# From Formal Techniques to Well-Founded Software Development Methods *

E. Astesiano, G. Reggio, and M. Cerioli

DISI, Università di Genova - Italy

**Abstract.** We look at the main issue of the Colloquium "Formal Methods at the Crossroads from Panacea to Foundational Support" reflecting on our rather long experience of active engagement in the development and use of formal techniques. In the years, we have become convinced of the necessity of an approach more concerned with the real needs of the software development practice. Consequently, we have shifted our work to include methodological aspects, learning a lot from the software engineering practices and principles.
After motivating our current position, mainly reflecting on our own experience, we suggest a *Virtuous Cycle* for the formal techniques having the chance of a real impact on the software development practices. Then, to provide some concrete illustration of the suggested principles, we propose and advocate a strategy that we call *Well-Founded Software Development Methods*, of which we outline two instantiations.

## 1 Introduction

The concern reflected in the title of the Colloquium[1] and explained in its motivation came as a no surprise to us. We have been involved in the development and experimental use of formal techniques since more than twenty years and we have witnessed and experienced dramatic changes. The changes, positive and negative, refer to many aspects of the research in the field: quality, amount, scope, relevance and acceptance. In our community of formal techniques there is no need to push for more research in theoretical work in general, since the advocates of the merits and value of such research abound, as much as the good papers in the field (see, e.g., the over 8000 pages of the 2001 issues of a prestigious journal such as TCS). Thus, here we concentrate more on some critical issues, mainly concerning our personal experience from which we try to learn some lessons.

The first uncontroversial fact is the large gap between theory and practice and the increasing awareness, starting in the middle 90's, that the many hopes for a

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

[1] 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.

prominent role of formal techniques in software development were fading away, as recognized even by some of the pioneers and more convinced supporters (see [12]). Things have not changed much since, but in a sense they have worsened for what concerns academic research on software engineering in general, as noticed by B. Meyer in [20]:

"*When considering the evolution of the field over the last decades, we cannot escape the possibly unpleasant observation that if much of the innovation in the '60s and 70's came from academic institutions, contributions from small entrepreneurial companies and the research labs of large companies dominated the '80s and '90s.*"

The reactions to the existence of such gap between theory and practice have been mixed and sometimes diverging. In his invited lecture at Formal Methods Europe '96 in Oxford [14] T. Hoare was suggesting

"*We must aim our future theoretical research on goals which are as far ahead of the current state of the art as the current state of industrial practice lags behind the research we did in the past. Twenty years perhaps?*"

History shows that such remark may well be applied to few fundamental pioneering concepts and insights, but cannot be taken as a rule for research in formal techniques. For example, more than twenty years ago, the foundations have been laid down for a neat formal treatment of requirement and design specifications, including clean semantics. Consider e.g., the enormous effort in the techniques for algebraic specification, as summarized in an IFIP State-of-the-art Report jointly produced by the members of the IFIP WG 1.3. [2], not to mention the wealth of results on denotational and operational semantic, for almost any kind of languages.

Now, after so many years of work in formal techniques, what do we have on the table of the current practice in the corresponding area? The most evident phenomenon is the Unified Modeling Language UML (see, e.g., [29]), not only a standard of the Object Management Group (OMG[2]), but a de facto industry standard for modelling/specifying systems at different levels of abstraction and from different viewpoints. The UML intends to "*provide a formal basis for understanding the modeling language*" and indeed it tries to incorporate not only many lessons from the software engineering side, but also some good ideas about abstraction, encapsulation, and dynamic behavior developed by researchers in formal techniques in the years before. Nevertheless, the result for most of us is frustrating: not even the static/syntactic semantics is clear, and as for the semantics in general "*UML expresses the operational meaning of most constructs in precise natural language*".[3] Still the UML represents a significant advance over current industry practices, for what concerns, e.g., the use of an appropriate level of abstraction and the separation between modelling/specification and implementation, an issue emphasized very recently by the adoption of the Model

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

[3] By the way, in this regard we can even find in the reference manual [29] a pearl statement such as "*The fully formal approach taken to specify languages such as ALGOL 68 was not approachable enough for most practical usage*".

Driven Architecture (MDA) [18] by the OMG, as a separate step from the building of a Platform Driven Architecture. Moreover, to be fair, one has to say more generally that, if on one side the figures related to failures in software projects are appalling, on the other side one can be amazed that the existing software is quite often satisfactory enough. Significantly, indeed, the cited talk by T. Hoare at Formal Methods Europe '96 [14] had the curious title "*How did software get so reliable without proof?*". I must however mention that the software consumer view may be different and not so positive, as explained in a very interesting book *The software Conspiracy* by M. Minasi [21].

