Formalism and Method [⋆]

Egidio Astesiano – Gianna Reggio

DISI
Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova, Italy
Via Dodecaneso, 35 – Genova 16146, Italy
{ astes, reggio } @ disi.unige.it
http://www.disi.unige.it

RATIO ET VIA

Abstract. Luckily, is getting strength the view that formal methods are useful tools within the context of an overall engineering process, heavily influenced by other factors that developers of formalisms should take into account.
We argue that the impact of formalisms would much benefit from adopting the habit of systematically and carefully relating formalisms to methods and to the engineering context, at various levels of granularity. Consequently we oppose the attitude of conflating formalism and method, with the inevitable consequence of emphasizing the formalism or even just neglecting the methodological aspects.
In order to make our reflections more concrete we illustrate our viewpoint addressing one particular activity in the software development process, namely the use of formal specification techniques.

1 Introduction

1.1 Introducing the case

Giving another invited talk, ten years after, at the last edition of TAPSOFT, in an ideal relay with the next year new ETAPS-FASE, inevitably stimulates a reflection on the variations of needs, attitudes and work witnessed in the past decade.

Ten years ago, in '87, we were still in a period of great optimism on the fundamental role of theory, and consequently the value, I would say the necessity, of formal methods in designing and developing software systems. One year before, at his inaugural lecture for LFCS, the Edinburgh Laboratory for Foundations of Computer Science, Robin Milner, also an invited speaker at TAPSOFT '87, was providing the following two principles for LFCS activity:

1. the design of computer systems can only properly succeed, if it is well grounded in theory

[⋆] This work has been partially supported by the projects 40%: "Modelli della computazione e dei linguaggi di programmazione" and "Progetto di una workstation multimediale ad architettura parallela".

2. the important concepts in a theory can only emerge through protracted exposure to application.

When in November '96, at the decennial celebration of LFCS, the current Director Don Sannella was recalling those principles, some of the attendees were feeling a bit uneasy, asking themselves and colleagues whether the second principle can still be asserted on experimental grounds. Indeed, the question was implicitly reflected in Cliff Jones's speech, when he was asking about the role of theoretical investigations, in particular of semantics, in the many enormously successful software products emerged in the decade. This problem was also touched in some of the invited lectures at TAPSOFT '95. Ehrig and Mahr, surveying a decade of TAPSOFT in [12], made a mixed-feeling remark that

> "Theory and practice today have further separated and the pressure for marketable solutions and routine application has increased. But again, it seems that new technology can not be thought without the contributions from theoretical and conceptual work. The question is therefore anew what formal methods can do in the future."

Goguen and Luqi in [16] began their talk with

> "Formal methods have not been accepted to the extent for which many computing scientists hoped."

Tony Hoare in his brilliant lecture at FME 96 [18] with the suggestive title "How did software get so reliable without proof?" admits a "large gap between theory and practice".

However, the reactions to this rather common feeling are quite different, beginning with the explanations of this situation. For Hoare in [19]

> "the problem of program correctness has turned out to be far less serious than predicted. Ten years ago, researchers into formal methods (and I was the most mistaken among them) predicted that the programming world would embrace with gratitude every assistance promised by formalisation to solve the problems of reliability that arise when programs get large and more safety-critical. Programs have now got very large and very critical – well beyond the scale which can be comfortably tackled by formal methods. There have been many problems and failures, but these have nearly always been attributable to inadequate analysis of requirements or inadequate management control. It has turned out that the world just does not suffer significantly from the kind of problem that our research was originally intended to solve."

Later on rather sadly he comments in [18] that

> "false predictions and broken promises ... nowadays are needed just to maintain a declining flow of funds for research."

A completely different view is taken by Goguen and Luqi who in [16] maintain that

"Failures of large software development projects are common today, due to the ever increasing size, complexity and cost of software systems. Although billions are spent each year on software in the US alone, many software systems do not actually satisfy users' needs. Moreover, many systems that are built are never used, and even more are abandoned before completion. Many systems once thought adequate no longer are."

Their view is very much in line with those in [15], the article "Software's Chronic Crisis" reporting on a second NATO workshop in '94 on the title issue.

"Studies have shown that for every six new large-scale software systems that are put into operation, two others are cancelled.
The average software development project overshoots its schedule by half; larger projects generally do worse. And some three quarters of all large systems are "operating failures" that either do not function as intended or are not used at all."

The failure of Arianne 5 in June '96, (after Hoare' speech at FME'96), with the careful explanation in the conclusions of the inquiring committee, was a spectacular (but exceptional ?) confirmation of this statement.
The discrepancies are not weaker when coming to draw the consequences. For Hoare in [18] rather drastically

"The final recommendation is that 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 ?"

And in [19] he proposes the "unification of theories" as the main "Challenge for Computing Science". Hoare's views are far from exotic and touch, from a particular angle, some deep truths; however he seems to discourage a close involvement of researchers in formal methods in the technology transfer process:

"...there are still grounds for hope. But this hope should be based on a more realistic appreciation of the proper and realistic timescales for technology transfer, which in every mature engineering discipline is measured in decades or centuries."

There is however a large number of other researchers who take a more positive approach, beginning with recognizing some mistakes in the promotion of formal methods. In the '89 edition of [24], a widely known book on SE, together with a significant support for formal methods, we find the following remark, which sounds particularly sad today.

"Some members of the computer science community who are active in the development of formal methods misunderstand practical software engineering and suggest that software engineering can be equated with the adoption of formal methods of software development. Understandably, such nonsense makes pragmatic software engineers very wary of their proposed solutions."

In the very informative announcement [23] of the '94 Monterey Workshop, on Formal Methods for Computer Aided Software Development we find the remark that

"The excessive optimism of the attitude that that everything important is provable helps to explain the excessive pessimism of the attitude that nothing important is provable."

The same overall problem has been addressed retrospectively by Christiane Floyd in her invited talk at TAPSOFT 95 [14], where she remarks that the survey by Ehrig and Mahr in [12]

"shows that many of the original claims associated with formal methods could not be fulfilled. Thus, the success reported rests on restating more realistic claims with respect to formal methods ..."

This is echoed in [12] itself, where Ehrig and Mahr, reporting on HDMS, an interesting concrete experimental application of formal methods, conclude that

"the experience around HDMS shows both advantages and difficulties of formal methods in software development and hints at ways of further research and at the same time teaches the limitations of formal methods in regard to the overall task of software development."

