Towards a Well-Founded UML-based Development Method*

Egidio Astesiano - Gianna Reggio
DISI, Università di Genova - Italy
{astes,reggio}@disi.unige.it

Abstract

This paper presents an attempt, perhaps unorthodox, at bridging the gap between the use of formal techniques and the current software engineering practices. After years of full immersion in the development and use of formal techniques, we have been led to suggest a Virtuous Cycle *philosophy, better marrying the rigour of formalities to the needs and, why not, the wisdom of current practices. What we have called* Well-Founded Software Development Methods *is a strategy compliant with that philosophy, that essentially aims at proposing methods where the formalities provide the foundational rigour, and perhaps may inspire new techniques, but are kept hidden from the user.*

In a stream of papers we have outlined an approach – a possible instantiation of a particular well-founded method – which is Model-Driven and adopts a UML notation. Here, after introducing our basic philosophy and the Well-Founded methods strategy, we outline in summary our sample approach and, as a new contribution, we show in some detail how to handle the Model-Driven Design (or Platform Independent Design) phase.

1 Introduction

The origin of the work presented in this talk goes back to the years when we have been involved in some projects with industry to apply our formal techniques to real-size case studies. It was soon evident that our work in formal methods had not taken into consideration some aspects of paramount importance in software development, generally qualifiable as methodological aspects. That has led us first to take care in some way of those aspects in the application of our techniques, and then to reflect on our approach. The results of our reflections were first presented in an invited talk at the last TAPSOFT in Lille in '97 (for a journal version, see [1]). The main findings of that investigation, significantly titled *Formalism and Method*, were

- the distinction between formalism and method,
- the inclusion of a formalism, if any, as a part of a method,
- the essential relevance of *modelling rationale* as the link between the end product (application, system) and the formal structures (formal models) representing it,
- the role of pragmatics.

We were arguing that the impact of formalisms would much benefit from the habit of systematically and carefully relating formalisms to methods and to the engineering context.

More generally, we have personally learnt how much it is advisable for people working in the area of the formal techniques to put the work into the perspective of what is considered essential for the development of software. There has been a clear lesson for us from our experience in projects with industry and has been a different attitude in the way we devise, use and advocate formal techniques, see [6]. We summarize our position in what may be called a *Virtuous Cycle*:

- Inspect and learn from software engineering practices and problems
- Look for/provide formal foundations when needed
- Experiment near-to practice but well-founded methods, hiding formalities
- Anticipate needs providing sound engineering concepts and methods

Clearly the above four directions are in a somewhat increasing order of difficulty and ingenuity. We have already done some work along the first three (the fourth being quite ambitious and to be judged by history and success). In this talk we will outline an experiment in what we call *Well-Founded Software Development Methods*; by that we roughly mean a revisitation or possibly a proactive proposal of engineering best practice methods, but with the guarantee that the notation is amenable to a rigorous formal foundation, though such formalization is not apparent to the user.

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

The experiment that we discuss here falls within a line of research that consists in looking at current development practices, noticing problems and attempting at a reformulation based upon, and inspired by, related work in formal techniques.

One of the foremost contributions coming from the software engineering side is the concept and use of development process models to guide the software development. We can take as a paradigmatic example, among the best well-known process models, the Rational Unified Process (RUP), proposed by the same authors of the UML (see [14]) and incorporating many insights coming from the software engineering best practices. The problems that we have encountered with RUP are twofold. On one side it relies on the UML as a supporting notation, which admittedly does not have a rigorous (neither static nor dynamic) semantics. On the other, to be liberal and accommodate, at least nominally, a number of variants, subcases and personal tastes, RUP gives so much freedom that a non-experienced user is often disconcerted among the possible modelling choices. These two kinds of problems have as a consequence that the resulting artifacts are much more prone to ambiguities, inconsistencies and the like. We have undertaken some work attempting at proposing a more stringent method, which we are in part experimenting in course projects. For example, in [3, 5] we have presented a new way of structuring the Requirement Specification and in [5] we have investigated the link between the analysis of the Problem Domain and the Requirement Capture and Specification. Our approach can be seen pursued within an overall RUP-compatible approach, with the aim of guiding the developer to

- use only semantically sound constructs,
- have better means for making the modelling decisions,
- produce as a result a set of artifacts under tighter constraints and as an overall result, to make the process faster, cutting sometimes endless discussions, and to better support consistency both in the construction and in the checking phase.

Though we have expressed our approach in a rigorous multiview, use-case driven and UML-based way, its essence is UML-independent and it could be even given a formal algebraic dress.

Before giving some technical highlights, let us mention the inspiring sources and the technical basis. First, the choice of a restricted subset of the UML constructs has been guided by a formal semantic analysis of those constructs. The general approach to address the semantic problems of UML, together with references to our work on more specific aspects, can be found in [16]. Essentially, it shows how the (generalized) labelled transition systems of our LTL approach [2] can be taken as a basis for defining what we call UML-systems as formal semantic models for all UML. Notably, that work has been pursued within the CoFI initiative [1] [13]. Second, we have incorporated some ideas from well-known methods, such as Structured Analysis [20] and the work of some pioneer methodologists such as M. Jackson [10, 11]. From the latter in particular we have taken the total separation of the domain from the application, a distinction somewhat blurred in many object-oriented approaches; while the distinction between the application and the environment especially comes from the Structured Analysis. Finally, from the overall formal (algebraic) approach, together with the strong typing as a must, we have also borrowed the idea of the black box abstraction of a software application and of its minimal white box structure to express the requirements about its interaction with the environment. To that end we have introduced the notion of "abstract state" for the software application, without providing an object-oriented structuring at a stage when such a structure is not required.

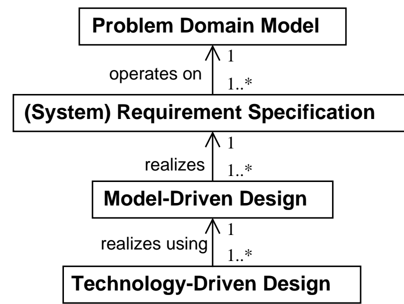

Figure 1. Artifacts

In this paper we first outline the overall structure of our approach and then we continue our investigation addressing the issue of the so-called Model-Driven Design. That is the stage between the requirement specification and the concrete design, when a first software architecture is laid down, which is totally platform independent, following the MDA terminology (see [12]). At the end we discuss our approach, especially in relation to the context of the current work on the subject.