All the above to say that, apart from some very few remarkable exceptions, there is little chance for theory in isolation to filter in the practice in a sensible way. We need to look ourselves at what is going on in the practice of software development. We are encouraged in this belief by the following two remarks, always by T. Hoare in [14], "*personal involvement in current practices ... may lead to quite rapid benefits, both to the practitioner and to the theorist*" and "*the influence of practice on the development of theory is more beneficial and actually quicker than the other way round.*" Indeed, the last two remarks may sound surprising w.r.t. the previous statement on the quest for theories looking twenty years ahead. We are instead inclined to believe that the three remarks complement each other to give a correct perspective. Here, obviously we restrict our attention to the interplay between research in formal techniques and its impact on software development practices. That restriction relies on an assumption better phrased, and with the authority of a pioneer of formal methods, by our good friend C. Jones in his invited talk at the first ETAPS-FASE'98 [17] "*I assume that the purpose of developing formal methods is to influence practical engineering of computer systems (whether hardware or software).*" Such assumption should sound trivial, but it must not be so, since he adds immediately: "*It is a measure of my unease with some research in the area of computer science that I feel necessary to state this fact*". That quotation leads us to introduce the main general thesis of our contribution and position, as an answer to the debate raised by this colloquium.

Let us assume that our concern is software development and then let us try to define a suitable role for formalization. Here we are soon faced with an ill-defined dichotomy that has to be dismissed, as pointed out by B. Meyer in [19]:

"*For some scientists, software development is a branch of mathematics; for some engineering it is branch of applied technology. In reality, it is both.*"

Thus, inevitably the *Great General Challenge* of the work in formal techniques is to address in a recognized way the relevant issues in software development, paying more attention at the engineering needs. We are well aware that the above statement sounds like preaching and needs to be made concrete. So, in the following we try to motivate and articulate our thesis. First we reflect on our own experience to draw some lessons (Sect. 2); then, to illustrate more concretely our view, we outline some of our current directions of research towards a viable strategy that we call *Well-Founded Software Development Methods* (Sect. 3).

Recall that in the conclusions we try to summarize and make an assessment of the proposed strategy.

## 2 Shifting the Focus and the Attitude of our Research in Formal Techniques

The following reflections are certainly biased by our experience and thus we think it is better to present them mainly as reflections on the work our group has done in the years on the subject.

### 2.1 On the Formalist Side

For what matters in this context, the work of our group on the use of formal techniques in the specification of systems has started at the beginning of the '80 when we have been engaged in a very ambitious national project on the prototypical design and implementation of a local area network, the Cnet project. Our task was the formal specification of the rather sophisticated software architecture, in two levels: the intra-node with Ada-like tasks and the inter-node with both point-to-point and broadcast communication. After some serious attempt at using CCS and CSP, we soon became aware that higher-level specification techniques, also able to manipulate complex structures, data and processes as data, were required. That led us to develop in the years an approach to the specification of possibly concurrent/reactive systems (see [4] and [6] for a later comprehensive outline), which, with some further improvement [3], was then used successfully and adopted as the technical basis in the EEC project (lead by D. Björner) for the full Formal Definition of Ada ('85/'87).

In those years, and during those projects, we have experienced the value of our formal setting to make precise any aspect of the modelled systems in an unambiguous way. First, in the Cnet project there was an endless feedback with the designers of the architecture and always the formalization was proved helpful. Then, in the Ada project the people in the board for the language standard were forced to accept some of our proposed modifications to make the language consistent. The presentation of the full definition, in a 13 hours seminar (by the first author and by J. Størbark Pedersen, then at DDC, DK) at Triangle Park-USA, to the people of the Ada Joint Program Office, and later at an international school, was received with enthusiastic comments, which were then forwarded to the EEC officers.

That was a time when we were convinced that the formal techniques were the real solution to put software development on a firm basis. That atmosphere was reflected in the title of another EEC initiative, lasting altogether seven years, the Esprit BR WG COMPASS (COMPrehensive Algebraic Approach to Software Specification) '89-96. It is within that project that was conceived the idea of publishing a comprehensive book on the algebraic specification of systems leading to the mentioned IFIP State-of-the-art Report sponsored by IFIP WG 1.3. [2]. Moreover, just before the end of the COMPASS initiative, we contributed

to start the CoFI initiative (Common Framework Initiative for algebraic specification and development of software) [22], whose first aim was to define, within a future family of related extension and restriction languages, CASL, the Common Algebraic Specification Language, approved by the IFIP WG 1.3, see [1]. Of course CASL, which encompasses all previously designed algebraic specification languages, has a clean, perfectly defined semantics. Within CoFI, some extensions of CASL have been provided to deal, e.g., with concurrency and reactivity; our LTL [6] can be seen as one of these extensions. Still the target of the work within CoFI was and is the research community and not industry practice.

### 2.2 Some Lessons from the Practice and the Software Engineering Sides

While contributing to the development and improvement of formal techniques we have been involved in some projects with industry to apply our techniques to real-size case studies. It was soon evident that our (us and our community) work 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 presented at the last TAPSOFT in Lille in '97 (for a journal version, see [5]). 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 (system, item) and the formal structures (models) representing it,
- the role of pragmatics.

