before.

A structured UML model should be a UML model containing various packages related by many possible qualified associations, as import, access, and specialize. A careful investigation of the package construct is needed, to determine which are precisely the underlying structuring mechanisms. Moreover, the actual package concept has been the subject of many criticisms, and new way to intend them has been proposed to be introduced in the forthcoming UML 2.0, see, e.g., [11]. For these reasons we have postponed the treatment of the UML structured models.

[6] This aspect is common to any visual notation, not just to the UML. We followed the standard constructive style when presenting JTN [12] a Java targeted visual notation, but at the expense of quite hard labour and of very long and complex definitions.

7 Dealing with Consistency in the UML Framework

7.1 Defining Consistency

Using the various ingredients of the UML framework defined in Sect. 6 we can now define the various kinds of consistency in the same way as for the logical-algebraic case of Sect. 5.

In the algebraic case we have that a basic specification is syntactically consistent (statically correct) whenever its signature is well-formed and all its formulae are well-formed over such signature.

For the UML case we can state

- A UML basic model *UBM*, essentially a pair $(\Sigma D, \{SME_1, \ldots, SME_k\})$ with ΣD signature diagram and for $i = 1, \ldots, k$, SME_i semantic model element, is *syntactically consistent* (*well-formed/statically correct*) iff ΣD is well-formed and for $i = 1, \ldots, k$ SME_i is well-formed over ΣD(defined by the association f-over, see Sect. 6).

Notice that in this way we do not need to define for all kinds of semantic model elements when they are pairwise syntactically consistent, just as in the logical-algebraic framework, we never had to define which pair of logical formulae are mutually syntactically consistent. Thus the technique that we have proposed is quite modular/scalable; indeed our definition of syntactic consistency can easily extended whenever the UML is extended; for each extension you have just to enlarge/restrict the set of the structural elements and of the semantic model elements, and if new kind of elements are added, just define the new well-formedness conditions w.r.t. the signature diagram.

The horizontal semantic consistency of the UML basic models is defined as in the algebraic case.

- A UML basic model *UBM* is *semantically consistent* iff its semantics, a set of UML-formal systems, is not empty ($UBM.\mathsf{sem} <> \{\}$).

Notice that this definition, obviously, requires to have at hand a UML formal, or at least quite precise, semantics.

We have thus given the general precise definition, but now we have to analyse the UML models to look for the possible causes of semantic inconsistency. In the algebraic case the only causes for inconsistency are the presence in the specification either of an unsatisfiable formula or of two mutually contradictory formulae, although to check it in the general case is an undecidable problem. Instead, we find that in a UML model there are many different causes of inconsistency and of very different nature. Here, we list some of the most relevant ones:

- a pre/postcondition is in contradiction with a method definition for the same operation;
- a pre/postcondition is in contradiction with an activity graph associated with the same operation;

– a pre/postcondition is in contradiction with a state machine associated with the same operation;
– a precondition on an operation is in contradiction with a state machine or an activity graph including a call of such operation;
– an invariant constraint is in contradiction with a state machine for the same class;
– a collaboration including a role for class C is in contradiction with a state machine for class C.
– . . .

Notice, that several cases are quite subtle depending on which is the chosen semantics for the UML constructs; for example, it is quite hard to decide when two collaborations including a role for a class C are contradictory. If we assume that the semantics of a collaboration is to present a possible execution/life cycle/ . . . of the modelled system, then two collaborations will never be in contradiction.

The semantics of the UML plays a fundamental role to discover the possible kinds of inconsistency. Moreover, such semantics should also help express in a precise (also if not formal way) the reason for the possible inconsistencies.

Quite surprisingly, the actual semantics of pre/postconditions does not produce inconsistent models, but just pseudo-inconsistent ones (see Sect. 5.2). Recall that the semantics of the pre/postconditions associated with an operation of [37] is precisely intended as follows[7]:

"postcondition: a constraint that must be true at the completion of an operation."

"precondition: a constraint that must be true when an operation is invoked." Thus, with this semantics a pre/postcondition does not constrain the associated operation, but just its usage; for example, an operation that will be never called satisfies any pre/postcondition.

Some cases of pseudo-inconsistencies are:

– an unsatisfiable invariant constraint on class C (it holds on trivial UML-formal systems where there are no instances of class C);
– two invariant constraints are contradictory (as before);
– two preconditions (postconditions) for an operation are contradictory (it holds on trivial UML-formal systems where the operation will be never called);
– . . .

