Formalism and Method ⋆

Egidio Astesiano and Gianna Reggio

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

RATIO ET VIA

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.

To make our reflections more concrete we illustrate our viewpoint addressing one particular activity within the software development process, namely the use of formal specification techniques.

To qualify the essential ingredients of a formal method for specification, we propose a pattern covering the formal and the methodological aspects and also their mutual relationships. Our pattern includes some novel concepts such as the relationship between end-products and formal models, which allows to relate in a rigorous way different methods, outlining the concept of compositionality and of simulation of methods.

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

⋆ Expanded version of an invited talk at TAPSOFT'97 (Lille) [6].
Partially funded by the MURST project: Sistemi formali per la specifica, l'analisi, la verifica, la sintesi e la trasformazione di sistemi software.

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 laying down the following two principles for LFCS activity:

- The design of computer systems can only properly succeed, if it is well grounded in theory.
- 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, many of the attendees were feeling uneasy, reflecting whether the first principle could 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 [14], made a mixed-feeling remark:

> 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 [18] 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 [20] 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 explanation of this situation. For Hoare in [21]

> 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.

Goguen and Luqi in [18] take a completely different view:

> 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 [17], 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 Ariane 5 in June '96, with the careful explanation 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 [20] 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 [21] 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 viewpoint, 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 [29], a widely known book on Software Engineering, 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 foreword [24] 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 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 [16], where she remarks that the survey by Ehrig and Mahr in [14] "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 consideration is echoed in [14] 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 regarding the overall task of software development.

Indeed what is emerging now in recent years is a different 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 [17]. For Goguen and Luqi in [18], in line with [16], "One major problem has been that formal methods have not taken sufficient account of the social context of computer systems."

From another perspective, in [24] 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 [13].

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 that sometimes leads to conflate the two things, always at the expense of the method. This danger is also reflected in [9], 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" and "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 [16]:

> 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 [16], between promoters of formal methods and experts/researchers in software engineering practice.

1.2 Stating our aims

Sometimes it is illuminating to go back to the origin of a word and this is indeed the case: "method" comes 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 either only "ratio" or only "via" is left of the two.

As Roel Wieringa has pointed out (in [Personal communication]; see also [33]), we should look at the engineering cycle known from other areas of product development (see [26]):

- analyse the problem (user needs, goals)
- synthesize solution specification(s)
- predict the effect of implementing the specs (properties of implementations etc.)
- evaluate these predictions w.r.t. the problem analysis iterate to an earlier task or choose a solution.

The idea is that in a *rational* process, design choices are made this way. Of course, there is the business that in practice, things are more chaotic and that we should fake a rational process etc.

The ability to predict what the product will be like is an essential part of engineering. If we cannot do this, and must wait for the implemented product in order to know what properties the product will have, we are just tinkering rather than practicing engineering design.

But if we want to make predictions, we must have a specification, formal or informal. To predict the properties of the implementation, we may perform experiments on a prototype, look at the experience of others, or deduce properties from the specification. In that last process, formal techniques play an essential role. Also, if after the fact we cannot state which design alternatives we looked at and why we chose one particular alternative, then we cannot justify the design. So the rational design cycle places formal techniques in the context of the design decisions. For me this is the connection between ratio (formalism) and via (the development process).

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 only some 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.
- To get or to understand a method it is essential to locate it within the context of the overall development process, in particular defining within that context its kind of activity and its target.
- A rationale should be mandatory; but "rationale" should mean something much more precise than just a few 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.
– 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 the 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 believe 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 think 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 emphasizing the difference between formalism and method, also providing some illustrative examples of analysis on that basis. Then we exploit the presented pattern for discussing two typical and important issues, compositionality and simulation, in a sense making the case that only at the method level we can provide concepts powerful enough, encompassing those related to formalisms, and more significant for their real use.

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 more as stimulating a debate and further research than proposing definitive conclusions or solutions.

2 A Pattern for Specification

2.1 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 [29,32] and [15] for a specific treatment of process modelling.

A development process will return some products of some kind (*end-product* from now on) to be delivered to the client; thus for each development process we may qualify what is the kind of its end-products. Notice that the end-products may be pure software, as programs for statistic analysis, or whole systems having also non-software parts, as information systems (which may have as components the clerks using it) or embedded systems (which may have as components some controlled mechanic and electronic devices). Furthermore a development process may return more than one products, for example the various versions of a software package. In Tab. 2.1 we present a list of keywords qualifying kinds of end-products currently found in the literature. Some items are enough standard and well-understood, whereas 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.

Using a software engineering terminology following [23], the end-products are either the "machine" or the "machine plus the application domain"; sometimes the end-products are also called "systems", as in [8].

A *development process* is a collection of *activities* with temporal/causal relationships among them; furthermore there are meta activities concerning the definition and the management of the development process. In Tab. 2.1 we present a tentative list of possible activities. The items in this list have been found in papers about software engineering.

Each activity at the end will return an artifact (a specification, some code, some documentation, a development process, etc.).

Some activity may require as mandatory inputs some artifacts that are the results of other activities (e.g., $\bullet$ in Tab. 2.1, which takes a requirement and a design specification and returns either a documentation of why the design is wrong or an ok).

A *method (formal method)* is a way to perform an activity of a particular kind (supported by formal techniques and tools).

In a very general way a *specification* is a description of (possibly some aspects of) an end-product (or of some of its parts) 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.

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

Table 1
Kinds of end-products

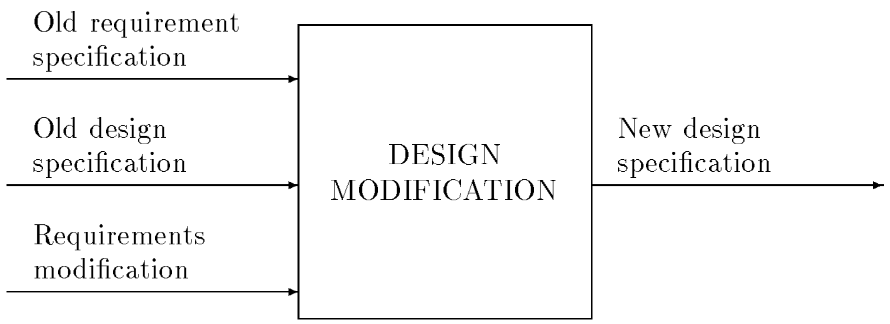
To give a requirement specification
To validate a requirement specification
To give a design specification
To give a design specification starting from a requirement one
To validate a design specification
To verify a design specification against a requirement specification •
To give an intermediate specification (i.e., 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 (also a single activity) by changing something in the inputs
To produce a new version of an already developed end-product (maintenance)
To support the development process definition and management
... ...

Table 2
Activities in the development processes

The reader may get the impression that here and in the following some relevant keywords of software engineering have been neglected; that it is not true. In our opinion they are embedded in the single parts of the various methods, but are not particular parts. We give some examples.

Tools Clearly the activities of a development process may, or better must, be supported by automatic tools; the available tools and how to use them are described in the guidelines, presentation and documentation parts (e.g., a theorem prover, which is valuable only together with proper guidelines, or a graphical interface for producing specifications, which is relative to a graphic presentation).

Evolution Evolution is a property of the development processes; and we may have development processes more or less evolutionary; the classic waterfall model is rather poor in this respect. It is clear that evolutionary development processes are made by particular activities, such as to modify a design given a modification on the original requirements.