Indeed what is emerging now in recent years is a diverse attitude viewing the software (system) development process as an overall engineering process into which formal methods can play a useful, not always prominent, role. On this view converge many of the authoritative citations reported in [15]. For Goguen and Luqi in [16], in line with [14],

"One major problem has been that formal methods have not taken sufficient account of the social context of computer systems."

From another perspective in [23] we find:

"Formal means definite, orderly, and methodical, and does not necessarily entail logic or proofs of correctness ... we believe this is the most appropriate sense for the word formal in the phrase formal methods."

We are among those who share the above attitude and, together with some other deep causes for the slow success of formal methods, we consider a major one the little concern of researchers about transfer issues, as indicated in the NIST survey [11].

Our talk will try to address what we see as a potential problem for the transfer issue, namely the excessive emphasis on formalism w.r.t. method which sometimes leads to conflate the two things, always at the expense of the method. This danger is also reflected in [7], the editorial of Broy and Jones for the 1996-8 issue of Formal Aspects of Computing where they warn that

"nor can the role of formal methods work be to develop branches of mathematics which only bear a superficial resemblance to the needs of computer science"

"the role of formalism must be to help design better systems and ensure that they are put on a firmer footing."

In a straight way the difference of attitudes is explained in [14]:

> "I suppose that from the formalist point of view the main point of interest here is the use of formal concepts in dealing with a practical problem. But from the human activity point of view, a formalized procedure is implied, prescribing at what time and for what purposes these concepts are supposed to be worked within software development projects. When and how this can or must be done, makes the difference."

Ideally our talk is in the line of continuing the dialogue, proposed in [14], between promoters of formal methods and experts/researchers in software engineering practice. Moreover we see it as an opportunity for contributing to the shift of emphasis from FASE as Formal Aspects of SE to FASE within ETAPS as Fundamental Approaches to SE.

1.2 Stating our aims

Sometimes it is illuminating to go back to the origin of the word and this is indeed the case: "method" come from Greek and means "way through"; the Latin substitute for it quite significantly is "via et ratio" but also "ratio et via", both conveying the meaning of "something rational with the purpose of achieving something, together with the way of achieving it". Looking at what happens, practice and literature, one often gets the impression that only either "ratio" or "via" is left.

Nowadays the suggestion of more closely connecting formalisms to methods is more or less explicit in many papers and books and it is not our intention to repeat warnings and suggestions, often more authoritative. Moreover let us clarify that by formal method here we do not mean at all just a comprehensive method for software development, but also one addressing a specific aspects of software development.

Here we want to advocate few peculiar points:

- a formalism does not provide a method by fiat; in principle a formalism can be associated with different methods or lead to no useful method at all; thus we propose to regard the "method", which includes a formalism, as the appropriate target of investigations concerned with formal aspects of software engineering; we even suggest to investigate the appropriate use of description patterns for presenting methods;
- in order to get and/or understand a method it is essential to locate it within the context of the overall development process, in particular defining the kind of activity in the context and the target it addresses;
- a rationale should be mandatory; but "rationale" should mean something much more precise than just some accompanying words of explanation;
- a clear picture of the purely formal and methodological parts (the various aspects of the mentioned pattern) is an essential tool for analysing and relating different methods;
- finally, at the metalevel, we believe that the study of methodological aspects of formal methods is in itself an interesting target of useful investigations and can be pursued with scientific rigour.

Our points come out of some years of experience in formal specifications and not in investigations on methodology. Thus on one hand we have not enough experience for handling with the above issues in general, nor for addressing aspects far from our experience. On the other hand we feel that addressing one particular rather well-known activity, namely the production of formal specifications, we can make our points more concrete and understandable. However we feel that some of the ideas presented in this paper can be exploited in some generality in relation to other aspects of the software development process.

Thus we first present a "pattern" for analysing a formal specification activity; then we provide some illustrative examples of analysis on that basis; finally we briefly discuss how to relate methods. Both for lack of room and for purpose (we hope to be read by people outside of the community of formalists) our style will be quite informal and sketchy. A more complete presentation, with some more rigorous discussions, especially on relating methods, is in a full paper [5].

We hope to be able to address other significant activities in some near future but also we much encourage other researchers to work on the issue. Finally, we invite the reader to consider this paper mainly as stimulating a debate and further research more than proposing definitive conclusions/solutions.

2 A Pattern for Specification

2.1 Locating the method within the development process context

Preliminaries We illustrate our points by analysing, as a case example, the problem of providing a formal specification. We use some generic assumptions about the software development process, without any commitment to a particular process model. For general references see [24, 26] and [13] for a specific treatment of process modelling.

- A development process will return at the end some kind of product (*end product*); thus for each development process we may qualify what is the kind of its end product. Notice that the end product may be pure software, as a program for statistic analysis, or a whole system having also non-software parts, as an information system (which may have as components the clerks using it) or an embedded system (which may have as components some controlled mechanic/ electronic devices). For the purpose of the current presentation our concept of end product will abstract from the features specific of the application domain. Domain knowledge and analysis is of paramount importance in practice, but it is not considered here, also because their role has not yet been investigated enough at the methodological level in connection with the use of formal methods.
  In Tab. 1 we present a list of keywords qualifying end products currently found in the literature. Some items are enough standard and well-understood, while other are rather ambiguous (marked by $*$) and others may be just variants used in some particular community (marked by $+$). Each of them has been found in papers presenting formal methods.
- A *development process* is a collection of some *activities* with temporal/causal relationship among them; furthermore there are "super/meta" activities concerning the definition and management of the development process. In Tab. 2

C/C++ programs	∗ Reactive systems
Ada programs	Real-time programs
Imperative programs without pointers	Real-time systems
Imperative programs with pointers	Hybrid programs
Imperative programs	Hybrid systems
Functional programs	∗ Dynamic systems
Functional modules/data types	Object-oriented programs
Nondeterministic programs	Object-oriented systems
Programs in an asynchronous language	Protocols
∗ Parallel programs	Information systems
∗ Distributed programs	Database systems
∗ Distributed systems	Embedded systems
∗ Distributed architectures	+ Agent systems
∗ Concurrent programs	. . . . . .
∗ Reactive programs	

Table 1. End products of a development process

we present a tentative list of possible activities. The items in this list have been found in papers about software engineering.