2 Method Overview

In this section we outline the essential steps, with the corresponding artifacts to be produced, see Fig. 1, in our multiview, use-case driven and UML-based software development process. We intend the Requirement Specification activity built over the Problem Domain Modelling and preliminary to Model-Driven Design, followed by Technology-Driven design.

We speak of a multiview description of a system (or of a software application) whenever it consists of a collection

[1] `http://www.cofi.info/`

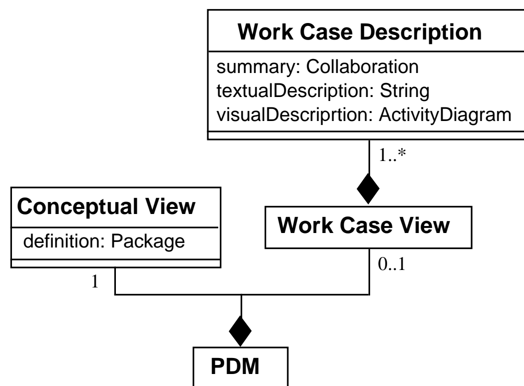

Figure 2. PDM **Structure**

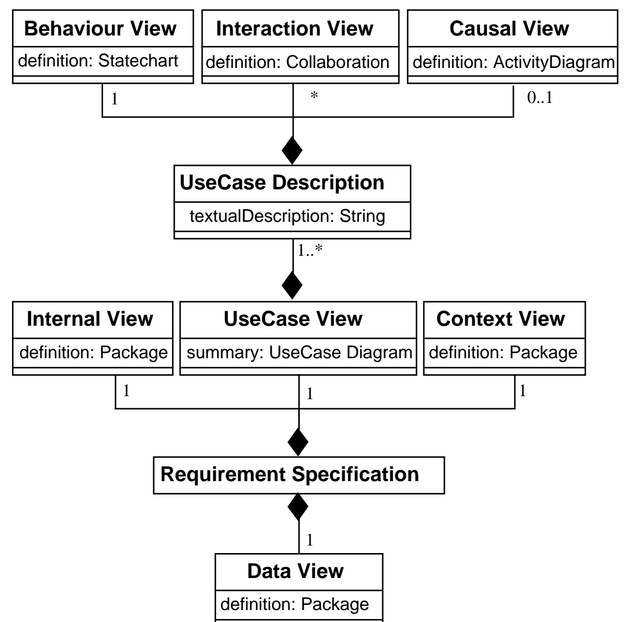

Figure 3. Requirement Specification **Structure**

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).

Notice that a description/specification split in many different views is not just any description/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.

Let us assume to have to develop a software application that we name in the following Application.

Problem Domain Model One of our assumptions, backed by our own experience, is the neat separation between the problem domain and the application (as much advocated in the work of pioneers, such as M. Jackson's [10]). To that purpose we propose a rather new way of structuring the problem domain model (shortly PDM), shown in Fig. 2 by a UML class diagram, and then the link with the application.

Our proposal, centered on two views, the Conceptual View and the Work Case View, in a structural sense encompasses the two most popular current approaches to domain modelling, namely conceptual modelling and business modelling, and can be reduced as a specialization to each of those. Then we propose an "application placement" activity, supported by a Application Placement Diagram, to relate the Application to the domain and, by that, to locate the Application boundary.

Some examples of PDMs can be found in [5].

Notice, that a PDM artifact may be used as a starting point for many different applications, as well as a Requirement Specification may be used for many different Model-Driven Designs, which in turn may be used to get many different Technology-Driven Designs (see the associations in Fig. 1).

Requirement Specification In our approach the Requirement Specification artifacts consist of different views of the Application, plus a part, Data View, needed to give a rigorous description of such views. Its structure is shown in Fig. 3.

Context View describes the context of the Application, that is which entities (*context entities*) and of which kinds may interact with the Application, and in which way they can do that. Such entities are further classified into those taking advantage of the Application (*service users*), and into those cooperating to accomplish the Application aims (*service providers*). That explicit splitting between the Application and the context entities should help avoid confusions between what exists and needs just to be precisely described (context entities) and what instead has to be developed (Application) on which we have to find (capture) the requirements The further splitting between users and providers should help distinguish which context entities cannot be modified by the developer (providers), and those which may be partly tuned by the developer (users), e.g., by fixing their interface towards the Application.

Use Case View, as it is now standard, shows the main ways to use the Application (*use cases*), making clear which actors take parts in them. Such actors are just *roles* (*generic instances*) for some context entities depicted in the Context View.

Internal View describes abstractly the internal structure

of the Application, that is essentially its Abstract State. It will help precisely describe the behaviour of the use cases, by allowing to express how they read and update it. UML allows a single use case to have a proper state, but we prefer to have a unique state for all the use cases, to help model their mutual relationships (e.g., if two use cases update the same information, we are led to detect and to handle possible conflicts).

Data View lists and makes precise all data appearing in the various views of the Application to help guarantee the consistency of the concepts used in such views.

Some of the above views (e.g., Internal View and Context View) are new w.r.t. the current methods for the OO UML-based specification of requirements. In our approach, they play a fundamental role to help ensure the consistency among the various use cases and of the whole specification.

Some examples of Requirement Specifications can be found in [5].

Model-Driven Design As currently widely advocated, by Model-Driven Design we intend the activity of providing a solution of the problem in terms of Model-Driven Architecture (see, e.g., [12]), namely an architecture based on the abstract modelling and independent of the implementation platform, to which is instead targeted the Technology-Driven Design.

In Sect. 4, also with the help of a running example, the algebraic lottery AL⊥ presented in Fig. 4, we will illustrate how we handle that step in our approach. The AL⊥ case study has been used also in [5], which contains also the relative PDM and Requirement Specification artifacts.

3 The Used UML

All the UML models produced following our method in the various phases of the development process must be prepared using only a specific subset of UML (to be precise of UML 1.3 [18]) trying to avoid its most problematic features, and the dark sides concerning semantics, both static and dynamic. Below, we briefly summarize what is included in this subset, and how we intend its semantics.

Class diagram containing classes, with attributes, operations and methods, specialization, aggregation and composition relationships, and user defined binary associations. Moreover, invariant constraints may be associated with classes, and pre-postconditions with operations.

Statechart without do actions and deferred events, and whose context class is active, has no methods, and its attributes are protected; for what concerns the semantics we assume that event queue is an ordered list.