For what concerns the vertical semantic consistency between two UML models we think that the approach chosen in the logical-algebraic framework could be sensible also in this case. The problem, is that we have to define the UML correspondent of a *specification function* and to visually represent it, that is a convenient way to define transformations over UML models. Unfortunately, there is no a standard accepted proposal, but this is a hot topic in the UML community, also because of its importance within the MDA approach (Model Driven Architecture) [29].

[7] Notice that UML pre/postconditions are quite different from the Hoare's one (a postcondition is not related with any precondition).

7.2 Checking Semantic Consistency

The list of all possible causes for semantic inconsistencies in a basic UML model seems to be very long, and a very careful analysis is needed to complete it (see Sect. 7.1). Furthermore, each kind of inconsistency poses a different kind of technical problems, from classical satisfaction problems in the first-order logic or in the Hoare logic, to check whether a sequence is a possible path in a transition tree, or if a transition tree is in agreement with a partial order. As a consequence, a method for helping detect all possible inconsistencies is not feasible. Moreover, the semantics inconsistencies are based on the UML semantics that may change, e.g., in semantics variation point, or because we are using a UML profile for a particular development method, or for a particular phase of the development, or for a particular application domain. On the other side, a development method based on the UML in general uses only models having a particular form (for example, some construct can be never used, or used only in a particular context, or used only in particular form, e.g., state machines only associated with active classes).

Thus, to survive with (horizontal) semantic inconsistencies we propose to design development methods based on the UML with the following characteristics.

- First of all, the method should require to produce UML models with a precise syntactic structure. Such structure must also guarantee the syntactical consistency of those models.
  For example, the method may require that
  - * for each operation of a class there is either a pre/postcondition or a method definition, but not both;
  - * a use case is complemented by a set of sequence diagrams, and that any of them represents an alternative scenario;
  - * at most one invariant is associated with each class and at most one pre/postcondition is associated with an operation.

  The problem of checking whether a model has the required syntactic form should be computable, and thus it should be possible to develop tools to perform such check; for example, tools for evaluating OCL formulae or tools based on XML technologies, or for evaluating conditional rules.
- The intended (formal) semantics of the UML models produced following the method should be defined.
- The UML models having the particular form required by the method and the chosen semantics should be analysed w.r.t. consistency.
  Thus, it should be possible to factorize the checking of the consistency into a precise list of subproblems. If the possible causes of inconsistency are too many or too subtle, perhaps the method needs to be revised, by making more stringent the structure of the produced models, or perhaps the troubles are due to the chosen semantics.
- For each inconsistency subproblem detected in the previous activity,
  - * it should be described rigorously for the developers in terms of UML constructs without formalities [if you know it, then you can avoid it].

* guidelines helping detect its occurrences should be developed, using the available techniques, such as automatic tools, inspection techniques, check lists, sufficient static conditions, ...

As an experiment, we have applied the approach proposed in this subsection to analyse a method for requirement specification based on UML, first introduced in [5], and presented in a more detailed way in [6]. Such method requires that the UML models presenting the requirements to have a precise form, and to be, obviously, statically consistent. The proposed approach seems to be quite effective in this case; indeed, the possible causes of semantical inconsistencies have been found explicitly, and they are not too many, because such method requires to produce UML models having a very tight structure, and makes precise the semantics of any used constructs. For what concerns the support to detect the various possible inconsistencies we are extending the tool ArgoUML[8]. with many new critiques, each one signalling either an inconsistency (for example, those concerning the static aspects) or a possible cause of inconsistency (e.g., a postcondition for an operation of a class with an invariant on the same class) with an explanation of the reason. One of the most problematic point, concerning semantic consistency, is to check whether a state machine is in contradiction with a sequence diagram; to help this check we are trying to use a prototyping tool [28].

The above experiment shown the fundamental importance of defining a precise explicit semantics of the used UML constructs. For example, in the method a state machine is used to define the behaviour of each use case, but its semantics is different from the original one. Indeed, following the new semantics a state machine describes A SET of the possible lives of the instances of the context class, whereas the original semantics states that a state machines describe ALL such lives. Thus, two state machines with the new semantics can never be in contradiction.

8 Conclusions

Consistency is a really big problem in practical software development, where scale, heterogeneity and methods do matter. The case of UML-based software development is a good example of the issues that may arise.