In Fig. 1 we outline those distinctions for the case when our aim is to provide a rigorous specification of a system, at some level of abstraction. In that paper we have also suggested the use of some systematic way of presenting a method, with the use of method patterns. The target of that study was essentially the formalist community. 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. In addition, we were opposing the widespread attitude of conflating formalism and method, with the inevitable consequence, within a community of formalists, of emphasizing the formalism or even just neglecting the methodological aspects. Curiously enough, we discovered that in a sense the abused phrase *formal method* is somewhat strange etymologically, since the word *method* comes from Greek and means *way through* and its Latin translation (both *ratio et via* and *via et ratio*) was making explicit that that word was conveying the meaning of *something rational with the purpose of achieving something, together with the way of achieving it.* Later it was very illuminating and encouraging to discover that more or less the same remarks,

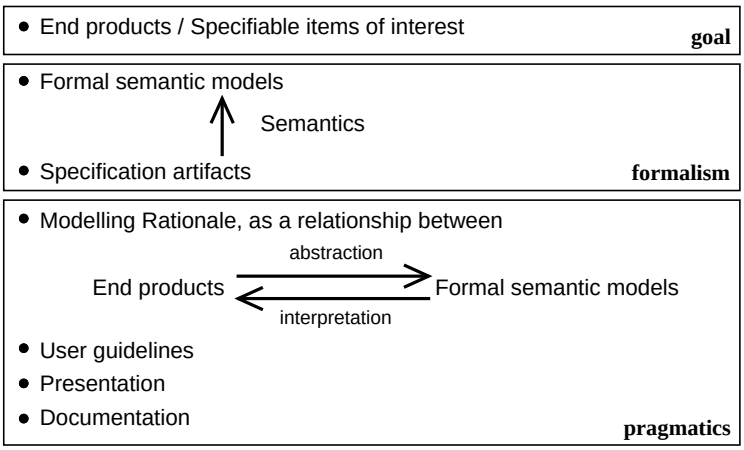


**Fig. 1.** Formalism and Method

but from a methodological viewpoint relating software development to the usual other engineering practices, were made by D. Parnas, who at about the same time was writing in [23] "*the phrase "formal methods" is strange and the issue of technology transfer of such methods even stranger*".

Since then, we were always trying to look at the lessons from the Software Engineering side. For example, the work of M. Jackson [15, 16] with his careful analysis of the method and modelling aspects was and is very inspiring. More generally, we have personally learnt how much it is advisable for people working in the area of the formal techniques to try to put the work into the perspective of what is considered essential for the development of software. To be short, we can, for example, look at the notions that A.I. Wassermann in his '96 Stevens lecture [30] considers fundamental in software engineering, which are Abstraction, Analysis and Design Methods and Notations, Prototyping, Software Architecture, Software Process, Reuse, Measurement, Tools and Integrated Environment. Definitely, the area of formal techniques has contributed enormously to provide tools for Abstraction, and a wealth of Notations; but how many of those are really linked to, or exploitable by, Methods of Analysis and Design? Then, it would be quite interesting to make a serious assessment of the contributions to the other six notions.

Another useful lesson and challenge comes from the continuous evolution in today's applications. In the words of C. Jones [17]

"*Much has happened in computing since the* stack *and the problem of* Dining Philosophers *were first taken as important paradigms on which to test formal approaches. Whatever the disadvantage of modern software (and I know many of them), significant systems are now constructed on top of flexible and general interfaces to packages which handle much of the detail of - for example - the precise presentation on the screen.*"

We do not believe that we formalists have coped with the evolution fast enough. Let us just mention one paradigmatic example. We have available plenty

of formal techniques for explaining the principles of programming languages (not to say of the multitude of researchers still working on the minimal details even of languages sometimes quite dated). But when, early this year, we have tried to set up a university course providing viable clean principles for building software in a component-based way, using open systems, and the like, we could not find any rigorous setting allowing us to teach the basic techniques without having to come out just with teaching the Java/SUN or the Microsoft/DCOM/.Net way. We are still looking around ....

### 2.3 The UML Case

We single out the emergence of the UML as a de facto standard (it is anyway an OMG standard) because it is for us a paradigmatic example of an avoidable mismatch between formal techniques and practical software engineering. In a bit more ideal world we would speak of a lost opportunity for good formalisms to meet best practices.

Indeed, the UML and the related work is trying to provide an answer to reasonable requests from the software developers. It is visual, multiview, offering notations for different phases of the development from requirements to document management, supportable and supported by tools. Moreover, in principle it incorporates a number of concepts and techniques provided in the years by various streams of research: object orientation; abstraction in a number of ways, starting with the central concept of model/specification; behaviour description by means of statemachines (borrowed from D. Harel's work) and activity diagrams (from Petri Nets); though not mandatory, OCL, the Object Constraint Language, which is a variant of first order logic for expressing properties on parts of the models. Altogether, the UML, used in a sensible way, is far better than the majority of notations and techniques used in the current industry practice.