We have to develop an application AL⊥ to handle algebraic lotteries. Our lotteries are said "algebraic" since the tickets are numbered by integer numbers, the winners are determined by means of an order *over such numbers, and a client buys a ticket by selecting its number. Whenever a client buys a ticket, he gets the right to another free ticket, which will be given at some future time, fully depending on the lottery manager decision. The number of a free ticket is generated by the set of the numbers of the already assigned tickets following some* law.
Thus a lottery is characterized by an order *over the integers determining the winners and a* law *for generating the numbers of the free tickets. To guarantee the clients of the fairness of the lottery, the order and the law, expressed rigorously with algebraic techniques, are registered by a lawyer before the start of any lottery.*
The application will be then realized as an on-line application, where the tickets must be bought and paid on-line using credit cards with the help of an external service handling them. Possible clients must register with the lottery application to play; and clients access the application in a session-like way. An external service takes care of the authentication of the clients.

Figure 4. The AL⊥ case study

Sequence/collaboration diagram we assume that the semantics of a sequence/collaboration diagram corresponds to describe a fragment of a possible execution of the modelled system.

Activity diagram only action states, decision nodes and synchronization states; we assume that the semantics of an activity diagram correspond to describe a partial ordering on the happening of some facts (executions of actions) in the modelled system.

Use case diagram without relationship between use cases.

Moreover, the expressions, the conditions and the constraints in any diagram are expressed by using OCL, the "logical language" for expressing the constraints of the UML, and the "actions" are the basic one required by the UML (operation call, creation and deletion of objects) plus assignment and generic control flow statements (conditional and while-loop).

All the produced UML models must be statically correct, or, using a UML-community terminology, statically consistent; see [4] for our proposal on how to rigorously define static consistency. Briefly summarizing,

a class diagram is statically consistent iff all the types used for operations and attributes are either defined in

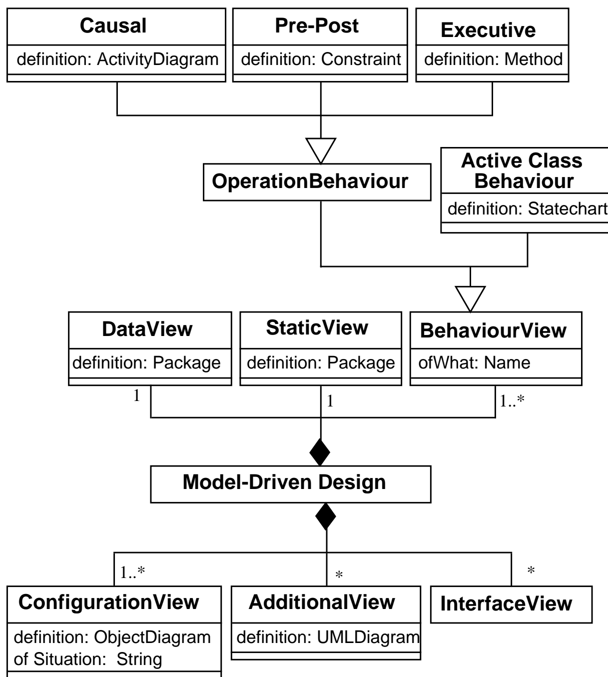

Figure 5. Model-Driven Design **Structure**

the diagram itself or are OCL predefined types, specialization relationship has no loop and the constraints are made of OCL expressions that are correct w.r.t. the considered class diagram itself.

a statechart is statically consistent iff all call events are built using operations of the context class, all conditions and effects are made respectively by OCL expressions and by actions that are correct w.r.t. the context class.

a sequence/collaboration diagram is statically consistent iff the roles refer to classes defined in the model, and all the messages are built by using operations, with correct arguments, of the class of the receiving role.

an activity diagram is statically consistent iff the action states and the conditions are respectively OCL expressions and actions correct w.r.t. the classes defined in the model.

It is then possible to give a formal semantics to the considered UML subset following, for example, our approach, see [16, 15], using (generalized) labelled transition systems as formal models of the systems described by a UML model.

4 Model-Driven Design

A Model-Driven Design, whose structure is reported in Fig. 5, consists of different views of the designed Application. The possible kinds of these views are listed below:

Data View describes the datatypes used by the entities composing the Application.

Static View introduces the classes typing the entities used to build the Application.

Behaviour View describes the behaviour of the entities of some given class; clearly, that class must be introduced in the Static View.

Configuration View describes which are the entities composing the Application in some given situation (e.g., initially), by stating which are their classes, how many they are, and how they are linked.

Additional View highlights an aspect of the Application concerning how some composing entities interact among them to accomplish some particular task. An additional view is optional and intended just for documentation; it should not add any information not already present in the other views.

Interface View describes the interface of the Application towards some context entities by presenting a GUI. Because there is not a standard established and convincing way to present GUI by using UML, this view is not a UML model, but just a document whose type is not fixed.

4.1 Data View

The Data View defines all datatypes used by the entities composing the Application.

Technically, the Data View is a UML package containing a class diagram, where all the classes are datatypes (UML stereotype ≪datatype≫ [2]) and where the relationships among classes are either specialization or aggregation or composition. An operation of a datatype may be defined either by pre-postconditions or by an associated method.

We show in Fig. 6 the Data View for our Model-Driven Design of AL_L; here to improve readability we do not detail some trivial data structures, PersonData and CreditCardData, and some trivial constraints are presented using the natural language.

4.2 Static View

The Static View introduces the classes typing the entities used to build the Application, which are of the following four different kinds:

[2] A UML datatype is a classifier whose instances are pure values, i.e., they have no identity and their state cannot be changed; thus the operation of a datatype operations are all pure functions.

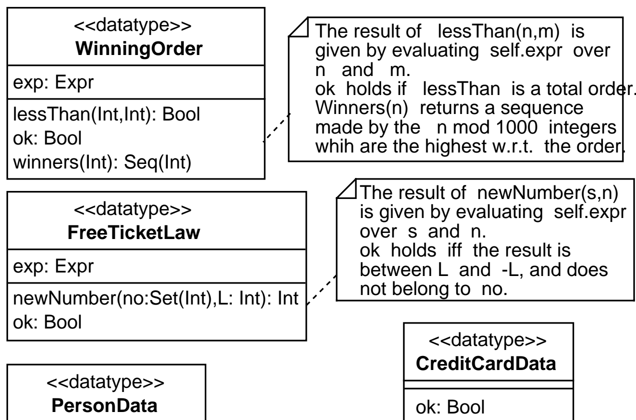

Figure 6. AL_L: Data View

context, external (w.r.t. the Application) entities that interact with it;

boundary, entities composing the Application that take care of the interaction with some context entities;

executor, entities composing the Application that perform some core Application activities;

store, entities composing the Application that contain persistent data[3].

Technically, the Static View is a UML package, importing the package Data View, and containing a class diagram, say CD, with the following characteristics.