To give a requirement specification
To validate a requirement specification
To give a design specification
To validate a design specification
To verify a design specification against a requirement specification
To give an intermediate specification (those not classifiable as requirement or design)
To validate an intermediate specification
To verify an intermediate specification against some other specification
To give some code
To validate some code
To verify some code against a design/intermediate specification
To check the *quality* of some specification/code
To reuse (replay) [a part of] a development process, or just an activity by changing something in the "inputs"
To produce a new version of the end product (maintenance)
To support the development process definition and management
. . . . . .

Table 2. Activities in a development process

- Each activity at the end will return some "products" (specification, code, documentation, a development process, etc.).
- Some activity may require as mandatory inputs some "products" which are the results of other activities.

– A *method (formal method)* is a way to perform an activity of a particular kind (supported by “formal techniques/tools”).
– In a very general way a *specification* is a description of (possibly some aspects of) an end product at some level of abstraction, which can be also intended as at some point in a development process.

In the following we will consider only the generic task of providing a formal specification. We will outline, so-to-speak, a “pattern” (in a broad sense, in the line of [1] and followers) for qualifying a formal specification method; our pattern illustrates in particular the relationships between formalism and method. A warning: we do not intend to be prescriptive; the paper has the main purpose of exploring some ideas and of stimulating a reflection; much has still to be clarified. The structure of the pattern is shown in Fig. 1.

CONTEXT

END PRODUCT	$\mathcal{EP}$	the end products of the development process
ACTIVITY		location of the activity in a development process

FORMALISM

FORMAL MODEL	$\mathcal{M}$	mathematical structures representing the end products
SPECIFICATION	$\mathcal{SPEC}$, $[\![\_]\!]$	specifications as artifacts

$\uparrow$

IMPACT ON METHOD

$\downarrow$

PRAGMATICS

RATIONALE	$\mapsto$	how the formal models model the end products
GUIDELINES		guidelines for the specification task
PRESENTATION		presentation of the specifications for humans
DOCUMENTATION		documenting the specification activity

Fig. 1. Aspects and components of a specification formal method

End product Since a specification method supports the activity of giving a description of some kind of end product, we have to qualify the kind of such end products.

The END PRODUCT part is expressed by qualifying the set of the considered end products, denoted by $\mathcal{EP}$. Generally speaking the description of the set is not formal. For our discussion we assume the existence of an oracle for deciding whether or not some end product is in $\mathcal{EP}$ or not, for every $\mathcal{EP}$. End products will play a major role in relating formalism to methods, as we are going to illustrate.

Quite often end products are structured, i.e. they exhibit an inner structure.

Activity We need to qualify the kind of specification we are dealing with and its place within the development process we are using. We stress the importance of locating an activity within its context.

A quick look at standard books on SE ([24, 26], e.g.) or to the various papers on development process models (see [13]), will show the reader the many ways "specification" is intended and the different roles in the process.

For example, the activity of giving the requirement specification may be used in a classic waterfall or spiral model; the activity of giving an intermediate specification may be used either in a uniform multistage model or in an intermediate step between design and code; within an object-oriented approach the distinction between requirement and design is blurred and the activity is much constrained by the specific approach.

This information allows also to know whether the formal method is part of a uniform/coordinated group of other formal methods to support the whole development process.

The parts/aspects END PRODUCT and ACTIVITY should allow to have a coarse idea of the "functionality" of the formal method.

2.2 Formalism

Formal model The *formal models* are a class of mathematical (set theoretic) structures $\mathcal{M}$, which formally represent the elements in $\mathcal{EP}$ at some abstraction level, depending on the kind of specification we are providing.

In this paper, we denote them by the words "formal models" to avoid confusion with the models of some logic and with the development process models.

Very well-known classes of formal models used by some formalism are:

- computable functions from memories (maps from locations into values) into memories for imperative programs;
- many-sorted algebras or first-order structures for functional modules and data types;
- synchronization trees (see, e.g. [22, 20]) for processes;
- sets of action traces (see, e.g. [17]) for processes.

Strangely enough, in several cases we find that this part is either obscure or given implicitly; instead, in our opinion, it should be given explicitly and in a very clear way.

Most often the formal models are classified into disjoint subclasses by considering structural/syntactic properties using a general concept of signature, as when using institutions (see e.g., [8]). Following this view we need to give:

- a class of signatures $\mathcal{SIG}$,
- for each Σ element of $\mathcal{SIG}$ the class of the formal models on that signature $\mathcal{M}_\Sigma$.

Sometimes the formal models are structured, i.e. exhibit an inner structure.

Specifications In a very general way a *specification*, as an artifact, is a description of an end product at some level of abstraction, which can also be intended at some point in the development process.

A *formal specification* is a way to determine a class of formal models: all those modelling the end product at such point in the development process.

Usually formal specifications are expressed by terms/programs in an appropriate *specification language*.

The specification component of a formal method consists of:

– a set of specifications $\mathcal{SPEC}$ (programs/terms of the specification language);
– and a semantic function $[\![\_]\!]$ (for the specification language), associating with each specification a class of formal models.

Notice that there are no assumptions on the cardinality of $[\![\mathrm{SP}]\!]$; it may be just a singleton.

$[\![\_]\!]$ must be a total (non-injective) function, whenever $\mathcal{SPEC}$ contains only the admissible specifications.

$[\![\_]\!]$ may be non-surjective: only some classes of formal models may be expressed using this specification language. The specification language is more or less powerful depending on how is large the codomain of $[\![\_]\!]$.

If the formal models are classified by signatures, then the specifications must have the form of pairs, whose first component is a signature, and the semantics will be a class of formal models on such signature.

2.3 Impact of formalism on method

We outline the impact that some features of a formalism may have on the method and thus on pragmatics; conversely some requirements on pragmatics have to be taken care in developing a formalism. Here we deal only with some aspects of the specification languages; in the full paper ([5]) we also analyse the role of formal models in this respect.

Structuring It is important to distinguish two different kinds of structuring:

Specification structuring A reasonable specification language should provide ways to modularly present complex specifications, by allowing to split them in sensible pieces, also to help maintenance and reuse, but these constructs are linked neither to the formal models nor to the end products. The importance of this kind of structuring has been widely recognized since early times, as witnessed in the various specification languages (see [28]).

End product structuring As indicated, sometimes the end products and/or the formal models are structured. A good specification language should offer ways to express this kind of structure, possibly avoiding to confuse it with the above one. A typical example is a combinator for parallelism (contrasted with a mechanism for incremental specification building, like enrichment or inheritance).