Domain knowledge If the supported end-products are very specific (e.g., microprograms for a particular chip, or a special class of protocols) the method may be completely driven by the associate domain knowledge.

2.2 Locating the method within the development process context

2.2.1 End-products

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

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

Quite often end-products are structured, i.e., they exhibit an inner structure. Such structure may be represented by a set of *composers*, that are (possibly partial) functions having $\mathcal{EP}$ as codomain; we can say that $\mathcal{EP}$ gets an algebraic structure. The simplest case is when the structure is homogeneous, i.e., when also all arguments of the composers are in $\mathcal{EP}$; but sometimes the end-products are built also from subparts that are not in $\mathcal{EP}$ (e.g., imperative programs made out from procedures). In the latter case $\mathcal{EP}$ gets a heterogeneous algebraic structure.

To determine, if any, a structure on the end-products it will help to see whether the considered method is modular or not and to discuss the characteristics of

ACTIVITY

To give a formal specification

CONTEXT

END PRODUCTS	$\mathcal{EP}$	the kind of the end-products of the development process
LOCATION		qualification and location of the activity in the development process

FORMALISM

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

↑

IMPACT ON METHOD

↓

PRAGMATICS

MODELLING	$\leftrightharpoons$	how the formal models model the end-products
GUIDELINES		guidelines for the specification task
PRESENTATION		presentation of the specifications for humans
DOCUMENTATION		documenting the performed task

Fig. 1. Components of a specification formal method

such modularity.

2.2.2 Qualification and location

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 Software Engineering (e.g., [29,32]) or to the various papers on development process models (see [15]), will show the reader the many ways "specification" is intended and the different roles in the process. For example, the activity designated as "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 specification 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 components END PRODUCTS and LOCATION should allow to have a coarse idea of the "functionality" of the specification formal method.

2.3 Formalism

2.3.1 Formal models

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 formalism and with the development process models.

Very well-known classes of formal models used by some formalisms 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., [25,22]) for processes
- Sets of action traces (see, e.g., [19]) for processes

Strangely enough, in several presentations of formalisms 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., [10]). Following this view we need to give:

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

Sometimes the formal models are structured, i.e., they exhibit an inner structure. Analogously to the case of the end-products, such structure may be represented by a set of *composers*, that are (possibly partial) functions having $\mathcal{M}$ as codomain; we can say that $\mathcal{M}$ gets an algebraic structure. Also in this case such structure may be either homogeneous or heterogeneous.

A structure on the formal models will help to define structuring operations over the specifications and to see whether the specification structure is compatible with the one of the end-products.

2.3.2 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 or programs in an appropriate *specification language*.

The component SPECIFICATIONS of a formal method consists of:

- A set of specifications $\mathcal{SPEC}$ (programs or terms of the specification language)
- A semantic function $[\![\_]\!]$ (for the specification language) associating with each specification a class of formal models

$$[\![\_]\!] \colon \mathcal{SPEC} \to \mathbf{P}(\mathcal{M})$$ [1]

Notice that there are no assumptions on the cardinality of $[\![\mathrm{SP}]\!]$, with $\mathrm{SP} \in \mathcal{SPEC}$; 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: not all the 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 components are signatures, and their semantics will be a class of formal models on such signatures.

In general, the specifications are structured, i.e., they exhibit an inner structure, because 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.

[1] $\mathbf{P}(\_)$ denotes the powerset (powerclass) operator.

As in the cases of the end-products and of the formal models, such structure may be represented by a set of *composers*, that are (possibly partial) functions having specifications as codomain; we can say that $\mathcal{SPEC}$ gets an algebraic structure. The specification language itself gives a precise syntax to such composers.

Note that the specification composers may be of different kinds.

Model-oriented A specification composer C_{SP} is *model-oriented* iff there exists a model composer C_M s.t.
for all $\mathrm{SP}_1, \mathrm{SP}_2 \in \mathcal{SPEC}$ [2]
$[\![C_{SP}(\mathrm{SP}_1, \mathrm{SP}_2))]\!] = \{C_M(\mathrm{M}_1, \mathrm{M}_2) \mid \mathrm{M}_1 \in [\![\mathrm{SP}_1]\!], \mathrm{M}_2 \in [\![\mathrm{SP}_2]\!]\}$.

Specification-oriented These composers are not linked to the formal models, precisely they are those not satisfying the above condition.

A typical example of specification-oriented composer is the union for property-oriented specifications (see the following Sect. 2.6.3), which builds a new specification just by making the union of the sets of formulae of two other specifications; also inheritance (in the sense of a mechanism for reusing specifications) and the possibility of defining specifications parameterized over something (e.g., parameterized algebraic specifications) are of this kind. The importance of this kind of structuring has been widely recognized since early times, as witnessed in the various specification languages (see [35] and in [28] the definition of a specification language institution independent).

Typical examples of model-oriented composers are the $\_ + \_$ operator of *CCS* and the sequential composer of the Hoare's logic.

It is important to avoid confusing the two kinds of structuring of a specification (e.g. sometimes the *CCS* $\_ + \_$ is used to simulate at some extent inheritance and union, which are lacking); also because their different role w.r.t. development. The model-oriented structuring embodies, throughout the formal models, the information on the structure of the intended end-products, whereas this is not true for the specification-oriented ones, and so this kind of structuring may be modified or forgotten during the development.

[2] Here and in the following for simplicity we consider all composers to be binary, but clearly their arities may be whatever.

2.4 Pragmatics

2.4.1 Modelling

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 $\leftrightharpoons$ between end-products and formal models, where $\mathrm{P} \leftrightharpoons \mathrm{M}$ means intuitively that P *is modelled by* M (or M *is a model for* P or M *models* P).

We consider $\leftrightharpoons$ to be a binary relation and not a function, because it may happen that $\mathrm{P} \leftrightharpoons \mathrm{M}$ and $\mathrm{P} \leftrightharpoons \mathrm{M}'$ with $\mathrm{M} \neq \mathrm{M}'$; in such cases M and M′ differ for irrelevant details (e.g., a data structure may be modelled by two algebras that either differ for the concrete syntax or are isomorphic). In general $\leftrightharpoons$ is not injective; this is sound, because 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 $\leftrightharpoons$ may be a subclass of $\mathcal{M}$; in such cases we have more formal models than we need, but that is not a problem. We may always assume that the domain of $\leftrightharpoons$ coincides with $\mathcal{EP}$.

We require:

1) if $\mathrm{P} \leftrightharpoons \mathrm{M}$, $\mathrm{P}' \leftrightharpoons \mathrm{M}'$ and $\mathrm{P} \leftrightharpoons \mathrm{M}'$, then also $\mathrm{P}' \leftrightharpoons \mathrm{M}$; graphically:

$$\begin{tikzcd}
\mathrm{P} \arrow[r, dash, "\leftrightharpoons"] \arrow[dr, dash, "\leftrightharpoons"'] & \mathrm{M} & \Rightarrow & & \mathrm{M} \\
\mathrm{P}' \arrow[r, dash, "\leftrightharpoons"] & \mathrm{M}' & & \mathrm{P}' \arrow[ur, dash, "\leftrightharpoons"] &
\end{tikzcd}.$$

Moreover we have to require the consistency of $\leftrightharpoons$ with the semantics of specifications, namely the semantics to be closed w.r.t. $\leftrightharpoons$ and the specifications to consider only formal models modelling some end-product:

2) If $\mathrm{P} \leftrightharpoons \mathrm{M}$, $\mathrm{P} \leftrightharpoons \mathrm{M}'$ and $\mathrm{M} \in [\![\mathrm{SP}]\!]$, then $\mathrm{M}' \in [\![\mathrm{SP}]\!]$.
3) For all $\mathrm{M} \in [\![\mathrm{SP}]\!]$, there exists $\mathrm{P} \in \mathcal{EP}$ s.t. $\mathrm{P} \leftrightharpoons \mathrm{M}$.