Still, strictly measured as support to rigorous development, it is a nightmare. Not only it lacks a comprehensive formal semantic foundation (that could apply to Java too), but the semantics is at its best based on common understanding. Differently from Java, even the informal semantics has still to be defined for some aspects and the one provided is a source of many ambiguities. Being the UML built following a typical notation-aggregation approach, even the attempts in the literature to provide a semantics by many researchers are limited to isolated subsets, usually with restrictive constraints w.r.t. their usage. Furthermore, there is another problem, how to assembly the different formal semantics of the various sub-notations to get a consistent semantics of the complete UML.[4] And of course it has the usual limitation of a purely object-oriented approach, namely the lack of direct support for concurrency, cooperation, distribution, mobility, and the like. Moreover, according to some studies by well-known software architecture

[4] This task is far from trivial and needs skills that our community developed and proved, for instance, in the definition of the CASL. Indeed, the Common Algebraic Specification Language includes features and aspects developed and studied in isolation in several other specification languages having non-obvious intersection.

experts, its support to define software architectures is definitely non-explicit and somewhat controversial. There are different possible ways for representing software architectures and none without some mismatch, at least as they are seen in that community.

Thus, why do we speak, ideally, of lost opportunity? Because we are convinced that at the same time of the birth of the UML, it was the mid-nineties, there was in our research community all the knowledge and the ability for coming out with a much better proposal encompassing by far the current UML. Our conviction is grounded on the studies we have performed both on many semantic issues of the UML (see [26, 27]) and on possible reformulation of some of its aspects in a cleaner way (see part of the following section). Admittedly, that ideal proposal could have happened in a different world, both because the market is following different routes than research, and because our research community is neither cooperative nor much sensible to the real needs of software development. Still we notice that it was not the case of a good substitute for the UML swept away by a market move; simply there was not such a substitute.

### 2.4 A Virtuous Cycle for the Research in Formal Techniques

There has been a clear lesson for us from the above and has been a different attitude in the way we devise, use and advocate formal techniques. 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). To be concrete, in the following section we will outline two experiments 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.

## 3 Some Strategies and Experiments in the Search for *Well-Founded Software Development Methods*

### 3.1 An Experimental Java Targeted Notation

Our oldest line of research in *Well-Founded Software Development Methods* (see [10]) stemmed directly from our work in the specification of concurrent systems with algebraic techniques and is aimed at providing a notation for the specifications, at design level, of concurrent/distributed/reactive systems. That notation is Java targeted, in the sense that the specified systems should be ideally implemented in Java, and has three special features: it is

– graphical, i.e., given by diagrams,
– completely formal, i.e., amenable to a formal description with algebraic techniques (but hidden from the user),
– endowed with an easily automatized (correct) translation into Java.

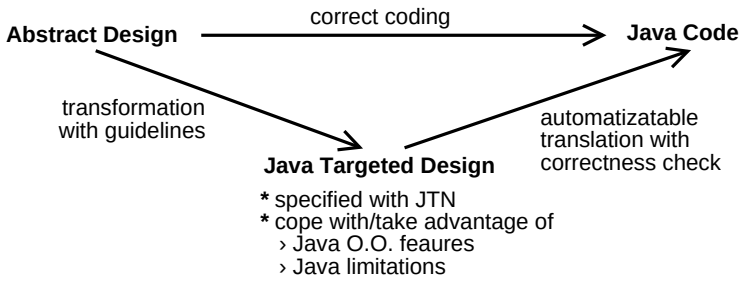


**Fig. 2.** The JTN approach

In Fig. 2 we provide a graphical view of the approach supported by JTN.

By *Abstract Design* we mean a specification given in the graphical notation of [28]. Essentially, it consists of a visual representation of an algebraic specification of a concurrent system at design level, following the LTL approach [6]. The syntactic form of such specifications guarantees that it corresponds to a positive conditional algebraic specification, which admits the existence of a unique (modulo isomorphism) well-defined labelled transition system matching the specification. Notably, a similar approach has been recently adopted within the MDA (Model Driven Architecture) OMG Proposal [18]. Using the MDA terminology, the abstract design specification is a platform independent model (PIM), in the sense that it does not depend on a particular implementation platform (Java), whereas the Java Targeted Design specification is a (Java) Platform Specific Model (PSM), and the transformation from the Abstract to the Java Targeted design corresponds to the mapping into the Java implementation platform.