Here, contrary to the vast majority of the current literature on the subject, we have taken an unorthodox approach, starting from the experience we have gained in many years of involvement with the logical-algebraic techniques. As an exploratory experiment, we have presented the problem and then tried to propose a framework for handling the consistency problems of the UML inspired by the framework of the logical-algebraic specifications. Admittedly at first sight the proposal may look naive; perhaps most people will be surprised by seeing a state machine playing the role of a formula; and indeed one may wonder what

[8] `http://argouml.tigris.org/`

can be the benefit of that analogy. But, if we exploit that analogy to build a UML framework for consistency, then we can forget the terminology (formulae, formal models, signatures, etc.) and handle the consistency issues knowing exactly what should be done and thus also what we are not able to do; for example, because we do not have at hand a precise semantics. In particular we have a setting where to locate, with merits and limits, the proposed approaches.

Our analysis is preliminary and incomplete, especially for what concern vertical consistency. Still, we believe that it shows the feasibility of the following, even for the new UML versions to come:

- a precise and sensible way to define the various kinds of inconsistencies (static, semantic intra-model, semantic inter-models);
- a workable method for detecting the static inconsistency (in other communities just known under the name of static correctness), that is quite modular and scalable;
- a possible methodological approach to cope with semantic intra-model inconsistency, especially with approaches that are, in our own terminology, "well-founded" and use "tight structuring".

As aside remarks, we believe that some ADT concepts and frames are still useful to provide clarification and practical guidance; but also new problems are appearing, such as the syntax presentation of visual multiview notations, the aspect currently handled in the UML by metamodelling.

As already pointed out before, there are two aspects not treated at all in our analysis, namely structuring and vertical consistency. For the first it seems more sensible to wait for a new version of the UML. As for the second, a link has to be established with the so-called MDA, Model Driven Architecture approach [29] proposed by OMG to developing software, where the relevant relationships between different UML models will be singled out; for example, those from the so called Platform Independent Models and the corresponding Platform Specific Models.

References

1. Consistency Problems in UML-based Software Development: Workshop Materials. In L. Kuzniarz, G. Reggio, J.L. Sourrouille, and Z. Huzar, editors, *Consistency Problems in UML-based Software Development: Workshop Materials*, Research Report 2002-06, 2002. Available at `http://www.ipd.bth.se/uml2002/RR-2002-06.pdf`.
2. E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. D. Mosses, D. Sannella, and A. Tarlecki. CASL : the Common Algebraic Specification Language. *T.C.S.*, 286(2), 2002.
3. E. Astesiano, B. Krieg-Brückner, and H.-J. Kreowski, editors. *IFIP WG 1.3 Book on Algebraic Foundations of System Specification.* Springer Verlag, 1999.
4. E. Astesiano and G. Reggio. Formalism and Method. *T.C.S.*, 236(1,2), 2000.
5. E. Astesiano and G. Reggio. Knowledge Structuring and Representation in Requirement Specification. In *Proc. SEKE 2002.* ACM Press, 2002. Available at `ftp://ftp.disi.unige.it/person/ReggioG/AstesianoReggio02a.pdf`.