Abstraction level Once we have given the formal models, we can qualify the *abstraction degree* of the specification language in the sense how much abstract its specifications can be, an so providing some information about at which points in the development process it may be used. The abstraction degree is related to the cardinality of the classes of formal models which are semantics of the specifications.

Semantics The technique used for providing semantics is not neutral. The semantic of a specification language can be given in:

– a rather direct/explicit and denotational way (e.g., as done by Hoare for *CSP*, [17]), by exhibiting the relative class of formal models;
– an indirect/implicit way, say as the limit of a diagram in some category (1) or defining two programs/specifications semantically equivalent iff their equality may be proved by a deductive system (2).

Techniques as (1) may be used as a quick way to establish the existence of such semantics, but after a direct characterization has to be provided; while those as (2) may be used to help work with the specifications, in order to provide simpler forms. However, in our opinion, providing an explicit way seems to be essential for SE purposes.

Style There are various specification styles. The most quoted distinction is between axiomatic/property-oriented and model-oriented; still other hybrid styles are possible.

Property-oriented (axiomatic) We prefer property-oriented, as more suggestive.

In general property-oriented specification methods use formal models classified by signatures. The ingredients are (see the concept of institution for a more general setting, also accounting for change in signatures, [8]): for each $\Sigma \in \mathcal{SIG}$,

– a set of sentences over Σ, $\mathcal{SEN}_\Sigma$;
– a validity notion, i.e. a binary relation $\models_\Sigma \subseteq \mathcal{M}_\Sigma \times \mathcal{SEN}_\Sigma$.

Specifications in this case are pairs, whose components are a signature and a subset of $\mathcal{SEN}_\Sigma$.

For what concerns the semantics, the basic way to define it is:
$[\![(\Sigma, \mathrm{S})]\!] = \mathit{Mod}((\Sigma, \mathrm{S}))$
where
$\mathit{Mod}((\Sigma, \mathrm{S})) = \{\mathrm{M} \mid \mathrm{M} \in \mathcal{M}_\Sigma \text{ and } \mathrm{M} \models_\Sigma \phi \text{ for all } \phi \in \mathcal{SEN}_\Sigma\}$[2].
The methodological ideas supporting this specification style are:

> *we describe the end product at a certain moment in its development by expressing all its "relevant" properties by formulae of the used logic.*

Clearly this aspect will have an enormous impact on the use of the formalism, as it should be reflected in the guidelines. In the presentation part, the formulae of the

[2] The elements of this class are usually called the models of the specification or of S.

used logic should be intuitively described by using the natural language in terms of properties of the formal models and via the rationale in terms of properties of the end products.

A property-oriented specification language may be evaluated by considering:

expressive power: how many/which are the classes of $\mathcal{M}$ which can be expressed by these specifications;

adequacy: which properties of the end products may be expressed by these specifications.

As an example, consider the specification languages μ-calculus and UNITY ([9]). The first has a big expressive power and a low adequacy for specifying protocols; indeed, it is hard to qualify its combinators in terms of properties on protocols. The latter is not very expressive, but it is quite adequate for nondeterministic imperative programs (its end products); indeed its few combinators correspond to basic relevant properties on them.

Model-oriented The ingredients for model-oriented specifications are:

- a class of specifications (a specification language) $\mathcal{SPEC}$;
- a basic semantic function: $[\![\_]\!]' \colon \mathcal{SPEC} \to \mathcal{M}$ (i.e. associating essentially one model with one specification);
- a partial order on $\mathcal{M}$: $\succeq$.

Then the semantics is defined by:
$[\![\mathrm{SP}]\!] = \{\mathrm{M} \in \mathcal{M} \mid [\![\mathrm{SP}]\!]' \succeq \mathrm{M}\}$
The methodological ideas supporting this specification style are:

> *we describe the end product at a certain moment in its development by giving a prototype/archetype of it using the specification language; then apart we say which are the irrelevant features of this archetype by the order $\succeq$ (*$\mathrm{M} \succeq \mathrm{M}'$ *means that* M' *differ from* M *for irrelevant details, which can thus be freely fixed later in the development).*

Perhaps, a better way to name this style should be *construction-oriented*, with the meaning that we specify an end product by construction (at the abstraction level supported by the method, i.e. depending on the formal models and on the specification language); afterward we would say when another construction may be equivalent.

If $\succeq$ is the identity, then we have a purely constructive specification style, the lowest level in a classification by abstraction degree.

A model/construction-oriented specification language may be evaluated by considering:

expressive power: how many/which are the formal models which can be expressed by $[\![\_]\!]'$ and how many/which are classes of $\mathcal{M}$ which can be expressed using these specifications;

formal model or end product-oriented: the formal model-oriented specification language may be further classified depending on whether their constructs are oriented towards the features of the formal models (e.g. $\_+\_$ and $\_.\_$ of *CCS*) or towards the end products (e.g. the LOTOS constructs for protocols).

A formal model-oriented specification language is more general and can be used in several different formal methods considering different classes of end products (think of λ-calculi); but it may be not very flexible and convenient for special classes of end products (it is possible to model any imperative program by using λ-calculi, but it is not sensible for useful purposes in practice). On the other hand, the end product-oriented specification languages could be used for very successful formal methods for particular classes of end products, and cannot easily nor sensible be adopted for very different end products (e.g. it is not convenient, if possible at all, to use *CCS* to specify fully distributed systems).

Some controversy between property and model-oriented has been and it is still going on, on various grounds. Perhaps different styles serve different purposes and different communities.

Borderline cases Sometimes, in an property-oriented specification formalism we have also another ingredient: a way to select one (some) special element out of the model class by additional properties, which cannot be expressed by using the formulae (constraints). In these cases the semantics is given by:

$[\![(\Sigma, \mathrm{S})]\!] = \{\mathrm{M} \mid \mathrm{M} \in \mathit{Mod}((\Sigma, \mathrm{S}))$ and ... additional constraints ... $\}$

Usually, we need to give some restrictions on (Σ, S) in order to have that the $[\![(\Sigma, \mathrm{S})]\!]$ is not empty.

Most typical examples are the observational and initial semantics; in the first we pick up a class of models, considered equivalent w.r.t. a set of observations; in the second we defined essentially one model on the basis of an induction principle in defining the individual elements of the model, plus an equality defined by logical deduction.