**A specification (model) consists of**

**Class diagram**
- datatype/passive/active class interfaces
- dependency - specialization relationships among them

**Body diagrams** for all classes in the class diagram, different for each kind of class

**Architecture diagram (system structure)** a graph where
- nodes: system constituents (passive/active objects)
- arcs: connectors (synchronous/asynchronous one-way channels, method calls)

**Sequence diagrams** (auxiliary, showing sample system executions)

**Fig. 3.** JTN: Specification/Model Structure

The JTN aims to mimic some nice diagrammatic features of UML, but, together with possessing a completely formal basis, that is totally absent in UML, it favours some technical distinctions that we consider methodologically useful.

For example, the elements building the systems are typed by means of classes, but classes are of three kinds: *datatype*, whose instances are just values, *passive*, whose instances are standard imperative objects, and *active*, whose instances are processes. Indeed, in the underlying formal semantics the three kinds of classes have completely different formal models. Moreover, classes, of any kind, are strictly encapsulated, that is they have an interface, describing in a precise way how their instances can interact with the outside entities, and a fully private body describing their behaviour. More details about the different kinds of classes are in Fig. 4. Moreover, each specification (model) has a precise structure, described in Fig. 3.

Notice, that JTN provides an *architecture diagram* for depicting the overall structure of the system. The lack of this kind of diagram in the UML has been criticized by many authors who have then provided various alternatives; on the other hand, that kind of diagram reminds of diagrams found so useful in the so-called Modern (and Postmodern) Structured Analysis (see[31]). Another distinction w.r.t. UMLis that in JTN the use of sequence diagrams is only complementary to the one of behaviour diagrams, to provide an insight of possible scenarios. It is instead well-known that the use of sequence diagrams in UML as a basic tool is the source of ambiguities for users (see in this respect the illuminating paper on a nice substitute, the live sequence charts of [13]).

Another novelty is the way the body of an active class, and then the behaviour of its instances is described. To this aim we have introduced what we call *a behaviour diagram*, just a diagrammatic form for representing some labelled transition system also expressible algebraically in LTL, but that, differently from UML statecharts, allow to visually depict also non-reactive behaviour (see [25] for a proposal of an integration within the UML). Finally, it is almost obvious that, because of its formal basis in JTN, there is a precise framework for defining and dealing with static semantics consistency problems, what is instead a problem in the UML, as witnessed by the many papers and different tools addressing that issue [8].

### 3.2 Reformulating Current Practice Development Methods

Another line of our research consists in looking at current development practices, noticing problems and attempting at a reformulation based upon, and inspired by, related work in formal techniques.

As we have already discussed, 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 [24]) and incorporating many insights coming from the software engineering best practices. The problems that

**DATATYPE CLASS**

**instances** values

**characterized by**

- Constructors
- Operations

**interface** visible constructors and operations

**body** private constructors and operations + definitions of all the operations by (ordered) lists of conditional rules of the form ($pt_i$ patterns built with the constructors)

$Op(pt_1,...,pt_n)$ | $Cond_1$ -> $exp_1$ | .... | $Cond_m$ -> $exp_m$

**PASSIVE CLASS**

**instances** standard objects

**characterized by**

- Attributes
- Methods

**interface** visible methods

**body** attributes + private methods + definitions of all the methods by (ordered) lists of conditional statements of the form ($pt_i$ patterns built with the constructors, $stat_i$ imperative statements acting on the object local state)

$M(pt_1,...,pt_n)$ | $Cond_1$ -> $stat_1$ | .... | $Cond_m$ -> $stat_m$

**ACTIVE CLASS**

**instances** processes

**characterized by**

- Attributes
- Interactions (with the external world)
  - Write/Read on one way typed synchronous channels
  - Write/Read on one way typed asynchronous channels
  - Call of methods of passive objects

**interface**

- Used synchronous channels
- Used asynchronous channels

**body** attributes + behaviour definition by a BEHAVIOUR DIAGRAM, i.e., a graph s.t.

- nodes are characterized by a name (just usual control states)
- arcs have the form

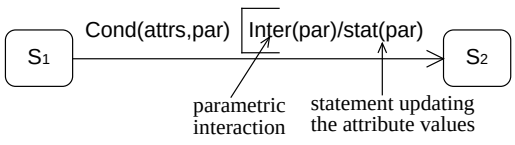


**Fig. 4.** JTN: CLASSES

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 [7, 9] we have presented a new way of structuring the Requirement Specification, 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 [27]. Essentially, it shows how the (generalized) labelled transition systems of our LTL approach [6] 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[5] [22]. Second, we have incorporated some ideas from well-known methods, such as Structured Analysis [32] and the work of some pioneer methodologists such as M. Jackson [15, 16]. From the latter in particular we have taken the total separation of the domain from the system, a distinction somewhat blurred in many object-oriented approaches; while the distinction between the system 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 system 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 system, without providing an object-oriented structuring at a stage when such a structure is not required.

We now give a short technical outline to get the flavour of the approach.