Assuming to have $\leftrightharpoons$, 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, $\mathcal{A}(\mathrm{Ps}) = \{\mathrm{M} \mid \exists\, \mathrm{P} \in \mathrm{Ps} \,.\, \mathrm{P} \leftrightharpoons \mathrm{M}\}$;
- for every class of models Mc, $\mathcal{I}(\mathrm{Mc}) = \{\mathrm{P} \mid \exists\, \mathrm{M} \in \mathrm{Mc} \,.\, \mathrm{P} \leftrightharpoons \mathrm{M}\}$.

We call $\mathcal{A}$ *abstraction* of end-products and $\mathcal{I}$ *interpretation* of formal models. From **1)** we have that

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

and from **2)** and **3)** we 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 end-products, thus requiring the relation $\leftrightharpoons$ to be compatible with $\sim$:

if $\mathrm{P} \leftrightharpoons \mathrm{M}$, then $\mathrm{M} \sim \mathrm{M}' \Leftrightarrow \mathrm{P} \leftrightharpoons \mathrm{M}'$.

Under this assumption $\leftrightharpoons$ associates with each end-product essentially one model (an equivalence class), thus $\leftrightharpoons$ is a function from $\mathcal{EP}$ into $\mathcal{M}/\sim$; 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. $\sim$.

Notice that such a $\sim$ always exists, under our assumption defined by $\mathrm{M} \sim \mathrm{M}'$ iff there exists P s.t. $\mathrm{P} \leftrightharpoons \mathrm{M}$ and $\mathrm{P} \leftrightharpoons \mathrm{M}'$.

2.5 Remaining components

The following three items in our pattern are 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, 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, whenever available.

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

Presentation We mean by *presentation* the interface with the user, in a broad sense, of a specification artifact. Users, here, can range from the clients, those financing the end-product, who need to understand a requirement specification in its own language (see [29], distinguishing requirement definition from requirement specification), to the implementors, to the specification builder himself, when a change is needed at some later stage. A presentation should hopefully consist 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 task for use in evolution and maintenance. The evolution in software development is now taken care in every process model (see [15]) and its importance in formal methods recognized (see [18]) also some prototype support tools are appearing ([30]).

2.6 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.

2.6.1 Abstraction level of specifications

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, and 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 that are semantics of the specifications. The less abstract specification methods are those where $[\![\mathrm{SP}]\!]$ has cardinality 1 or is just an isomorphism class.

2.6.2 Specification semantics

The technique used for providing the semantics of specification language is not neutral; indeed such semantics can be given in

- A rather direct, explicit and denotational way (e.g., as done by Hoare for *CSP*, [19]), by exhibiting the relative class of formal models
- An indirect or implicit way, say as (1) the limit of a diagram in a category, (2) defining that two specifications are semantically equivalent iff their equality may be proved by a deductive system.

However, in our opinion, providing an explicit way seems to be essential for software engineering purposes; to help people to grasp the meaning of specifications. Techniques as (1) may be used as a quick way to establish the existence of such semantics, whereas those as (2) may be used to help work with the specifications, to provide simpler forms or to show that two specifications coincide.

2.6.3 Specification style

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

Property-oriented (axiomatic) We prefer the term *property-oriented*, as more suggestive than axiomatic. In general property-oriented specifications 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, e.g., in [10]): for each $\Sigma \in \mathcal{SIG}$,

- A set of sentences (or formulae) over Σ, $\mathcal{SEN}_\Sigma$
- A validity notion (i.e., a binary relation $\models_\Sigma \subseteq \mathcal{M}_\Sigma \times \mathcal{SEN}_\Sigma$)

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

For what concerns the semantics, the basic way to define it is

$$[\![(\Sigma, \mathrm{S})]\!] = Mod(\Sigma, \mathrm{S})$$

where

$$Mod(\Sigma, \mathrm{S}) = \{\mathrm{M} \mid \mathrm{M} \in \mathcal{M}_\Sigma \text{ and } \mathrm{M} \models_\Sigma \phi \text{ for all } \phi \in \mathcal{SEN}_\Sigma\}^{3}.$$

[3] The elements of this class are usually called the models of the specification.

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 sentences provided by the formalism.*

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 sentences should be intuitively described by using the natural language in terms of properties of the formal models and via the modelling (see Sect. 2.4.1) 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 the sentences ?

Adequacy Which properties of the end-products may be expressed by the sentences?

As examples, consider the specification languages μ-calculus ([31]) and UNITY ([11]). 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 (constructive) The ingredients for model-oriented specifications are:

- A class of specifications $\mathcal{SPEC}$
- 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*, or *constructive*, with the meaning that we specify an end-product by construction (at the abstraction level supported by the method, that is 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 or construction-oriented specification language may be evaluated by considering:

Expressive power How many/which formal models can be expressed by $[\![\_]\!]'$? and how many/which classes of $\mathcal{M}$ can be expressed by $\succeq$?

Formal-model– or end-product–oriented The model-oriented specification languages 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 many different formal methods considering different classes of end-products (think of λ-calculus); but it may be not very flexible and thus suitable for special classes of end-products (it is possible to model any imperative program by using λ-calculus, 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 sensibly be adopted for different kinds of end-products (e.g., it is not convenient, if possible at all, to use *LOTOS* to specify fully distributed systems).

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

Borderline cases Sometimes, in a property-oriented specification formalism we have also another ingredient: a way to determine one (few) formal models starting from of the model class by additional properties, which cannot be expressed by using the sentences (e.g., constraints). In these cases the semantics is given by:

$$[\![(\Sigma, \mathrm{S})]\!] = \{\mathrm{M} \mid \mathrm{M} \in \mathcal{M}_\Sigma \text{ and additional constraints using } Mod(\Sigma, \mathrm{S})\}$$

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

The observational and the initial semantics are among the most typical examples; in the first case we pick up the class of models, considered equivalent w.r.t. a set of observations to those belonging to *Mod*(SP); in the second we define essentially one model (the initial element of *Mod*(SP)) on the basis of an induction principle for defining the individual elements of the model, plus an equality defined by logical deduction.

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

3 Illustrative cases

In this section by *reactive system* we mean in general a system able to evolve along the time possibly reacting to its external environment disregarding other features; thus a parallel, concurrent, distributed system is a particular case; sometimes in the literature the term *process* is used with the same general meaning.

3.1 Methods based on CCS

CCS, the calculus of communicating systems [25], has been introduced originally as a formalism for describing reactive and concurrent systems, in close analogy with the role of λ-calculus for sequential computations. Together with *CSP* [19] 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 PRODUCTS (Non-distributed) reactive systems.

LOCATION *CCS* can be used both for requirement specifications (say **CCS-R**) and design specifications (say **CCS-D**) in a fragment of a naive water-fall development process represented in Fig. 2.

FORMAL MODELS 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., [25]).