6. E. Astesiano and G. Reggio. Tight Structuring for Precise UML-based Requirement Specifications: Complete Version. Technical Report DISI–TR–03–06, DISI, Università di Genova, Italy, 2003. Available at `ftp://ftp.disi.unige.it/person/ReggioG/AstesianoReggio03c.pdf`.
7. L. Baresi and M Pezzè. Improving UML with Petri Nets. *ENTCS*, 44(4), 2001.
8. H. Baumeister, M. Cerioli, A. Haxthausen, T. Mossakowski, P. Mosses, D. Sannella, and A. Tarlecki. Formal Methods '99 - Casl , The Common Algebraic Specification Language - Semantics, 1999. Available on compact disc published by Springer-Verlag.
9. P. Bhaduri and R. Venkatesh. Formal Consistency of Models in Multi-View Modelling. In [1].
10. J.-P. Bodeveix, T. Millan, C. Percebois, C. Le Camus, P. Bazes, and L. Ferraud. Extending OCL for Verifying UML Model Consistency. In [1].
11. T. Clark, A. Evans, and S. Kent. A Metamodel for Package Extension with Renaming. In J.-M. Jezequel, H. Hussmann, and S. Cook, editors, *Proc. UML'2002*, number 2460 in LNCS. Springer Verlag, Berlin, 2002.
12. E. Coscia and G. Reggio. JTN: A Java-targeted Graphic Formal Notation for Reactive and Concurrent Systems. In Finance J.-P., editor, *Proc. FASE 99*, number 1577 in LNCS. Springer Verlag, Berlin, 1999.
13. J. Davies and C. Crichton. Concurrency and Refinement in the UML. *ENTCS*, 70(3), 2002.
14. J. Derrick, D. Akehurst, and E. Boiten. A Framework for UML Consistency. In [1].
15. H.D. Ehrich. On the Realization and Implementation. In *Proc. MFCS'81*, number 118 in LNCS, pages 271–280. Springer Verlag, Berlin, 1981.
16. H. Ehrig, H.J. Kreowski, B. Mahr, and P. Padawitz. Algebraic Implementation of Abstract Data Types. *T.C.S.*, 20, 1982.
17. G. Engels, J.H. Hausmann, R. Heckel, and S. Sauer. Testing the Consistency of Dynamic UML Diagrams. In *Proceedings of IDPT 2002*, 2002.
18. G. Engels, J.M. Kuester, and L. Groenewegen. Consistent Interaction of Software Components. In *Proceedings of IDPT 2002*, 2002.
19. G. Engels, J. M. Kuster, L. Groenewegen, and R. Heckel. A Methodology for Specifying and Analyzing Consistency of Object-Oriented Behavioral Models. In V. Gruhn, editor, *Proceedings of the 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9)*. ACM Press, 2001.
20. G. Engels, J. M. Kuster, and R. Heckel. Towards Consistency-Preserving Model Evolution. In *Proceedings ICSE Workshop on Model Evolution, Florida, USA, May 2002*, 2002.
21. M. Fowler and K. Scott. *UML Distilled: Second Edition*. Object Technology Series. Addison-Wesley, 2001.
22. M. Grosse-Rhode. Integrating Semantics for Object-Oriented System Models. In F. Orejas, P. G. Spirakis, and J. van Leeuwen, editors, *Proceeding of ICALP'01*, number 2076 in LNCS. Springer Verlag, Berlin, 2001.
23. C. Gryce, A. Finkelstein, and C. Nentwich. Lightweight Checking for UML Based Software Development. In [1].
24. B. Hnatkowska, Z. Huzar, L. Kuzniarz, and L. Tuzinkiewicz. A Systematic Approach to Consistency within UML Based Software Development Process. In [1].
25. J. Jurjens. Formal Semantics for Interacting UML subsystems. In B. Jacobs and A. Rensink, editors, *FMOODS 2002*, volume 209 of *IFIP Conference Proceedings*. Kluwer, 2002.

26. K. Lano, D. Clark, and K. Androutsopoulos. Formalising Inter-model Consistency of the UML. In [1].
27. X. Li and J. Lillius. Timing Analysis of UML Sequence Diagram. In R. France and B. Rumpe, editors, *Proc. UML'99*, number 1723 in LNCS. Springer Verlag, Berlin, 1999.
28. V. Mascardi, G. Reggio, E. Astesiano, and M. Martelli. From Requirement Specification to Prototype Execution: a Combination of Multiview Use-Case Driven Methods and Agent-Oriented Techniques. In *Proc. SEKE 2003*. ACM Press, 2003. To appear.
29. OMG Architecture Board MDA Drafting Team. Model Driven Architecture (MDA). Available at `http://cgi.omg.org/docs/ormsc/01-07-01.pdf`, 2001.
30. P.D. Mosses. CoFI: The Common Framework Initiative for Algebraic Specification and Development. In M. Bidoit and M. Dauchet, editors, *Proc. TAPSOFT '97*, number 1214 in LNCS. Springer Verlag, Berlin, 1997.
31. Rational. Rational Unified Process© for System Engineering SE 1.0. 2001.
32. G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analysing UML Active Classes and Associated State Machines – A Lightweight Formal Approach. In T. Maibaum, editor, *Proc. FASE 2000*, number 1783 in LNCS. Springer Verlag, Berlin, 2000.
33. G. Reggio, M. Cerioli, and E. Astesiano. Towards a Rigorous Semantics of UML Supporting its Multiview Approach. In H. Hussmann, editor, *Proc. FASE 2001*, number 2029 in LNCS. Springer Verlag, Berlin, 2001.
34. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley, 1999.
35. B. Schatz, P. Braun, F. Huber, and A. Wisspeintner. Consistency in Model-Based Development. In *Proc. of ECBS'03*. IEEE Computer Society, Los Alamitos, CA, 2003.
36. J. L. Sourroille and G. Caplat. Checking UML Model Consistency. In [1].
37. UML Revision Task Force. *OMG UML Specification 1.3*, 2000. Available at `http://www.omg.org/docs/formal/00-03-01.pdf`.
38. M. Wirsing. Algebraic Specifications. In J. van Leeuwen, editor, *Handbook of Theoret. Comput. Sci.*, volume B. Elsevier, 1990.