[5] `http://www.brics.dk/Projects/CoFI/`

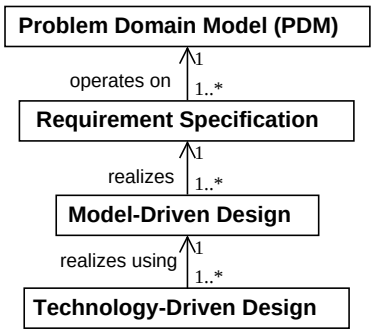


**Fig. 5.** Artifacts

The context of our work is sketchily represented in Fig. 5, where we present some essential steps (artifacts to be produced) in a modern 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. 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., [18]), namely an architecture based on the abstract modelling and independent of the implementation platform, to which is targeted the Technology-Driven Design. Notice, that a PDM artifact may be used as a starting point for many different systems, 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. We cannot deal here with all those activities; however, we intend to stress that in our approach Requirement Specification is the first activity in which the (Software) System is taken into consideration.

The Requirement Specification activity we propose assumes that the Problem Domain Modelling produces as an artifact, PDM, a UML object-oriented description of the part of the real world that concerns the System, but without any reference to the System itself nor to the problem to which the System provides a solution.

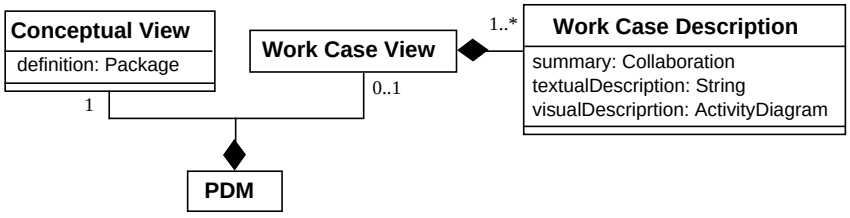


**Fig. 6.** PDM Structure

The structure of a PDM in our proposal is shown in Fig. 6. We propose to model the various entities present in the domain by the Conceptual View, a UML package including a class diagram, where the classes may be also active, thus with a dynamic behaviour, also autonomous, and not just purely reactive. Then, the most relevant cooperation among such entities may be modelled in the Work Case View part, which consists of a special kind of workflows named *work cases.*

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 a "system placement" activity, to relate the System to the domain and, by that, to locate the System boundary.

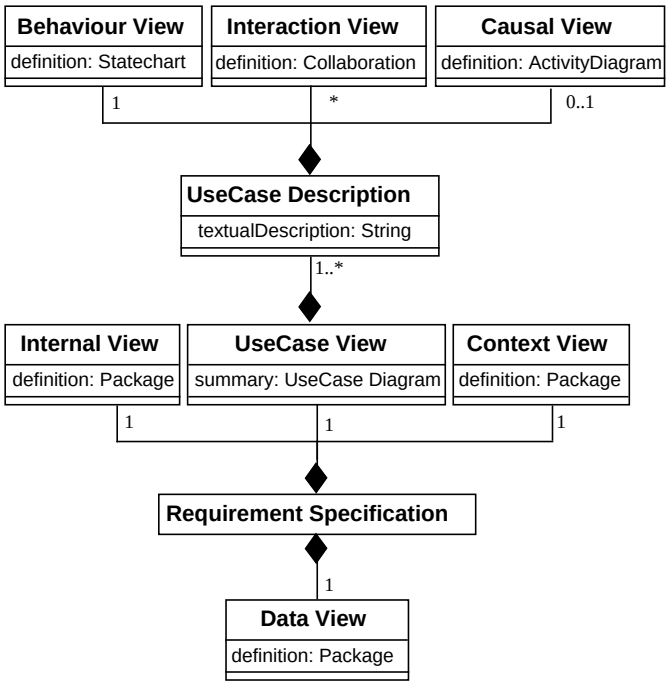


**Fig. 7.** Requirement Specification Structure

In our approach, the Requirement Specification artifacts consist of different views of the System, plus a part, Data View, which lists and makes precise all data appearing in the various views of the System to help guarantee the consistency of the concepts used in such views. Its structure is shown in Fig. 7 by a UML class diagram.

Context View describes the context of the System, that is which entities (*context entities*) and of which kinds may interact with the System, and in which way they can do that. Such entities may appear int he PDM or to be external

entities needed by the particular problem/solution under consideration. That explicit splitting between the System 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 (System) and on which we have to find (capture) the requirements The context entities are further classified into those taking advantage of the System (*service users*), and into those cooperating to accomplish the System aims (*service providers*). This 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 System, since it is sensible to require that they adapt themselves to use the System.

Internal View describes abstractly the internal structure of the System, which 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.

Use Case View, as it is now standard, shows the main ways to use the System (*use cases*), making clear which actors take parts in them.

The Use Case View consists of a UML "Use Case Diagram" and of a Use Case Description for each use case appearing in it. The actors appearing in the Use Case diagram are possible *roles* (*generic instances*) for the entities outside the System interacting with it (context entities) defined in the Context View. Thus, each actor will be denoted by a name, expressing the played role, and by a class, appearing in the Context View, showing which kind of context entities may play such role.