If the constraints pick up a single model, then a specification formalism given in this way is both property-oriented, we give the/some properties of the end product, but in the same time is model/constructive-oriented, since we build up in the end one model.

2.4 Pragmatics

Rationale In order to provide a rationale for why some end products have been given some specifications, and thus a basis for validation and comprehension, a method should provide the connection between the formal models and the end products it is addressing. On the basis of some years of experience, we believe this to be a fundamental aspect, whose importance is unfortunately often underestimated.

Let us provide some suggestions, at the risk of some oversimplification, on how to handle this issue in a somewhat rigorous way.

Essentially we must provide the means for establishing a binary relation "$\mapsto$" between end products and formal models, where $\mathrm{P} \mapsto \mathrm{M}$ means intuitively that P is modelled by M (or M is a model for P or M models P). In general $\mapsto$ is not injective; this is sound, since the formal models cannot, and should not, cover all aspects of the end products, and so several end products may be modelled by the same formal model. Also the codomain of $\mapsto$ may be a subclass of $\mathcal{M}$; in such cases we have more formal models than we need, but that is not a problem.

The domain of $\mapsto$ should coincide with $\mathcal{EP}$; otherwise the formal method considers only a part of the end products, and thus it is better to change the definition of $\mathcal{EP}$.

Assuming $\mapsto$, we can then formally define a connection pair $(\mathcal{A}, \mathcal{I})$ between end products and formal models:

- for every set of end products Ps, $\mathcal{A}(\mathrm{Ps})$ is the class of formal models M s.t. for some $\mathrm{P} \in \mathrm{Ps}$, $\mathrm{P} \mapsto \mathrm{M}$;
- for every class of models Mc, $\mathcal{I}(\mathrm{Mc})$ is the set of the end products P s.t. for some $\mathrm{M} \in \mathrm{Mc}$, $\mathrm{P} \mapsto \mathrm{M}$.

We call $\mathcal{A}$ *abstraction* of end products and $\mathcal{I}$ *interpretation* of formal models and assume that if $\mathrm{P}' \mapsto \mathrm{M}'$, $\mathrm{P} \mapsto \mathrm{M}'$ and $\mathrm{P} \mapsto \mathrm{M}$, then also $\mathrm{P}' \mapsto \mathrm{M}$; graphically

$$\begin{tikzcd}
\mathrm{P}' \arrow[r, dash, "\mapsto"] \arrow[dr, dash, "\mapsto" near start] & \mathrm{M}' \\
\mathrm{P} \arrow[r, dash, "\mapsto"] \arrow[ur, dash, "\mapsto" near start] & \mathrm{M}\ ;
\end{tikzcd}$$

then this amount to say that

a) $\mathcal{I}(\mathcal{A}(\mathcal{I}(\mathrm{Mc}))) = \mathcal{I}(\mathrm{Mc})$
b) $\mathcal{A}(\mathcal{I}(\mathcal{A}(\mathrm{Ps}))) = \mathcal{A}(\mathrm{Ps})$

Moreover we have to require some consistency with the semantics of specifications, namely the semantics to be closed w.r.t. $\mapsto$: if M and M$'$ both models P and $\mathrm{M} \in [\![\mathrm{SP}]\!]$, then $\mathrm{M}' \in [\![\mathrm{SP}]\!]$, thus further constraining **a)** to have also

c) $\mathcal{A}(\mathcal{I}([\![\mathrm{SP}]\!])) = [\![\mathrm{SP}]\!]$.

Most often it will be sensible to have a (partial) equivalence relation $\sim$ on formal models, with the intuitive meaning of being "essentially equivalent" in representing an end product, thus requiring the relation $\mapsto$ to be compatible with $\sim$ (i.e.: if $\mathrm{P} \mapsto \mathrm{M}$, $\mathrm{P} \mapsto \mathrm{M}'$, then $\mathrm{M} \sim \mathrm{M}'$; and if $\mathrm{M} \sim \mathrm{M}'$, $\mathrm{P} \mapsto \mathrm{M}$, then $\mathrm{P} \mapsto \mathrm{M}'$).

Under this assumption $\mathcal{A}$ associates with each end product essentially one model (an equivalence class); thus if Mc is closed w.r.t. $\sim$, then $\mathcal{A}(\mathcal{I}(\mathrm{Mc})) = \mathrm{Mc}$ and also **a)** and **b)** hold together with **c)**, if we require, as it should, the semantics to be closed w.r.t. $\sim$.

It is worthwhile noticing that such a $\sim$ always exists, under our assumption defined by $\mathrm{M} \sim \mathrm{M}'$ iff there exists P s.t. $\mathrm{P} \mapsto \mathrm{M}$ and $\mathrm{P} \mapsto \mathrm{M}'$.

The following three items in our pattern are only briefly qualified, but our brevity should not be taken as a sign of scarce relevance. From our experience we firmly believe that they are rather fundamental for the practical acceptance of a formalism. However, we have not much room here for such important parts and moreover their relevance is luckily becoming more and more recognized.

Guidelines This part consists of the guidelines for steering and helping the task of producing in the best possible way the specifications of the end products. These guidelines should consider also the use of software tools.

The guidelines are understandably driven by the preceding parts in our pattern, but notice the fundamental role played by context and rationale, if we want seriously provide professional guidelines.

Presentation We mean by this the interface with the user, in a broad sense, of a specification product. Users, here, can range from the clients, those financing the end product, who need to understand a requirement specification in its own language (see [24], distinguishing requirement definition from requirement specification), to the implementors, to the specification builder herself/himself, when a change is needed at some later stage. A presentation should hopefully consists of text, with formal and natural language parts, graphical interfaces and animation. A presentation can influence the formalism, which should demonstrably be compatible with sensible friendly presentations.

Documentation We refer to documenting the specification process for use in evolution and maintenance. The evolution in software development is now taken care in every process model (see [13]) and its importance in formal methods recognized (see [16]) also some prototype support tools are appearing ([25]).

3 Analysing and Relating Methods

The pattern we have outlined for relating formalism and method also provides a key for analysing and relating formal methods or just formalisms.

We will first give few illustrative examples, exploring the relevance of methodological aspects. Then we touch the issue of relating methods.

3.1 Some illustrative cases

Methods based on *CCS* *CCS*, the calculus of communication systems (see [22]), has been introduced originally as a formalism for describing reactive/concurrent systems, in close analogy with the role of λ-calculus for sequential computations. Together with *CSP* (see [17]) it has been recognized as a major theoretical advance in concurrency and has provided a basis for some derived methods.