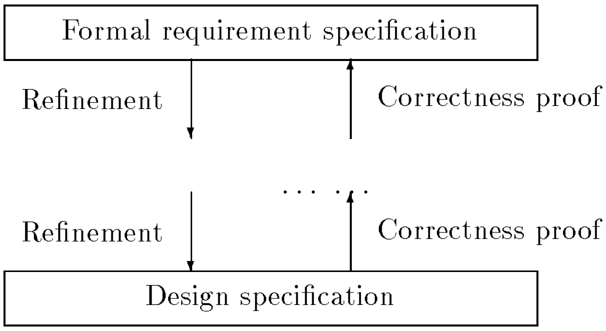


Fig. 2. Development process for *CCS* methods

MODELLING A reactive system R is modelled by a synchronization tree ST, where the nodes of ST represent the intermediate (interesting) situations of the life of R and the labelled arcs of ST the possibilities of R of passing from a state to another one. Note that

- Here a labelled arc (a transition) $s \xrightarrow{l} s'$ has the following meaning: R in the state s has the *capability* of passing into the state s' by performing a transition, where the label l represents the interaction with the external (to R) 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.

In this case $\leftrightharpoons$ is not a function, because a reactive system may be modelled by infinitely many trees differing at most for the used labels. The equivalence relation on the synchronization trees $\sim$, making $\leftrightharpoons$ a function, is defined by:

$\mathrm{ST}_1 \ \sim \mathrm{ST}_2$ iff ST_2 can be obtained by ST_1 by renaming in a bijective way the arc labels.

SPECIFICATIONS **CCS-R** specifications follow a model-oriented style. Every specification consists of a so-called behaviour expression, that is a term in the *CCS* language. The basic semantics of *CCS* is the standard strong bisimulation (see [25]), which gives the synchronization tree associated with a behaviour expression. The relation $\succeq$ is the weak bisimulation preorder; weak bisimulation means forgetting irrelevant (not all) internal moves in a synchronization tree; $\mathrm{ST}_1 \succeq \mathrm{ST}_2$ iff ST_1 is weakly simulated by ST_2.

The specification language, *CCS*, offers both formal model-oriented constructs ($\_ \,.\, \_$, $\_ + \_$) and end-product-oriented constructs ($\_ \parallel \_$). Sometimes the latter is used also for structuring complex specifications of sequential processes.

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

3.2 Methods based on algebraic specifications

Among the methods based on algebraic specifications we consider:

CADT The classical abstract data types specification method, see [34]
SMoLCS-R The SMoLCS method for requirement specifications, see [3,12]
ASSRS The method exemplified by M. Bidoit et al. in their treatment of the steam boiler problem, see [7] (**ASSRS** stands for Algebraic Specification of Sequential Reactive Systems)

Strikingly enough, in all cases, the formalism is essentially the same, whereas the end-products, and consequently, the modelling techniques of such methods are really different.

END PRODUCTS

CADT The usual, static so-to-speak, data types (lists, stacks, bulletin board, etc.)
ASSRS The non-concurrent and non-parallel reactive systems (shortly sequential processes)
SMoLCS-R The reactive systems

LOCATION All these methods cover the formal specification of the requirements in a simple development process schematically reported in Fig. 3.

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

SPECIFICATIONS 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 temporal, modal and deontic logic respectively to express liveness and safety properties on the behaviour of the reactive systems (see, e.g., [12]).

MODELLING

CADT In this case the modelling is trivial: carriers and interpretations of operations and predicates represent respectively the values (classified by types), the operations and tests of the data type.

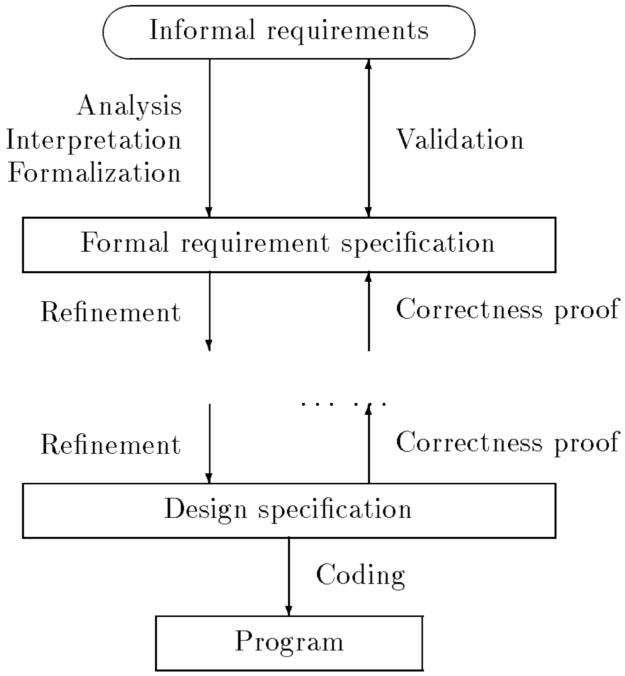


Fig. 3. Development process for methods based on algebraic specifications

ASSRS A sequential process receives information from the external world and sends them to it; thus, it is modelled by an activity function, which given a set of input messages (information received from outside) and its actual state returns a new state and a set of output messages (information sent outside).

The signature of the associated algebra will have three sorts

`input-message-set`, `state`

and two operations

`Answer`: `input-message-set` $\times$ `state` $\to$ `output-message-set`

`Next-State`: `input-message-set` $\times$ `state` $\to$ `state`,

These functions allow to represent the activity function.

SMoLCS-R A part of the modelling is supported at the syntactic level, where some of the sorts are qualified as *dynamic* and are such that for each of them, say ds, there exit a corresponding sort of labels l_ds and a labelled transition predicate $\_ \xrightarrow{-} \_$: $ds\ l\_ds\ ds$. Given an algebra L, each one of its reactive sorts, say ds, determines a labelled transition system $(\mathrm{L}_{ds}, \mathrm{L}_{l\_ds}, \longrightarrow^{\mathrm{L}})$ representing a type of reactive systems.

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

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

There is also a variant of the SMoLCS method for design specifications; it shares all components with **SMoLCS-R** except, obviously, location and specifications. Its specifications follow a borderline style using many-sorted first-order conditional logic (see [5]), plus the constraint on the models picking up the initial element, exactly one, modulo isomorphism.

The presentation part for both **SMoLCS-R** and **SMoLCS-D** includes a way to complement formal specifications with informal ones ([4]) and graphic ones ([27]). Guidelines have been developed too, and are briefly sketched in [27].

4 Analysing Compositionality

Compositionality is one of the basic technical principles supporting modularity in software development. Let us propose a version of it for methods.

Assume to have a formal method FM, whose relevant components are $\mathcal{EP}$, $\mathcal{M}$, $\mathcal{SPEC}$, $[\![\_]\!]$ and $\leftrightharpoons$ respectively, and that the structures on end-products, formal models and specifications are given by the signatures $\Sigma_{\mathcal{EP}}$, $\Sigma_{\mathcal{M}}$ and $\Sigma_{\mathcal{SPEC}}$ respectively. In the following, given a signature of composers Σ, we write $C \in \Sigma^+$ to denote a composer either belonging to Σ or derived by composing those in Σ.

The usual concept of compositionality is not interesting when applied to methods.

We say that FM *is compositional* iff

for each specification composer $C_{SP} \in \Sigma_{\mathcal{SPEC}}$, there exists an end-product composer $C_P \in \Sigma^+_{\mathcal{EP}}$, s.t.:

for all SP_1, $\mathrm{SP}_2 \in \mathcal{SPEC}$

$\{C_P(\mathrm{P}_1, \mathrm{P}_2) \mid \mathrm{P}_1 \in \mathcal{I}([\![\mathrm{SP}_1]\!]), \mathrm{P}_2 \in \mathcal{I}([\![\mathrm{SP}_2]\!])\} = \mathcal{I}([\![C_{SP}(\mathrm{SP}_1, \mathrm{SP}_2)]\!])$.