The attributes of each class in CD must be protected.[4] The # symbol signalling that an attribute is protected may be omitted, since it should appear in front of any attribute.

Each class in CD is of one of the following stereotypes:

≪context≫ stereotype of active class. An instance of a ≪context≫ class is a context entity.

Context classes cannot have attributes, and there should be one of them for each class of stereotype ≪SU≫ or ≪SP≫ in the Context View of the Requirement Specification.

≪boundary≫ stereotype of active class. An instance of a ≪boundary≫ class is an object that takes care of the interaction of the Application with some context entities. It receives messages from the context entities, analyses them and afterwards either sends back an immediate answer (e.g., if the received message contains an error) or interacts with other entities as required by the message.

≪store≫ stereotype of passive class. An instance of a ≪store≫ class is an object that contains persistent data (e.g., a database).

≪executor≫ stereotype of active class. An instance of an ≪executor≫ class is an object that performs some core Application activities as result of information received from the context through the boundaries; clearly during such activities it may collaborate with other entities.

A specialization relationship in CD may be only between classes having the same stereotype.

Composition and aggregation relationships may be used to define "subobjects/subentities". Composing classes may have the same stereotype of the composed class or no stereotype; in this case they are just used to modularly decompose a complex class.

Each association in CD, different from specialization/composition/aggregation, must be of the stereotype ≪communication≫.

A ≪communication≫ association from class C to class C' (visually depicted as an oriented arrow from C to C') means that an instance of class C interacts with some instances of class C' by calling their operations. The multiplicity assertions state how many instances it will interact with.

The ≪communication≫ associations should respect the following conditions:

- no communication association may leave a ≪store≫ class, because stores are fully passive and should be used only by the executors and the boundaries;
- ≪context≫ classes may communicate only with ≪boundary≫ classes;
- any multiplicity constraint about ≪context≫ classes must be in accord with those appearing in the Context View of the Requirement Specification;
- any class in the diagram must be connected by a communication chain with at least one context class (otherwise it means that a part of the system is isolated and thus useless).

A ≪communication≫ association may be anonymous, and in this case it just models the information flows inside the Application, or it may be named, in this case it models also the fact that the source class has an attribute named as the association itself and whose type is the target class, or a set of the target class (depending on the multiplicity).

We show in Fig. 7 the Static View of the AL⊥ case study. The AL_L application is quite simple and so also its architecture, shown in Fig. 7, is rather simple; we do not need ≪executor≫ classes, because the ≪boundary≫

[3] Data which are preserved also when the Application stops or breaks down.

[4] A protected attribute is visible only inside its class and inside any of its specializations.

classes may take care of the activity due to the requests of the users, using in some cases some auxiliary classes, e.g., GiveFreeTickets and DrawTickets. There are store entities for the database of the registered clients (ClientInfo) and the data about the current lottery (Lottery). In such diagram we use for the classes of the stereotype ≪context≫ the same icons used in the requirement phase, precisely, the parallelogram for the service providers and the stick-man for the service users.

4.3 Behaviour View

A Behaviour View describes the behaviour of the entities of a given class (introduced in the Static View). We think that store entities, being just data containers, have a standard behaviour consisting in receiving operation calls; because we further assume that any call may be received in any moment, and that no two calls may be executed simultaneously, to model their behaviour it is sufficient to model their operations. For this reason, technically we have two different kinds of Behaviour View:

- describing the behaviour of either a ≪boundary≫ or an ≪executor≫ (active) class; in this case it is a statically consistent statechart associated with such class;
- describing the behaviour of an operation of a ≪store≫ (passive) class; in this case it may be:
  - a UML method definition associated with such operation (that is a program written using the UML actions);
  - a constraint of the form pre/postcondition for such operation, expressed using OCL;
  - an activity diagram associated with such operation. An activity diagram defines the behaviour of an operation at a rather abstract level; in particular the fork construct is used to define activities whose ordering is irrelevant.

It is mandatory to define the behaviour of any ≪boundary≫ and ≪executor≫ class, and of any nontrivial operation of any ≪store≫ class (trivial operation are, for example, those getting and setting the attributes). The behaviour of the context entities cannot be defined, because they are not one of the aims of the design. Some information on their behaviour may be found in the Context View of the Requirement Specification; if it may help understand the design, it may be copied in the design model.

We assume that the entities composing the Application behave in a parallel way without any restriction; under this assumption, the description of the behaviour of the composing entities it is sufficient to describe the behaviour of the whole Application.

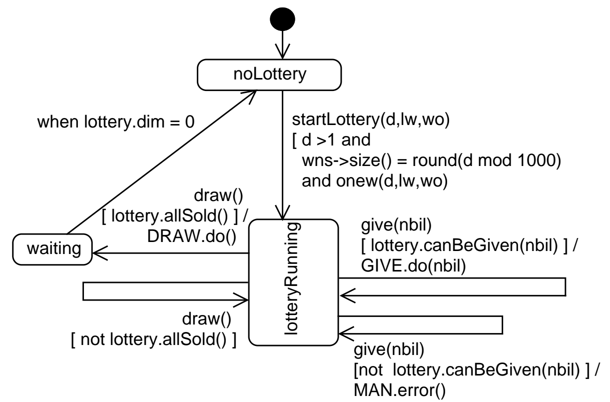

Figure 8. ForManager: Behaviour View

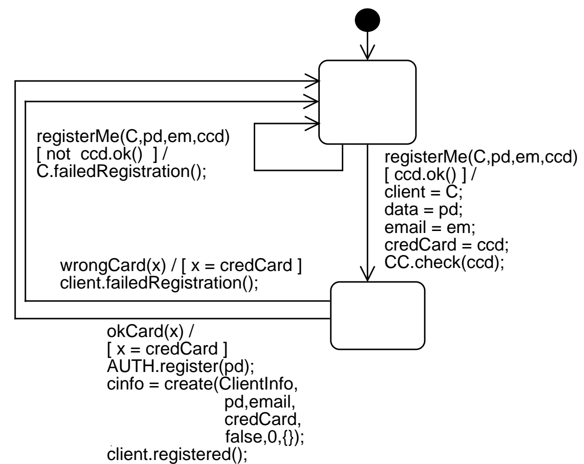

Figure 9. ToRegister: Behaviour View

In our approach, the decomposition of the Application in entities should be considered as a kind of logical decomposition, in the sense that such decomposition does not convey any precise information on which activities must/may/cannot be performed in parallel. The aspects of Application concerning parallelism and distribution will be considered in the next Technology-Driven Design steps.