It is very interesting to explore the differences between the original *CCS* formalism and its use in a method. We will pick up two particular methods, among the many possible, based on *CCS*, used in practice and shown in the literature.

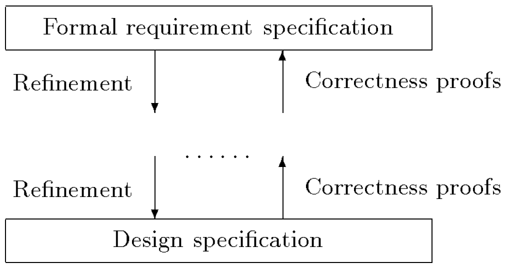
END PRODUCT: Dynamic systems (reactive, concurrent, parallel).

FORMAL MODEL: Let us consider here, for simplicity, as models the synchronization trees, i.e. labelled transition trees modulo strong bisimulation. A variety of other choices, usually variations of strong bisimulation, are possible, not always easily definable in an explicit way (see e.g. [22]).

RATIONALE: A dynamic system D is modelled by a synchronization tree, where the nodes in the tree represent the intermediate (interesting) situations of the life of D and the arcs of the tree the possibilities of D of passing from a state to another one. It is important to note that

– here an arc (a transition) $s \xrightarrow{l} s'$ has the following meaning: D in the state s has the *capability* of passing into the state s' by performing a transition, where label l represents the interaction with the external (to D) world during such move; thus l contains information on the conditions on the external world for the capability to become effective, and on the transformation of such world induced by the execution of the action; so transitions correspond to *action capabilities*;
– the precise form of the states is irrelevant, only the action capabilities starting from them matter, and so two states can be distinguished only if they have different action capabilities.

ACTIVITY: *CCS* can be used both for defining requirements (say **CCS-R**) or design (say **CCS-D**) in a fragment of the development process represented by:



SPECIFICATION: **CCS-R** specifications follow a model-oriented style. Every specification consists of a so-called behaviour expression, i.e. a term in the *CCS* language.

The basic semantics of *CCS* is the standard strong bisimulation (see [22]), i.e. it gives the synchronization tree associated with a behaviour expression.

The $\succeq$ relation is the weak bisimulation preorder; weak bisimulation means forgetting irrelevant (not all) internal moves in a synchronization tree; $t_1 \succeq t_2$ iff t_1 is weakly simulated by t_2.

The specification language, *CCS*, offers both formal model-oriented constructs (_._, _+_) and end product-oriented constructs (_||_).

For what concerns structuring constructs, we have the "rename" construct, which helps structure the specifications, and _||_ which allows to follow the end product structure. Sometimes the latter must be used for structuring the specification (the specification of a simple sequential process may be expressed as the parallel composition of several smaller processes).

The specification for **CCS-R** are similar; the only difference is that in this case the $\succeq$ relation is the identity.

Methods based on "algebraic specifications" We consider the classical ADT method, say **CADT**, see [27], originally devised for specifying abstract data types, the SMOLCS method for requirement specifications, say **SMoLCS-R** (see [2, 10]), and the method exemplified by M. Bidoit et al. in their treatment of the steam boiler problem (see [6]), that we call here **ASSRS**, for Algebraic Specification of Sequential Reactive Systems. Strikingly enough, in all cases, the underlying formalism is essentially the same.

FORMAL MODEL: (Isomorphism classes of) First-order structures with equality, usually many-sorted.

SPECIFICATION: in any case the specification style is property-oriented and the specification language allows structured versions of first-order many sorted logic with equality (*PLUSS* for **ASSRS** and *METAL* for **SMoLCS-R**). Here we consider the simplest version of **SMoLCS-R**: the one based on first-order logic; there are several variants where the logic is extended with combinators of either temporal or modal or deontic logic in order to to express liveness and safety properties on the behaviour of the dynamic systems, see e.g. [10].

The differences among the considered methods become evident only looking at the methodological aspects, in particular at the end products and at the rationales.

END PRODUCT: In **CADT** the end products are the usual, static so-to-speak, data types (lists, stacks, bulletin board, etc.); for **ASSRS** the sequential reactive dynamic systems and for **SMoLCS-R** the reactive concurrent dynamic systems.

RATIONALE *for* **CADT**: Trivial: carriers and interpretations of operations/predicates represent respectively the values (classified by types) and the operations/tests for handling them.

RATIONALE *for* **ASSRS**: A sequential system receives/sends information from/to the external world. Thus, it is modelled by a function which given a set of input messages (information from outside) and its actual state returns a new state and a set of output messages (information for outside).

The signature of the associated algebra will have the sorts "set of input messages", "set of output messages", "state" and two operations with functionality

"set of input messages" $\times$ "state" $\rightarrow$ "set of output messages"

and

"set of input messages" $\times$ "state" $\rightarrow$ "state",

respectively. These functions allow to represent the activity of the system.

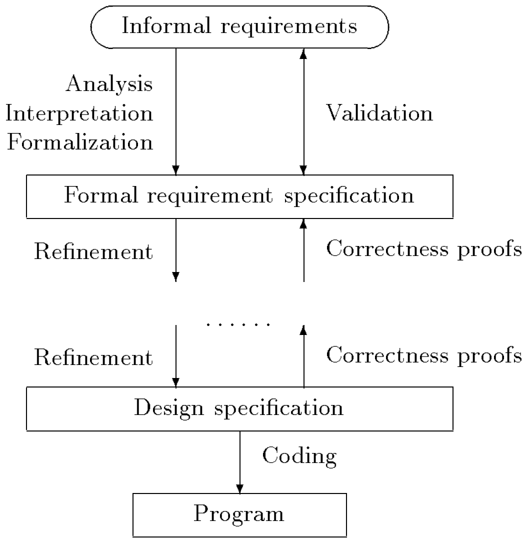
RATIONALE *for* **SMoLCS-R**: Part of the rationale is supported at the syntactic level, where some of the sorts are qualified as *dynamic* and are s.t. for each of them, say *ds*, there exits a corresponding sort of labels *l_ds* and a labelled transition predicate $\_ \xrightarrow{\_} \_ : ds\ l\_ds\ ds$; this is reflected in the models.

Given an algebra L, each one of its dynamic sorts, say *ds*, determines the labelled transition system $(\mathrm{L}_{ds}, \mathrm{L}_{l\_ds}, \_ \xrightarrow{\_} \_^{\mathrm{L}})$ representing a type of dynamic systems.