Many existing formal methods are not compositional in this sense; for example, almost all those having specification composers specification-oriented (see Sect. 2.3.2); and a specification formalism without specification-oriented composers may be really poor and not flexible.

The right notions about compositionality for methods may be informally expressed by the following sentence:

> End-products made by putting together several parts may be specified (at some abstraction level) by putting together the specifications of such parts

and are formally defined below.

- FM *supports the structure of* $\mathcal{EP}$ iff
  for each end-product composer $C_P \in \Sigma_{\mathcal{EP}}$, there exists a specification composer $C_{SP} \in \Sigma^+_{\mathcal{SPEC}}$, s.t.:
  for all SP_1, $\mathrm{SP}_2 \in \mathcal{SPEC}$
  $\{C_P(\mathrm{P}_1, \mathrm{P}_2) \mid \mathrm{P}_1 \in \mathcal{I}([\![\mathrm{SP}_1]\!]), \mathrm{P}_2 \in \mathcal{I}([\![\mathrm{SP}_2]\!])\} = \mathcal{I}([\![C_{SP}(\mathrm{SP}_1, \mathrm{SP}_2)]\!])$.
- FM *weakly supports the structure of* $\mathcal{EP}$ iff
  for each end-product composer $C_P \in \Sigma_{\mathcal{EP}}$, there exists a specification composer $C_{SP} \in \Sigma^+_{\mathcal{SPEC}}$, s.t.:
  for all SP_1, $\mathrm{SP}_2 \in \mathcal{SPEC}$
  · $\{C_P(\mathrm{P}_1, \mathrm{P}_2) \mid \mathrm{P}_1 \in \mathcal{I}([\![\mathrm{SP}_1]\!]), \mathrm{P}_2 \in \mathcal{I}([\![\mathrm{SP}_2]\!])\} \subseteq \mathcal{I}([\![C_{SP}(\mathrm{SP}_1, \mathrm{SP}_2)]\!])$,
  · $\mathcal{A}(\{C_P(\mathrm{P}_1, \mathrm{P}_2) \mid \mathrm{P}_1 \in \mathcal{I}([\![\mathrm{SP}_1]\!]), \mathrm{P}_2 \in \mathcal{I}([\![\mathrm{SP}_2]\!])\}) \supseteq \mathcal{A}(\mathcal{I}([\![C_{SP}(\mathrm{SP}_1, \mathrm{SP}_2)]\!]))$
  (it is equivalent to require equality instead of containment in this last point; indeed $\mathcal{A}$ is monotonic w.r.t. set inclusion).

For example, **CCS-D** is weakly compositional, but not compositional, because the formal model of the parallel composition of two processes can be used also to model a sequential process, indeed it is false that

$\{proc_1$ in parallel with $proc_2 \mid proc_1 \in \mathcal{I}([\![BE_1]\!]), proc_2 \in \mathcal{I}([\![BE_2]\!])\} \supseteq$
$\mathcal{I}([\![BE_1 \mid\mid BE_2]\!])$,

because $\mathcal{I}([\![BE_1 \mid\mid BE_2]\!])$ contains also sequential processes having the same synchronization tree of the parallel composition of the two processes.

Now we examine the compositionality properties of the components of a method.

- the formalism (i.e., $\mathcal{M}$, $\mathcal{SPEC}$ and $[\![\_]\!]$) *supports the structure of* $\mathcal{M}$ iff
  for each formal model composer $C_M \in \Sigma_{\mathcal{M}}$, there exists a specification composer $C_{SP} \in \Sigma^+_{\mathcal{SPEC}}$, s.t.
  for all SP_1, $\mathrm{SP}_2 \in \mathcal{SPEC}$
  $\{C_M(\mathrm{M}_1, \mathrm{M}_2) \mid \mathrm{M}_1 \in [\![\mathrm{SP}_1]\!], \mathrm{M}_2 \in [\![\mathrm{SP}_2]\!]\} = [\![C_{SP}(\mathrm{SP}_1, \mathrm{SP}_2))]\!]$.
- $\leftrightharpoons$ *weakly supports the structure of* $\mathcal{EP}$ iff

for each end-product composer $C_P \in \Sigma_{\mathcal{EP}}$, there exists a formal model composer $C_M \in \Sigma^+_{\mathcal{M}}$, s.t.
$C_P(\mathrm{P}_1, \mathrm{P}_2) \leftrightharpoons \mathrm{M} \Leftrightarrow$ there exist M_1, M_2 s.t. $\mathrm{P}_1 \leftrightharpoons \mathrm{M}_1$, $\mathrm{P}_2 \leftrightharpoons \mathrm{M}_2$ and $\mathrm{M} = C_M(\mathrm{M}_1, \mathrm{M}_2)$;

- $\leftrightharpoons$ *supports the structure of* $\mathcal{EP}$ iff
$\leftrightharpoons$ weakly supports it and $\mathrm{P} \leftrightharpoons C_M(\mathrm{M}_1, \mathrm{M}_2) \Rightarrow$ there exist P_1, P_2 s.t. $\mathrm{P}_1 \leftrightharpoons \mathrm{M}_1$, $\mathrm{P}_2 \leftrightharpoons \mathrm{M}_2$ and $\mathrm{P} = C_P(\mathrm{P}_1, \mathrm{P}_2)$.

Proposition 1 *If the formalism of* FM *supports the structure of* $\mathcal{M}$ *and* $\leftrightharpoons$ *(weakly) supports the structure of* $\mathcal{EP}$*, then* FM *(weakly) supports the structure of* $\mathcal{EP}$*.*

PROOF. Let $C_P \in \Sigma_{\mathcal{EP}}$. By the hypothesis there exists $C_M \in \Sigma^+_{\mathcal{M}}$, and $C_{SP} \in \Sigma^+_{\mathcal{SPEC}}$ with the appropriate properties.

$\mathcal{I}([\![C_{SP}(\mathrm{SP}_1, \mathrm{SP}_2)]\!])$ = (because F supports the structure of $\mathcal{M}$)
$\mathcal{I}(\{C_M(\mathrm{M}_1, \mathrm{M}_2) \mid \mathrm{M}_1 \in [\![\mathrm{SP}_1]\!], \mathrm{M}_2 \in [\![\mathrm{SP}_2]\!],\})$ = (by definition of $\mathcal{I}$)
$\{\mathrm{P} \mid \exists\ \mathrm{M}_1 \in [\![\mathrm{SP}_1]\!], \mathrm{M}_2 \in [\![\mathrm{SP}_2]\!], \text{ s.t. } \mathrm{P} \leftrightharpoons C_M(\mathrm{M}_1, \mathrm{M}_2)\} = \mathrm{A}$

$\{C_P(\mathrm{P}_1, \mathrm{P}_2) \mid \mathrm{P}_1 \in \mathcal{I}([\![\mathrm{SP}_1]\!]), \mathrm{P}_2 \in \mathcal{I}([\![\mathrm{SP}_2]\!])\} =$
$\{C_P(\mathrm{P}_1, \mathrm{P}_2) \mid \exists\ \mathrm{M}_1 \in [\![\mathrm{SP}_1]\!], \mathrm{M}_2 \in [\![\mathrm{SP}_2]\!] \text{ s.t. } \mathrm{P}_1 \leftrightharpoons \mathrm{M}_1, \mathrm{P}_2 \leftrightharpoons \mathrm{M}_2\} = \mathrm{B}$

Because $\leftrightharpoons$ weakly supports the structure of $\mathcal{EP}$, we have that
$C_P(\mathrm{P}_1, \mathrm{P}_2) \leftrightharpoons C_M(\mathrm{M}_1, \mathrm{M}_2)$, and so $\mathrm{B} \subseteq \mathrm{A}$.