In Fig. 8, 9 and 10, we present the behaviour of the most relevant ≪boundary≫ classes (those of the boundaries towards the service providers are quite trivial). To keep the statecharts corresponding to the various behaviour views quite simple and to follow our method, which requires to fully encapsulate the used (passive) classes, we have defined many auxiliary operations, for example, allSold and canBeGiven for the We collected in Fig. 11 the Behaviour Views for the operations of the passive classes, which have all the form of a method definition.

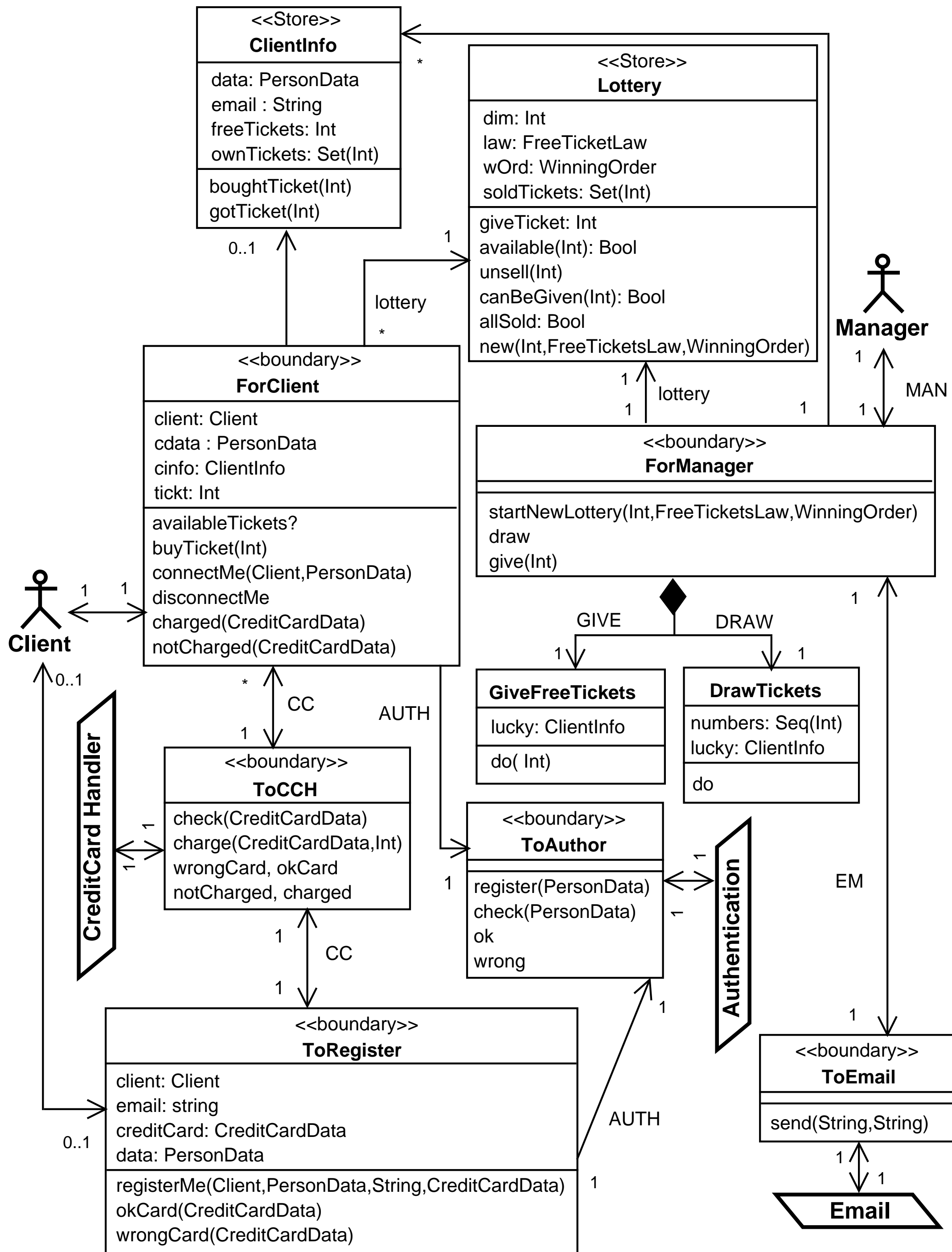

Figure 7. AL_L Static View

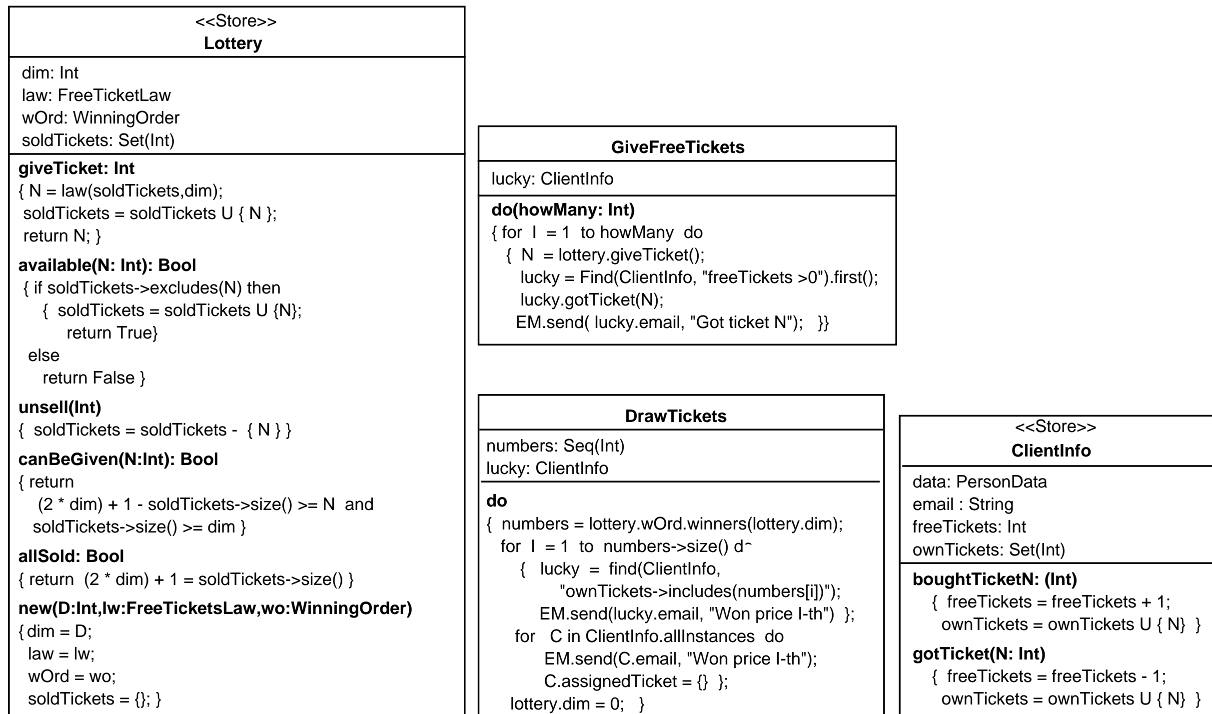

Figure 11. Operations of passive classes: Behaviour Views

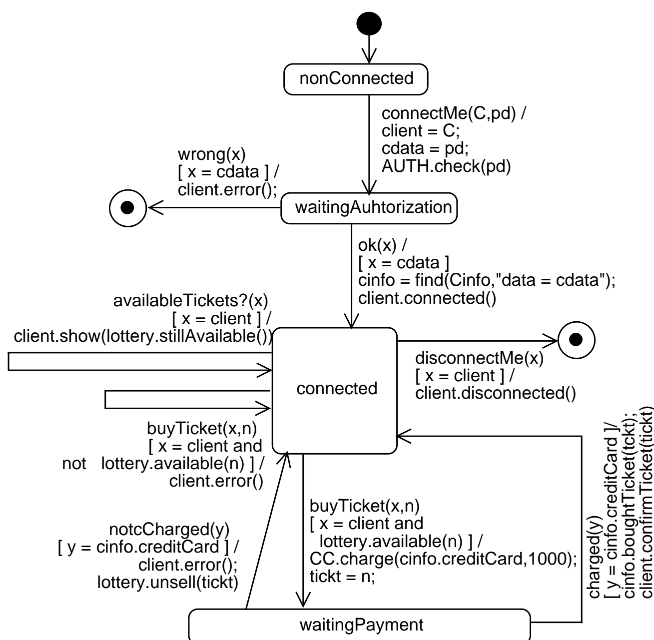

Figure 10. ForClient: Behaviour View

4.4 Configuration View

A Configuration View describes the run-time structure/architecture of the Application at some given point/ situation during its life, by stating which are the entities composing it and how they interact among them.

Technically, a Configuration View is a collaboration diagram without messages in statically consistent with the Static View [5]; it describes the entities composing the application in the given situation, and the links of the communication associations show how such entities cooperate among them.