A Use Case Description consists of a textual presentation and of one or more views, of different kinds, of the use case.

Any Use Case Description must include a Behaviour View, which is a statechart describing the complete behaviour of the System with respect to such use case. Such statechart has particular features. Its events may be only call of the operations of the interfaces of the class corresponding to System, its conditions may test only the System Abstract State and the event parameters, and its actions may only be calls of the operations of the actors, as defined by their interfaces, or actions updating the System Abstract State.

A Use Case Description may include any number of Interaction View, which are sequence (or collaboration) diagrams representing the interactions happening in a scenario of the use case among the context entities and the System. Any Interaction View must be coherent with the Behaviour View (that is, it must represent a particular execution of the complete behaviour of System described by such view). We think that the Interaction View are really important, showing visually who does what, but they are complementary to the Behaviour View because they cannot show under which conditions the various messages may be exchanged and their effects on the System Abstract State.

A Use Case Description may include also a Causal View, which is an activity diagram describing all the relevant facts happening during the use case and their

causal relationships. The relevant facts (technically represented by action-states of the activity diagram) can be only calls of the interface operations of System by the actors, calls of the operations of the actors by System, UML actions producing side effects on the System Abstract State. Also the Causal View must be coherent with the Behaviour View, in the sense that the causal relationships among "facts" that it depicts may happen in the behaviour depicted by the state chart.

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.

## 4 Conclusions

Reflecting on our own experience, we have advocated a shift of the research in formal techniques toward a stronger engagement in issues immediately relevant for the software development practice. That is not in opposition to providing the necessary formal foundations when needed, nor to proposing new fundamental ideas, as long as they have a clear potential applicative target. Our emphasis was especially in the direction of the care for the methodological aspects. We have singled out the case of UML as paradigmatic, in our view, of what could have been achieved by our community and was not.

The illustration here of some of our current research had no ambition of showing "The way", but only the meaning of a sincere effort in trying to lighten some suffered isolation; moreover, it is an effort to exploit a significant and mature background in formal techniques, whose value we never underestimate. In that sense, we are convinced that the strategy of *Well-Founded Software Development Methods* has a lot of potential value.

We summarize the (meta)guidelines constituting the essence of that strategy and what we can gain by that.

There is a not so obvious preliminary assumption that the issues addressed are potentially relevant to software engineering practice and that, addressing them, 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 have called the formalism (not so) splendid isolation; this is a kind of first imperative.

A second imperative is the rigorous discipline coming from the area of formal techniques, that imposes both the use of semantically well-defined structures and of rigorously justified methods.

A third and last imperative is that the formal aspects should be not forced on the end-users, who in principle are not experts, but used in the background to achieve the second imperative.

To clarify by an example the three points and their benefits, in our sample activity presented in the previous sections

- we have only used those UML constructs and in such restricted way that their semantics is clear, in the sense that a formal semantics exists (that

is the case of statecharts) or need only to be worked out in detail (that is the case of active classes); from that we can precisely answer in the natural language any relevant question for clarifying ambiguities, without exposing the formal definitions;
- the method for structuring, e.g., the requirements leads to a collection of artifacts that have passed a number of consistency constraints, and, thus, they are much less prone to faults and moreover define an unambiguous body of requirements.

There is another aspect that can be seen as an aside benefit. It may happen that the disciplined rigour that we borrow from the formal techniques habit, shows the inadequacy of some proposed construct or techniques and stimulates the introduction of new ones. That has been the case of the inadequacy of sequence diagrams and the proposal by D. Harel and W. Damm of Live Sequence Charts (see [11]) and by us of behaviour diagrams [25] and of Context diagrams, as we have seen in the paper.

On the sociological side, 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 comparing 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. In conclusion we believe that *Well-Founded Software Development Methods* should be explored seriously and improved, of course, with the help of appropriate tools exploiting their hidden formal basis, as much as it happens with other engineering professions and practices (for this point see the interesting remarks in [23]).

Because of the fast and somewhat wild evolution in information technology in general and in software applications in particular, it is difficult to assess the situation and look at the right new directions of research. Still, we see some *fundamental challenges* ahead. We mention two.

The first comes from our research interests and can be summarized in the slogan *Beyond UML*. Though the UML (and the OMG initiatives) has tried to touch many important issues and satisfy some real needs, from abstract modelling (MDA) to multiview and to support for tools, we need to go beyond UML in a number of directions. The object oriented technology underpinning the UML is inadequate, as it is now recognized by many, for a real quality leap; the notation is shaking and not cleanly defined, endangering consistency; the semantic foundations are missing; also due to the lack in semantic foundations, the devel-

opment of powerful semantic tool is difficult (look at a comprehensive view of the problem by D. Harel in [13]); the support for evolution is left to the user.