$\mathcal{A}(\mathcal{I}([\![C_{SP}(\mathrm{SP}_1, \mathrm{SP}_2)]\!]))$ = (from **c)** of Sect. 2.4.1) $[\![C_{SP}(\mathrm{SP}_1, \mathrm{SP}_2)]\!]$ =
(because F supports the structure of $\mathcal{M}$)
$\{C_M(\mathrm{M}_1, \mathrm{M}_2) \mid \mathrm{M}_1 \in [\![\mathrm{SP}_1]\!], \mathrm{M}_2 \in [\![\mathrm{SP}_2]\!],\}$ = (by **3)** of Sect. 2.4.1)
$\{C_M(\mathrm{M}_1, \mathrm{M}_2) \mid \mathrm{M}_1 \in [\![\mathrm{SP}_1]\!], \mathrm{M}_2 \in [\![\mathrm{SP}_2]\!], \exists\ \mathrm{P}_1 \in \mathcal{EP}, \mathrm{P}_2 \in \mathcal{EP}, \mathrm{P}_1 \leftrightharpoons \mathrm{M}_1, \mathrm{P}_2 \leftrightharpoons \mathrm{M}_2\} = \mathrm{C}$

$\mathcal{A}(\{C_P(\mathrm{P}_1, \mathrm{P}_2) \mid \mathrm{P}_1 \in \mathcal{I}([\![\mathrm{SP}_1]\!]), \mathrm{P}_2 \in \mathcal{I}([\![\mathrm{SP}_2]\!])\}) = \mathcal{A}(\mathrm{B}) =$
$\{\mathrm{M} \mid \exists\ \mathrm{M}_1 \in [\![\mathrm{SP}_1]\!], \mathrm{M}_2 \in [\![\mathrm{SP}_2]\!], \mathrm{P}_1 \in \mathcal{EP}, \mathrm{P}_2 \in \mathcal{EP}, \mathrm{P}_1 \leftrightharpoons \mathrm{M}_1, \mathrm{P}_2 \leftrightharpoons \mathrm{M}_2, C_P(\mathrm{P}_1, \mathrm{P}_2) \leftrightharpoons \mathrm{M}\}$
Because $\leftrightharpoons$ weakly supports the structure of $\mathcal{EP}$, it contains C.

Because $\leftrightharpoons$ supports the structure of $\mathcal{EP}$, we have that $\mathrm{A} \subseteq \mathrm{B}$. $\square$

5 Relating Methods

We present here another application of the proposed pattern, showing how methods and not just formalisms can be compared. First we define a notion of simulation of methods and then discuss how methods can be simulated by simulating and translating their formalisms.

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

FM and FM$'$ are *comparable* only if their LOCATION's are homogeneous; it makes no sense to compare a method for requirement specification with another one for low level design; furthermore they should have common end-products, that is $\mathcal{EP} \cap \mathcal{EP}'$ is not empty, or better it contains relevant end-products. Of course, we can compare the two methods only when restricted to consider $\mathcal{EP} \cap \mathcal{EP}'$.

If the LOCATION's of FM and of FM$'$ are homogeneous and $\mathcal{EP}' \subset \mathcal{EP}$, then we say that FM *has a wider spectrum than* FM$'$ *has*, or that FM$'$ *has a smaller spectrum than* FM *has*.

This relationship is not a measure of absolute merit. Indeed, if one has to work only with a precise class of end-products, a specifically developed method may be better than a general purpose one (perhaps some domain knowledge has been incorporated, or it is simpler to learn, ...). On the other hand, a general purpose method may be good for the case one has to work with different kinds of applications, because in this case the big effort to learn it has to be made only once.

A different relationship concerns with expressiveness.

We say that FM is *more powerful (expressive)* than FM$'$ is iff $\mathcal{EP}' \subseteq \mathcal{EP}$ and for all SP$' \in \mathcal{SPEC}'$, there exists SP $\in \mathcal{SPEC}$ s.t. $\mathcal{I}'([\![\mathrm{SP}']\!]') = \mathcal{I}([\![\mathrm{SP}]\!])$ (any class of end-products specifiable using FM$'$ can also be specified using FM).

5.1 Simulating methods

We want to know whether a formal method FM can be simulated by another one FM$'$ having similar LOCATION's and s.t. $\mathcal{EP}' \subseteq \mathcal{EP}$. In this case it is not sufficient to know that the latter is more powerful, but we want to know also how to find the specifications of FM$'$ that can be used to simulate those of FM.

A *simulation* of FM by FM$'$ is a total function $\mathbf{Sim}\colon \mathcal{SPEC} \to \mathcal{SPEC}'$ s.t.
for all SP $\in \mathcal{SPEC}$, $\mathcal{I}([\![\mathrm{SP}]\!]) = \mathcal{I}'([\![\mathbf{Sim}(\mathrm{SP})]\!]')$, that is the following diagram commutes:

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

If **Sim** is non-injective, then FM is richer (FM$'$ is poorer), that is FM offers more tools for presenting specifications (e.g. to modularly decompose specifications making them more readable, as the possibility of declaring procedures in a programming language). Clearly, if **Sim** is non-surjective, then FM$'$ is more powerful (FM is less powerful).

As an example we can try to simulate **ASSRS** by **SMoLCS-R**, see Sect. 3.2.

The two methods are comparable, because they have the same LOCATION component and $\mathcal{EP}_{\mathbf{ASSRS}} \subset \mathcal{EP}_{\mathbf{SMoLCS-R}}$; indeed, **ASSRS** consider only sequential reactive systems, and being more precise, only those without local non-determinism, that are reactive systems where the reception of a stimulus from outside in a given state can produce only one reaction.

The simulation function $\mathbf{Sim_{AS}}$ associates with a **SMoLCS-R** specification `SP` the following **ASSRS** specification:

```
use SP
dsort state: _ -- _ --> _
op  <_;_>: input-message-set output-message-set -> lab_state
ax  s -- < is; Answer(s,is) > --> Next-State(s,is)
ax  forall l: lab_state
    exists is: input-message-set, os: output-message-set
    l = < is; os >
ax  < is; os > = < is'; os' >  iff  is = is' and os = os'
```

$\mathbf{Sim_{AS}}$ is total, injective and clearly non-surjective.

5.2 Relating methods via formalisms

Let us now relate methods by looking at the relationships between their formalisms.