In Fig. 12 we present two Configuration Views for the AL_L application depicting respectively the initial configuration, and the situation when one client is trying to register, another one is connected and a third one is registered but not connected.

4.5 Additional View

An Additional View describes how some entities of the Application interact among them to accomplish some particular task. An additional view is optional and intended just for documentation; it should not add any information not already present in the other views.

[5]This means that the roles and the links are typed by the classes and associations present in the Static View, and that any multiplicity constraint in that view is respected.

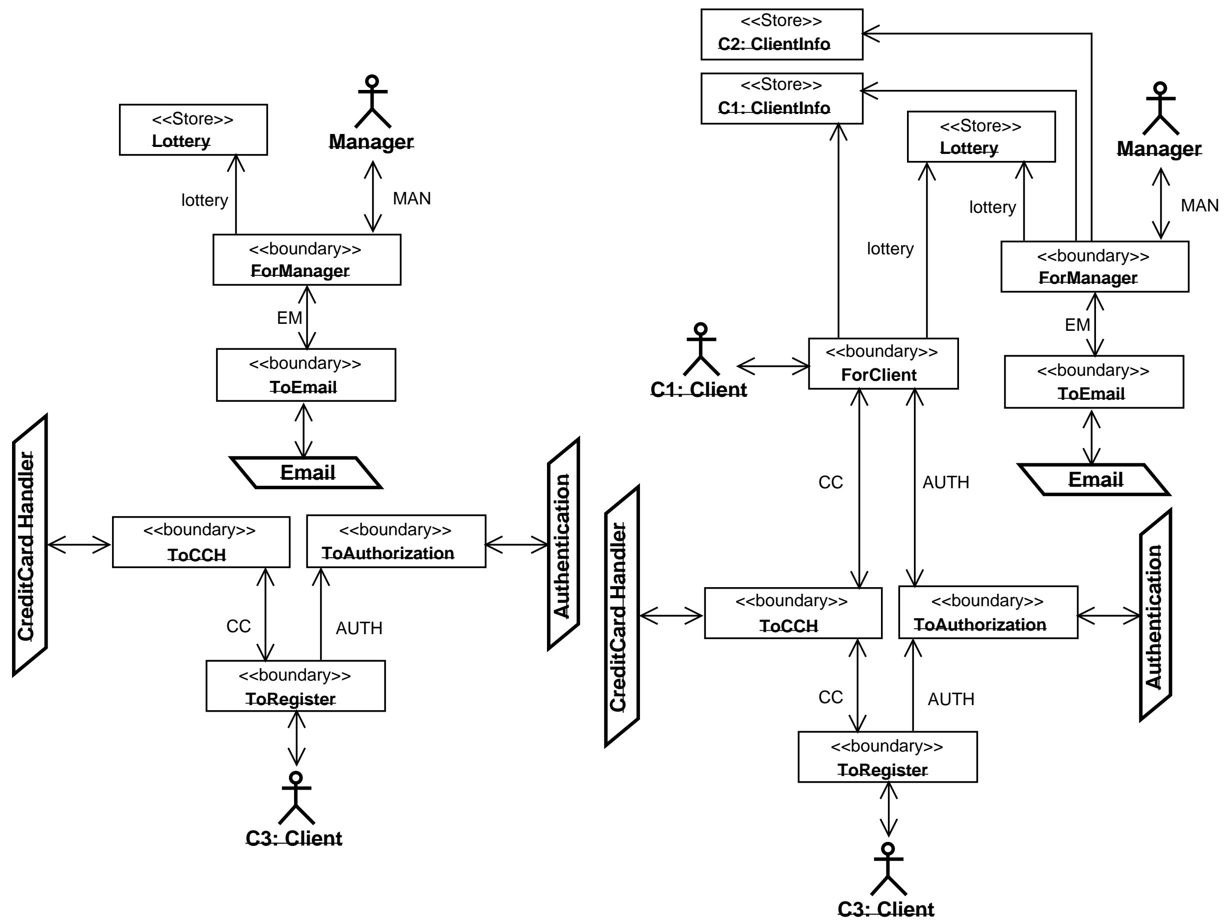

Figure 12. AL_L: Configuration Views

Technically, an Additional View consists of a UML diagram (either sequence or collaboration or activity) statically consistent w.r.t. the class diagram of the Static View. Moreover, any information on the behaviour of the entities composing the Application presented by an additional view must be consistent with the complete description of that behaviour presented by the various Behaviour Views.