The second challenge we see is in the lack of foundations and principles for some current trends in software development and applications characterized by components, open systems, web services, middleware stratifications and variety, mobile and heterogeneous access, comprehensive proprietary platforms (.Net, Java/SUN,...). Though some of those issues are nominally addressed by some current formal research, it seems to us that, as most often in the past, the way those issues are addressed, more with the obsession of mathematical elegance and sophistication than with applicability in mind, shows a little chance of making an impact on software development practices.

## References


1. 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):153–196, 2002.
2. E. Astesiano, B. Krieg-Brückner, and H.-J. Kreowski, editors. *IFIP WG 1.3 Book on Algebraic Foundations of System Specification*. Springer Verlag, 1999.
3. E. Astesiano and G. Reggio. Direct Semantics of Concurrent Languages in the SMoLCS Approach. *IBM Journal of Research and Development*, 31(5):512–534, 1987.
4. E. Astesiano and G. Reggio. SMoLCS-Driven Concurrent Calculi. In H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, editors, *Proc. TAPSOFT'87, Vol. 1*, number 249 in Lecture Notes in Computer Science, pages 169–201. Springer Verlag, Berlin, 1987.
5. E. Astesiano and G. Reggio. Formalism and Method. *T.C.S.*, 236(1,2):3–34, 2000.
6. E. Astesiano and G. Reggio. Labelled Transition Logic: An Outline. *Acta Informatica*, 37(11-12):831–879, 2001.
7. 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`.
8. E. Astesiano and G. Reggio. Consistency Issues in Multiview Modelling Techniques. Technical Report DISI–TR–03–05, DISI, Università di Genova, Italy, 2003. To appear in Proc. WADT 2002.
9. 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/ReggioEtAll03c.pdf`.
10. 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 Lecture Notes in Computer Science. Springer Verlag, Berlin, 1999.
11. W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001.
12. H. Ehrig and B. Mahr. A Decade of TAPSOFT: Aspects of Progress and Prospects in Theory and Practice of Software Development. In P.D. Mosses, M. Nielsen, and M.I. Schwartzbach, editors, *Proc. of TAPSOFT '95*, number 915 in Lecture Notes in Computer Science, pages 3–24. Springer Verlag, Berlin, 1995.

13. D. Harel. From Play-In Scenarios to Code: An Achievable Dream. *IEEE Computer*, 34(1):53–60, 2001.
14. C.A.R. Hoare. How did Software Get so Reliable Without Proof? In M.-C. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, number 1051 in Lecture Notes in Computer Science, pages 1–17. Springer Verlag, Berlin, 1996.
15. M. Jackson. *Software Requirements & Specifications: a Lexicon of Practice, Principles and Prejudices*. Addison-Wesley, 1995.
16. M. Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems*. Addison-Wesley, 2001.
17. C. Jones. Some Mistakes I Have Made and What I Have Learned FromThem. In E. Astesiano, editor, *Proc. FASE'98*, number 1382 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1998.
18. OMG Architecture Board MDA Drafting Team. Model Driven Architecture (MDA). Available at `http://cgi.omg.org/docs/ormsc/01-07-01.pdf`, 2001.
19. B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1997.
20. B. Meyer. Software Engineering in the Academy. *Computer*, 34(5):28–35, 2001.
21. M. Minasi. *The Software Conspiracy*. Mc Graw Hill, 2000.
22. 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 Lecture Notes in Computer Science, pages 115–137. Springer Verlag, Berlin, 1997.
23. D.L. Parnas. "Formal Methods" Technology Transfer Will Fail. *J. Systems Software*, 40(3):195 – 198, 1998.
24. Rational. Rational Unified Process© for System Engineering SE 1.0. 2001.
25. G. Reggio and E. Astesiano. An Extension of UML for Modelling the non Purely-Reactive Behaviour of Active Objects. Technical Report DISI–TR–00–28, DISI, Università di Genova, Italy, 2000. Available at `ftp://ftp.disi.unige.it/person/ReggioG/ReggioAstesiano00b.pdf`.
26. 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.
27. 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.
28. G. Reggio and M. Larosa. A Graphic Notation for Formal Specifications of Dynamic Systems. In J. Fitzgerald and C.B. Jones, editors, *Proc. FME 97 - Industrial Applications and Strengthened Foundations of Formal Methods*, number 1313 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1997.
29. UML Revision Task Force. *OMG UML Specification 1.3*, 1999. Available at `http://www.rational.com/media/uml/post.pdf`.
30. A.I. Wasserman. Toward a Discipline of Software Engineering. *IEEE Software*, 13(6):23–31, 1996.
31. R.J. Wieringa. *Requirements Engineering: Frameworks for Understanding*. John Wiley, 1996.
32. E. Yourdon. *Modern Structured Analysis*. Prentice-Hall, 1989.