Assume to have two specification formalisms, F and F$'$, whose components

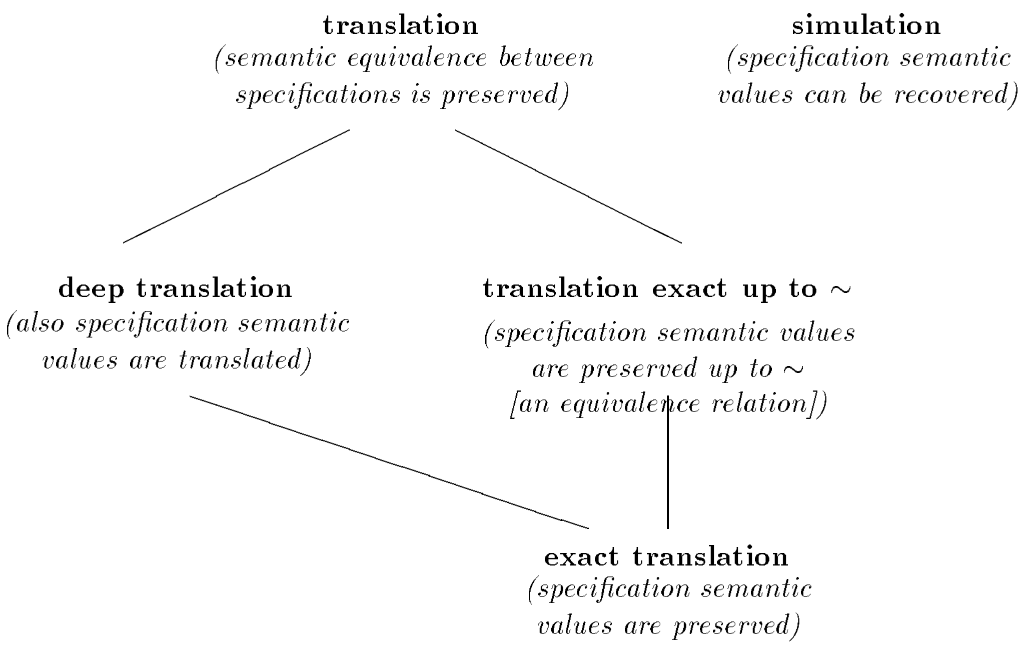


Fig. 4. Relationships between formalisms

given following Sect. 2 are respectively $(\mathcal{M}, \mathcal{SPEC}, [\![\_]\!])$ and $(\mathcal{M}', \mathcal{SPEC}', [\![\_]\!]')$. We can define several different relationships between F and F′; some derived by the analogous relations between institutions (see [2]) and others derived by the translations between languages (a specification formalism may be a language). Such relationships are graphically summarized in Fig. 4.

Let $\mathbf{S}\colon \mathcal{SPEC} \to \mathcal{SPEC}'$ be a total function and $\mathbf{M}\colon \mathcal{M} \to \mathcal{M}'$, $\mathbf{M}'\colon \mathcal{M}' \to \mathcal{M}$ be two functions.

- $\mathbf{S}$ is a *translation* of F into F′ iff for all $\mathrm{SP}_1, \mathrm{SP}_2 \in \mathcal{SPEC}$,
  $[\![\mathrm{SP}_1]\!] = [\![\mathrm{SP}_2]\!]$ iff $[\![\mathbf{S}(\mathrm{SP}_1)]\!]' = [\![\mathbf{S}(\mathrm{SP}_2)]\!]'$.
- $(\mathbf{S}, \mathbf{M})$ is a *deep translation* of F into F′ iff
  it is a translation and for all $\mathrm{SP} \in \mathcal{SPEC}$,
  $\mathbf{P}(\mathbf{M})[\![\mathrm{SP}]\!] = [\![\mathbf{S}(\mathrm{SP})]\!]'$[4]; that is the diagram in Fig. 5.a commutes.
- if $\mathcal{M} = \mathcal{M}'$ and $\sim$ is an equivalence relation on $\mathcal{M}$, $\mathbf{S}$ is a *translation exact up to* $\sim$ of F into F′ iff
  it is a translation and for all $\mathrm{SP} \in \mathcal{SPEC}$,
  $\{[\mathrm{M}]_\sim \mid \mathrm{M} \in [\![\mathrm{SP}]\!]\} = \{[\mathrm{M}]_\sim \mid \mathrm{M} \in [\![\mathbf{S}(\mathrm{SP})]\!]'\}$;
  if $\sim$ is the identity, then $\mathbf{S}$ is an *exact translation*;
- $(\mathbf{S}, \mathbf{M}')$ is a *simulation* of F by F′ iff for all $\mathrm{SP} \in \mathcal{SPEC}$,
  $\mathbf{P}(\mathbf{M}')([\![\mathbf{S}(\mathrm{SP})]\!]') = [\![\mathrm{SP}]\!]'$, that is the diagram in Fig. 5.b commutes.

Obviously, we have that deep translations and translations up to are translations and that exact translations are deep and exact up to identity; whereas

[4] Given $f\colon A \to B$ and $X \subseteq A$, $\mathbf{P}(f)(X) = \{f(x) \mid x \in X\}$.

$$\begin{tikzcd}
\mathbf{P}(\mathcal{M}) \arrow[d, "\mathbf{P}(\mathbf{M})"] & \mathcal{SPEC} \arrow[l, "[\![_]\!]"'] \arrow[d, "\mathbf{S}"] \\
\mathbf{P}(\mathcal{M}') & \mathcal{SPEC}' \arrow[l, "[\![_]\!]'"']
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
\mathbf{P}(\mathcal{M}) & \mathcal{SPEC} \arrow[l, "[\![_]\!]"'] \arrow[d, "\mathbf{S}"] \\
\mathbf{P}(\mathcal{M}') \arrow[u, "\mathbf{P}(\mathbf{M}')"'] & \mathcal{SPEC}' \arrow[l, "[\![_]\!]'"']
\end{tikzcd}$$

a) b)

Fig. 5.

there is no relationship between translations and simulations.

Note that relating formalisms is not relating methods. Indeed, if we give a relationship between the formalisms of two formal specification methods, then not always we have a relationship between the two methods: the modellings have to be taken into account too; also an exact translation may be not a method simulation. Below we give sufficient conditions for deriving a relationship between methods from one between their formalisms.

Proposition 2 *Let* FM *and* FM$'$ *be two formal specification methods with formalisms* F *and* F$'$ *respectively.*

(i) Let $(\mathbf{S}, \mathbf{M}')$ *be a simulation of* F *by* F$'$*; if* $\mathcal{I}'(\mathrm{Mc}') = \mathcal{I}(\mathbf{P}(\mathbf{M}')(\mathrm{Mc}'))$*, that is if the following diagram commutes, then* $\mathbf{S}$ *is a simulation of* FM *by* FM$'$*.*

$$\begin{tikzcd}
 & \mathbf{P}(\mathcal{M}) \arrow[dl, "\mathcal{I}"'] \\
\mathbf{P}(\mathcal{EP}') & \\
 & \mathbf{P}(\mathcal{M}') \arrow[ul, "\mathcal{I}'"] \arrow[uu, "\mathbf{P}(\mathbf{M}')"']
\end{tikzcd}$$

(ii) Let $(\mathbf{S}, \mathbf{M})$ *be a deep translation of* F *by* F$'$*; if* $\mathcal{I}'(\mathbf{P}(\mathbf{M})(\mathrm{Mc})) = \mathcal{I}(\mathrm{Mc})$*, that is if the following diagram commutes, then* $\mathbf{S}$ *is a simulation of* FM *by* FM$'$*.*

$$\begin{tikzcd}
 & \mathbf{P}(\mathcal{M}) \arrow[dl, "\mathcal{I}"'] \arrow[dd, "\mathbf{P}(\mathbf{M})"] \\
\mathbf{P}(\mathcal{EP}') & \\
 & \mathbf{P}(\mathcal{M}') \arrow[ul, "\mathcal{I}'"]
\end{tikzcd}$$

(iii) Let $(\mathbf{S}, \mathbf{M})$ *be a translation exact up to* $\sim$ *of* F *into* F′*; if* $\leftrightharpoons$ *and* $\leftrightharpoons'$ *are compatible with* $\sim$*, that is*

$\mathrm{P} \leftrightharpoons \mathrm{M}$ *and* $\mathrm{M} \sim \mathrm{M}'$ *implies* $\mathrm{P} \leftrightharpoons \mathrm{M}'$

$\mathrm{P} \leftrightharpoons' \mathrm{M}$ *and* $\mathrm{M} \sim \mathrm{M}'$ *implies* $\mathrm{P} \leftrightharpoons' \mathrm{M}'$,

then $\mathbf{S}$ *is a simulation of* FM *by* FM′.