The Additional Views are completely optional, in the sense that they do not influence the design of the Application; their role is just to document some aspects of the Application by presenting them, in more appealing ways, as scenarios or causally related facts.

We think however that trying to produce some Additional Views may help the developers check the design or understand the behaviour of some parts of the Application, and that a development method may usefully requires to produce many of them. For example, Additional Views may be used to present how the various Interaction View and Causal View of the use cases presented in the Requirement Specification have been realized; or they can be produced by the designer to help clarify the behaviour of the Application in some cases before giving the Behaviour Views of the concerned entities.

We show in Fig. 13 and 14 two additional views for the AL_L application, corresponding to two interactions of two use cases in the Requirement Specification, given in [5]. They do not define at all the behaviour of the AL_L application, already fully defined by the various Behaviour Views, but help understand how the two use cases are realized by the Model-Driven Design, and to become confident in the correctness of that realization. For example, in Fig. 13 we can immediately see that the manager will never receive a confirmation that the ticket has been effectively given out.

4.6 Interface View

An Interface View is associated with a ≪boundary≫ class present in the Static View and describe the graphical interface of the instances of such class towards the context entities interacting with them. An Interface View is mandatory for each boundary class interacting with human context entities.

There is no specific request on the format of an Interface View. Some possible ways to present an Interface View are

- a free visual presentation of a GUI,
- a UML presentation of a GUI using perhaps appropriate stereotypes, such as button, check box, and menú,
- a definition of some textual line commands.

The only relevant point is that the connection between the elements of the interface and the operations of the associated boundary and context entity classes must be precisely stated.

For lack of room here we do not give here any example of Interface View.

5 Checking the Quality of a Model-Driven Design

Here we list some checks to be performed on the produced Model-Driven Design to detect problems.

Minimality The following checks avoid to define useless parts of the design.

- All datatypes defined in the Data View are used at least once, and all their operations are called at least once.
- All operations of a ≪store≫ /≪executor≫ / ≪boundary≫ class are called at least once.

If one of the above checks fails, it is easy to solve the problem by removing some part of the design.

- All operations of a ≪context≫ are called at least once by a ≪boundary≫.

In this case we can have also a design error (some requirement is not fully implemented) or some redundancy in the requirements.

Soundness
The assumptions on the communications among the entities composing the Application as stated by the ≪communication≫ association must be respected by the behaviours described buy the various Behaviour Views.

Correctness
In the development method that we have presented here, we use a subset of UML to which a formal semantics can be given (see Sect. 3). Today, a lot of formal notions and techniques partly supported by software tools are available to precisely state, and then to formally check, whether a Model-Driven Design is correct w.r.t. a Requirement Specification. However, we do not think that with the current state-of-the-art we can devise a practical development method including a formal proof of the correctness of the produced design. Instead, we propose an inspection technique grounded in the underlying formal framework allowing to gain an acceptable confidence that a Model-Driven Design correctly realizes a given Requirement Specification. Because of the precise structuring in different views of both the requirement and the abstract design models, this inspection activity may be structured in a set of related tasks, each one covering a particular aspect; and it is then possible to implement a software tool realizing a wizard guiding the developer through those tasks.

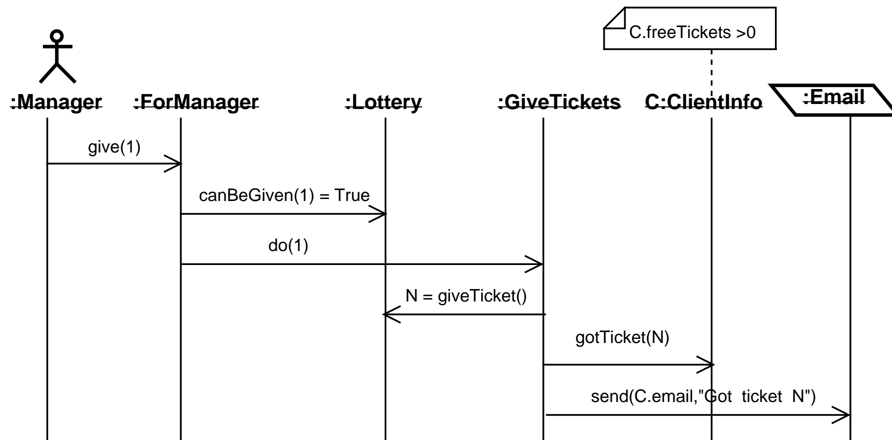

Figure 13. The manager gives one free ticket: Additional View**:**

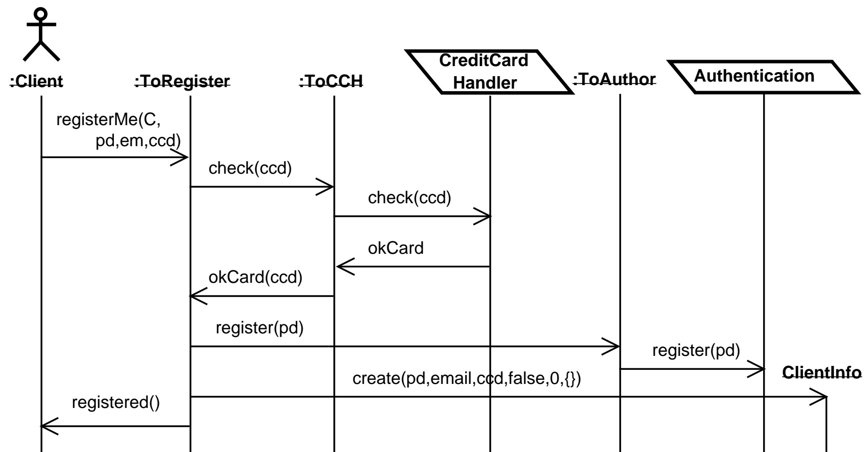

Figure 14. Client registration: Additional View**:**

6 Related work and conclusions

We have presented here in a very short outline a possible instantiation of a general strategy that we call Well-Founded Software Development Methods. That strategy is guided by three imperatives:

- the issues addressed should be potentially relevant to software engineering practice; to that end, we look at and borrow, as much as we can, from the best practices in the field and from the methodology side, avoiding what we see as the formalism (not so) splendid isolation;
- the discipline coming from the area of formal techniques imposes both the use of semantically well-defined structures and of rigorously justified methods;
- the formal aspects should not be forced on the end-users, who in principle are not experts, but used in the background to achieve the second imperative.