The interpretation is like for **CCS-R** and **CCS-D** with three important differences: everything can be typed; states may be relevant and, more importantly, the **CADT** method for static structures is embedded. Notice that in this way labels may have states as subcomponents, thus allowing to express also the so-called higher-order dynamic systems.

Clearly, we can handle in this way also structured dynamic systems; i.e. systems having components which are in turn other dynamic systems; in these cases we have algebras with several sorts corresponding to states and labels, together with the associated transition predicates.

ACTIVITY: All these methods cover the formal specification of the requirements:



SMoLCS-D This is the SMoLCS method for "design" specifications; it shares all components with **SMoLCS-R** except, obviously, activity and specifications. Its specifications follow a borderline style using many-sorted first-order conditional logic (see [3]), plus the constraint on the models picking up the initial element, exactly one, modulo isomorphism.

Rewriting Logic (RL) Rewriting logic, shortly *RL* (see [21]), is a formalism paradigmatic for understanding the role of the methodological aspects: apparently small variations in the formalism may cause strikingly big differences.

Being apparently based on the definition of transition systems, with the possibility of defining combinators like those for parallelism and the like, it resembles *CCS* and, because of its algebraic setting, the version of SMoLCS for design specifications, **SMoLCS-D**. But a careful analysis, following our pattern, of the method associated to *RL*, say **RL**, reveals the differences.

END PRODUCT: Non-reactive (closed) dynamic systems.

FORMAL MODEL: The formal models are classified by signatures (many-sorted first-order without predicate symbols).

A Σ-formal model is essentially a Σ-algebra A plus a transition system $(STATE, \_ \Longrightarrow \_)$, where $STATE = \cup_{s \in Sorts(\Sigma)} \mathrm{A}_s$ and the transition relation $\Longrightarrow$ satisfies particular conditions: if $s \Longrightarrow s'$, then s and s' are of the same sort, it is reflexive, transitive and closed by congruence w.r.t. the operations of Σ; moreover the transitions are decorated by additional information about their structure in terms of other transitions.

Here we have given a concise set-theoretic presentation of the RL models (see [4] for a complete presentation), but notice the original one, in [21]), adopts the language of category theory.

RATIONALE: The elements of the carriers of a formal model correspond to intermediate states in the life of (types) of dynamic systems as for CCS and SMoLCS methods, but the interpretation of the transitions is very different.

First of all, here transitions are not labelled and so there is no idea of interaction with the external world. Indeed, $s \Longrightarrow s'$ with its additional information i represents a (either partial or complete) behaviour of the dynamic system and i gives information on the structure of such behaviour (e.g., it is the sequential composition of two other partial behaviours).

SPECIFICATION: The specifications of **RL** follows a borderline style using a combination of equational logic on the operations of the signature and of conditional rules for defining the transitions, plus an initiality constraint.

3.2 Relating Methods

Here we briefly outline some interesting ways of relating methods exploiting the concepts introduced so far. A more rigorous and comprehensive treatment is in the full paper [5].

We distinguish between replacing a method with another one and by translating its formalism into another one, getting a new method.

Assume we have two specification methods, say FM and FM', given following the pattern of Sect. 2. The relevant components, for the issues we are considering here, are respectively $(\mathcal{EP}, \mathcal{M}, \mathcal{SPEC}, [\![\_]\!], \mapsto)$ and $(\mathcal{EP}', \mathcal{M}', \mathcal{SPEC}', [\![\_]\!]', \mapsto')$.

A comparison is sensible only if the end products of the two methods are comparable, i.e. if $\mathcal{EP} \cap \mathcal{EP}'$ is not empty, or better if it contains relevant end products. Of course, we can compare the two methods only when restricted to consider $\mathcal{EP} \cap \mathcal{EP}'$.

A *replacement* of FM by FM' is a function

Rep: $\mathcal{SPEC} \to \mathcal{SPEC}'$

s.t.

- it is compatible with the semantics, i.e. if $[\![\mathrm{SP}_1]\!] = [\![\mathrm{SP}_2]\!]$, then $[\![\mathbf{Rep}(\mathrm{SP}_1)]\!]' = [\![\mathbf{Rep}(\mathrm{SP}_2)]\!]'$,
- and for all SP $\in \mathcal{SPEC}$ s.t. $\mathcal{I}([\![\mathrm{SP}]\!]) \subseteq \mathcal{EP} \cap \mathcal{EP}'$, $\mathcal{I}([\![\mathrm{SP}]\!]) = \mathcal{I}'([\![\mathbf{Rep}(\mathrm{SP})]\!]')$;

i.e. the following "partial" diagram commutes:

$$\begin{tikzcd}
 & \mathbf{P}(\mathcal{M}) \arrow[dl, "\mathcal{I}"'] & \mathcal{SPEC} \arrow[l, "\llbracket\_\rrbracket"'] \arrow[dd, "\mathbf{Rep}"'] \\
\mathbf{P}(\mathcal{EP} \cap \mathcal{EP}') & & \\
 & \mathbf{P}(\mathcal{M}') \arrow[ul, "\mathcal{I}'"] & \mathcal{SPEC}' \arrow[l, "\llbracket\_\rrbracket'"']
\end{tikzcd}$$

where $\mathbf{P}(X)$ denotes the collection of the parts of X.

If **Rep** is partial, then only a part of the specifications of FM can be replaced by those of FM'.

If **Rep** is non-injective, then FM is finer (FM' is coarser), i.e. FM' allows to give more abstract specifications.

If **Rep** is non-surjective, then FM' is more powerful (FM' is less powerful), i.e. FM' allows to give more specifications.

In the literature there are various ways of relating formalisms: in particular we have the notions of simulation and of translation. While the second one is essentially what one would expect for analogy with other kinds of translations ($\mathcal{SPEC}$ is translated into $\mathcal{SPEC}'$ and $\mathcal{M}$ into $\mathcal{M}'$), simulations are a bit more sophisticated: if F' simulates F, then the semantics of F' specifications can be understood in terms of the semantics of those of F (while $\mathcal{SPEC}$ is translated into $\mathcal{SPEC}'$, $\mathcal{M}'$ is sent back into $\mathcal{M}$).

It can be shown that both simulation and translation can lead to method replacement under some reasonable assumptions on the two rationales.

Consider now the case when an existing method FM with formalism F has to be modified to use a different formalism F'; e.g. since the original one is no more supported, or a new one is equipped with more software tools.

How to recover/integrate the specifications produced using the original method? How to exploit all experience gained on the original method and in some sense how to keep the "method" ?

The key idea is to provide a suitable translation of F into F' and then derive a modified method by transferring the rationale along with the translation.

In [5] and [4] we consider the interesting case of the relationship between **RL** and **SMoLCS-D**. The interest lies in the fact that the two methods have comparable end products, similar activities, formal models and specifications with common features, as the two specification languages almost coincide for syntax; furthermore these two are the only methods in the literature having such common aspects, and frequently they are confused. By trying to relate the methods reveals the important differences.

Not only the end products of **RL** are a subset of those of **SMoLCS-D**.

Out of three possible simulations between the two formalisms (the most sensible ones) only one of them will result in a method replacement (of **RL** by **SMoLCS-D**).

Moreover the almost obvious embedding of *RL* into the **SMoLCS-D** formalism does not provide a method replacement.

As a last remark it can be shown that the method we have called **ASSRS** (by Bidoit et al.), apparently less related to **SMoLCS-D** than **RL**, because of the underlying similar concepts in the rationale, can be replaced in a very natural way by **SMoLCS-D**.

4 Conclusions

We started with some general discussion on the permanent controversy on formal methods and the current rather confusing situation, with different authoritative views on what should be done in the formal methods area. Adopting the view that researchers should take more care of the technology transfer problem, we have advocated a more explicit connection of a formalism to the methodological aspects for really getting an effective formal method.

Being the time and our experience not mature enough for addressing the problem in its globality (we do not even know whether it would be sensible), we have confined ourselves to discuss in some detail the activity of providing formal specifications. We have presented some basic ideas on how to provide a pattern qualifying the different aspects of a method, distinguishing between context, formalism and pragmatics and relating them in a method.

Though of preliminary character, we feel that some of the ideas can be exploited in other different contexts and perhaps generalized as a useful conceptual tool. Morever we hope to have shown that this is a subject of interesting investigation itself (but please look at the full paper for a more comprehensive and rigorous treatment).

Finally, we will welcome useful comments, constructive criticism and suggestions.

References

1. C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language*. Oxford University Press, 1977.
2. 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.
3. E. Astesiano and G. Reggio. Labelled Transition Logic: An Outline. Technical Report DISI–TR–96–20, DISI – Università di Genova, Italy, 1996.
4. E. Astesiano and G. Reggio. On the Relationship between Labelled Transition Logic and Rewriting Logic. Technical Report DISI–TR–96–19, DISI – Università di Genova, Italy, 1996.
5. E. Astesiano and G. Reggio. Formalism and Method. Technical Report DISI-TR-97-3, DISI – Università di Genova, Italy, 1997. Full version.
6. M. Bidoit, C. Chevenier, C. Pellen, and J. Ryckbosh. An Algebraic Specification of the Steam-Boiler Control System. In J.-R. Abrial, E. Borger, and H. Langmaack, editors, *Formal Methods for Industrial Applications*, number 1165 in Lecture Notes in Computer Science, pages 79–108. Springer Verlag, Berlin, 1996.
7. M. Broy and C. Jones. Editorial. *Formal Aspects of Computing*, 8(1–2), 1996.

8. R.M. Burstall and J.A. Goguen. Institutions: Abstract Model Theory for Specification and Programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, 1992.
9. M. Chandy and J. Misra. *Parallel Program Design: a Foundation.* Addison-Wesley, 1988.
10. G. Costa and G. Reggio. Specification of Abstract Dynamic Data Types: A Temporal Logic Approach. *T.C.S.*, 173, 1997. To appear.
11. D. Craigen, S. Gerhart, and T. Ralston. An International Survey of Industrial Applications of Formal Methods: Volume 1 Purpose, Approach, Analysis and Conclusions. Technical Report NIST GCR 93/626, NIST, 1993.
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. A. Finkelstein, J. Kramer, and B. Nuseibeh, editors. *Software Process Modelling and Technology.* John Wiley & Sons, 1994.
14. C. Floyd. Theory and Practice of Software Development: Stages in a Debate. In P.D. Mosses, M. Nielsen, and M.I. Schwartzbach, editors, *Proc. of TAPSOFT '95*, number 915 in Lecture Notes in Computer Science, pages 25–41. Springer Verlag, Berlin, 1995.
15. W. Wayt Gibbs. Software's Chronic Crisis. *Scientific American*, (9):72–81, 1994.
16. J. Goguen and Luqi. Formal Methods and Social Context in 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 62–81. Springer Verlag, Berlin, 1995.
17. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, London, 1985.
18. 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.
19. C.A.R. Hoare. Unification of Theories: A Challenge for Computing Science. In M. Haveraaen, O. Owe, and O.-J. Dahl, editors, *Recent Trends in Data Type Specification*, number 1130 in Lecture Notes in Computer Science, pages 49–57. Springer Verlag, Berlin, 1996. 11th Workshop on Specification of Abstract Data Types joint with the 8th general COMPASS workshop. Oslo, Norway, September 1995. Selected papers.
20. I.S.O. LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. IS 8807, International Organization for Standardization, 1989.
21. J. Meseguer. Conditional Rewriting as a Unified Model of Concurrency. *T.C.S.*, 96:73–155, 1992.
22. R. Milner. *Communication and Concurrency.* Prentice Hall, London, 1989.
23. Monterey. Announcement of the Monterey "Workshop on Formal Methods for Computer Aided Software Development". 1994.
24. J. Sommerville. *Software Engineering: Third Edition.* Addison-Wesley, 1989.
25. J. Souquières and N. Lévy. Description of Specification and Developments. In *Proc. of International Symposium on Requirements Engineering RE'93.* IEEE Computer Society, Los Alamitos, CA, 1993.
26. H. van Vliet. *Software Engineering: Principles and Practice.* John Wiley & Sons, 1993.
27. M. Wirsing. Algebraic Specifications. In J. van Leeuwen, editor, *Handbook of Theoret. Comput. Sci.*, volume B, pages 675–788. Elsevier, 1990.

28. M. Wirsing. Algebraic Specification Languages: An Overview. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specification*, number 906 in Lecture Notes in Computer Science, pages 81–115. Springer-Verlag, Berlin, 1995.

Table of Contents

This article was processed using the LaTeX macro package with LLNCS style