There is a sociological consideration at the root of our strategy. It is a constant finding that formal techniques are liked and accepted only by people extremely well-trained in formal techniques, that is not the case of the vast majority of software engineers. Moreover, there is a reported tension, if not a contrast, between formalism and productivity/efficiency. Hiding formalities, but keeping their disciplined rigour in the methods they use, is a strategy that can overcome both disadvantages. In support of this belief, we have made some experiments in the last two years with undergraduate students without any formal background, with the exception of an elementary short course in mathematical logic. The results have been very encouraging, especially compared with previous experiments, some using explicitly formal techniques with students better skilled in formalities and, on the opposite side, some using visual techniques (UML) and related methods as they usually are, namely without any underpinning rigour.

The approach that we have outlined (see [5] for an extended version with interesting and complex case studies), is in the line of some of the best-known methods for software development, adopting a multiview and use case approach and using the UML notation. But it departs from them, at least to our knowledge, in some important respects, both from the methodological and the technical viewpoint.

First, on the method side, the overall major goal is to propose a more systematic and stringent approach, in the sense that the overall structure of our artifacts is constrained in order to tightly relate the components and have at hand the possibility of performing a number of consistency checks. This view contrasts with the almost total freedom given, for example in RUP [14], where the structure is just based on the use case description. The same freedom, just use case diagrams and use case description, is given for the Requirement Specification phase in COMET [9], in sharp contrast with the detailed structure and the many constrained guidelines and notations for Analysis and Design. That level of freedom is, on the other hand, explicitly advocated, for example in [8], on the basis that experience matters more than stringent structuring and rules. There the underlying philosophy is admittedly the same of the Agile Methods Movement (see [17], for an interesting discussion and references). However, while we do not deny that highly skilled and experienced software developers perhaps need only loose guidelines and a supporting liberal notation, from our experience we have seen that, for less experienced people, such liberality is a source of endless discussions, contrasting choices and a proliferation of inconsistencies. Moreover, we believe that our "tight and precise" imperative and the related techniques may help from one side reduce the amount and the fuzzy verbosity of some documentation and on the other provide effective guidelines for passing to the design and then the implementation phase, though we have not yet explored all the later phases.

The approach taken in Catalysis [7], that in other details shows some similar general views to ours, is not directly comparable, being an overall transformational approach based on components that are refined from business modelling to implementation units. But definitely our way of structuring requirements is not targeted to a transformational approach; we are more interested in providing a separate step preliminary to devise in a rather structurally independent manner, a model-driven software architecture of the system.

Indeed our approach is totally compliant with the OMG Model Driven Architecture philosophy (see [12]) and it is within that framework that we intend to explore the connection with the implementation phase, passing from Platform Independent Models to Platform specific Models and then to code.

Currently we are developing some supporting tools, starting from the open and freee ArgoUML, for checking the consistency of our artifacts. Moreover we are investigating the use of inspection techniques to check the correctness of the Model-Driven Design w.r.t. the Requirement Specification. There too the formal techniques help; indeed those techniques are guided by the notions developed by Hoare (abstraction function in data type implementation) and by Sannella and Wirsing (correct implementation of abstract data types), see [19].

References

[1] E. Astesiano and G. Reggio. Formalism and Method. *T.C.S.*, 236(1,2), 2000.

[2] E. Astesiano and G. Reggio. Labelled Transition Logic: An Outline. *Acta Informatica*, 37(11-12), 2001.

[3] 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`.

[4] E. Astesiano and G. Reggio. Consistency Issues in Multiview Modelling Techniques. In *Recent Trends in Algebraic Development Techniques, Selected Papers of the 15th International Workshop WADT'02*, Lecture Notes in Computer Science. Springer Verlag, 2003. To appear. Available at `ftp://ftp.disi.unige.it/person/ReggioG/AstesianoReggio03b.pdf`.

[5] 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`. It will appear in "Radical Innovations of Software and Systems Engineering in theFuture", Proc. of 2003 Monterey Workshop, Lecture Notes in Computer Science, Springer Verlag.

[6] E. Astesiano, G. Reggio, and M. Cerioli. From Formal Techniques to Well-Founded Software Development Methods. In *Proc. of The 10th Anniversary Colloquium of the United Nations University International Institute for Software Technology (UNU/IIST): Formal Methods at the Crossroads from Panacea to Foundational Support. Lisbon - Portugal, March 18-21, 2002.*, Lecture Notes in Computer Science. Springer Verlag, Berlin, 2003. To appear. Available at `ftp://ftp.disi.unige.it/person/ReggioG/AstesianoEtAll03a.pdf`.

[7] D. F. D'Souza and A. C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Object Technology Series. Addison-Wesley, 1999.

[8] M. Fowler and K. Scott. *UML Distilled: Second Edition*. Object Technology Series. Addison-Wesley, 2001.

[9] H. Gomaa. *Designing Concurrent, Distributed and Real-Time Applications with UML*. Addison-Wesley, 2000.

[10] M. Jackson. *Software Requirements & Specifications: a Lexicon of Practice, Principles and Prejudices*. Addison-Wesley, 1995.

[11] M. Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems*. Addison-Wesley, 2001.

[12] OMG Architecture Board MDA Drafting Team. Model Driven Architecture (MDA). Available at `http://cgi.omg.org/docs/ormsc/01-07-01.pdf`, 2001.

[13] P. Mosses. CoFI: The Common Framework Initiative for Algebraic Specification and Development. In M. Bidoit and M. Dauchet, editors, *Proc. TAPSOFT '97*, number 1214 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1997.

[14] Rational. Rational Unified Process© for System Engineering SE 1.0. Technical Report Rational Software White Paper: Tp 165, 8/01, 2001.

[15] 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 Lecture Notes in Computer Science. Springer Verlag, Berlin, 2000.

[16] 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 Lecture Notes in Computer Science. Springer Verlag, Berlin, 2001.

[17] D. T and B. B. The Agile Methods Fray. *Computer*, 2001.

[18] UML Revision Task Force. *OMG UML Specification 1.3*, 2000. Available at `http://www.omg.org/docs/formal/00-03-01.pdf`.

[19] M. Wirsing. Algebraic Specifications. In J. van Leeuwen, editor, *Handbook of Theoret. Comput. Sci.*, volume B. Elsevier, 1990.

[20] E. Yourdon. *Modern Structured Analysis*. Prentice-Hall, 1989.