(iv) Let $\mathbf{S}$ *be an exact translation of* F *into* F′*; if* $\leftrightharpoons$ *and* $\leftrightharpoons'$ *coincide, then* $\mathbf{S}$ *is a simulation of* FM *by* FM′.

PROOF. *i)* $\mathcal{I}([\![\mathrm{SP}]\!]) =$ *(because* $(\mathbf{S}, \mathbf{M}')$ *is a simulation of formalisms)* $\mathcal{I}(\mathbf{P}(\mathbf{M}')([\![\mathbf{S}(\mathrm{SP})]\!]')) =$ *(by the hypothesis)* $\mathcal{I}'([\![\mathbf{S}(\mathrm{SP})]\!]')$.

ii) $\mathcal{I}([\![\mathrm{SP}]\!]) =$ *(by the hypothesis)* $\mathcal{I}'(\mathbf{P}(\mathbf{M})([\![\mathrm{SP}]\!])) =$ *(because* $(\mathbf{S}, \mathbf{M})$ *is a deep translation)* $\mathcal{I}'([\![\mathbf{S}(\mathrm{SP})]\!]')$.

iii) From the hypothesis, we have that if $\mathrm{M} \in [\![\mathrm{SP}]\!]$*, then there exists* $\mathrm{M}' \in [\![\mathbf{S}(\mathrm{SP})]\!]'$ *s.t.* $\mathrm{M} \sim \mathrm{M}'$*; and similarly if* $\mathrm{M}' \in [\![\mathbf{S}(\mathrm{SP})]\!]'$*, then there exists* $\mathrm{M} \in [\![\mathrm{SP}]\!]$ *s.t.* $\mathrm{M} \sim \mathrm{M}'$.
$\mathcal{I}([\![\mathrm{SP}]\!]) = \{\mathrm{P} \mid \exists\ \mathrm{M} \in [\![\mathrm{SP}]\!]\ s.t.\ \mathrm{P} \leftrightharpoons \mathrm{M}\} =$
$\{\mathrm{P} \mid \exists\ \mathrm{M}' \in [\![\mathbf{S}(\mathrm{SP})]\!]'\ s.t.\ \mathrm{P} \leftrightharpoons \mathrm{M}'\} = \mathcal{I}'([\![\mathbf{S}(\mathrm{SP})]\!]')$.

iv) Trivial. $\square$

Below we define some rather natural relationships, graphically reported in Fig. 6, among the formalisms of the formal methods presented as examples in Sect. 3 and show which ones may be uplifted to the method level.

- $Id_1, \ldots, Id_4$ are the trivial embeddings between the formalisms (recall that all of them are essentially first-order algebraic specifications).
- Tr_2 translates **SMoLCS-R** specifications into **ASSRS** ones by transforming labels plus states into sets of input messages (states are needed because in **SMoLCS-R** a label may lead to several different states); formally Tr_2 associates with a **SMoLCS-R** specification `SP`, where `state` is its main dynamic sort, the following **ASSRS** specification

```
use SP
sorts input-message-set, output-message-set
```

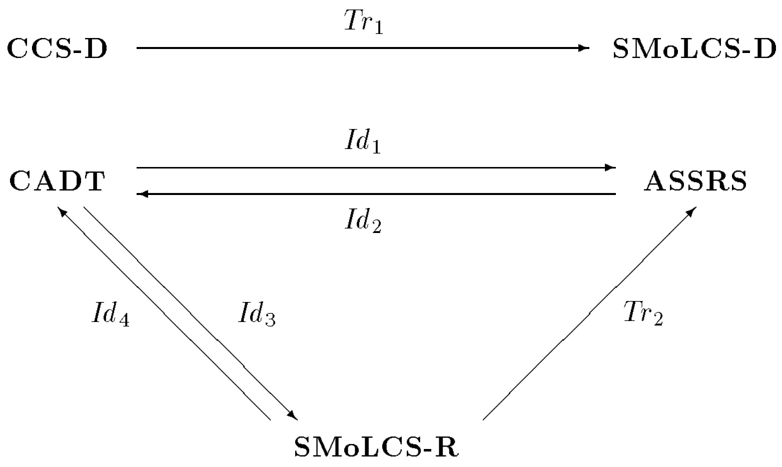


Fig. 6. Relationships between example formalisms

```
opns  Null: -> output-message-set
      <_;_>: lab_state state -> input-message-set
axioms
  forall os: output-message-set . Null = os
  forall is: input-message-set
     exists l: lab_state, s: state . is = < l; s >
  < l; s > = < l'; s' >  iff  l = l' and s = s'
  if  s -- l --> s'  then
    Answer(s,<l,s'>) = Null and  Next-State(s,<l,s'>) = s'
```

- Tr_1 translates a CCS term, whose associated synchronization tree is ST, into a **SMoLCS-D** specification with one dynamic sort `state` and semantics L, s.t. the elements of sort `state` in L bijectively correspond to the states of ST, the elements of sort `lab_state` bijectively correspond to the labels of ST, and the interpretation of `-->` corresponds to the arcs of ST.

Tr_1 is a translation exact up to isomorphism on LTL-structures; and $Id_1, \dots,$ Id_4 and Tr_2 are exact translations.

However not all of them are method simulations.

Tr_1 is a method simulation, indeed both methods have essentially the same modelling.

Id_1, Id_3 are method simulations, indeed both **ASSRS** and **SMoLCS-R** have **CADT** as a submethod for handling the data structures used by the processes.

Id_4 and Id_2 are not method simulations, because the specifications of processes become specifications of static data structures.

Tr_2 is not a method simulation, because the specification of a process sending and receiving numbers becomes the specification of a process offering no reaction to whatever stimuli. Notice that there is no way to simulate **SMoLCS-R** with **ASSRS**, because the first does not offer a way to distinguish within an interaction of a process with the external world what is received and what is sent outside; instead the correct simulation of **ASSRS** by **SMoLCS-R** has been given before in Sect. 5.1.

5.3 Replacing the formalism in a method

Sometimes an existing method FM with formalism F has to be modified to use a different formalism F′; for example, because the original one is no more supported, or a new one is equipped with more software tools.

How to recover or 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 relationship between F and F′, and then derive a modified method, which is a simulation of the original one.

Let $\mathrm{F} = (\mathcal{M}, \mathcal{SPEC}, [\![\_]\!])$ and $\mathrm{F}' = (\mathcal{M}', \mathcal{SPEC}', [\![\_]\!]')$ be the old and the new formalisms respectively; and let $(\mathbf{S}, \mathbf{M})$ be a deep translation of F into F′.

To get a new method we just need to define a new modelling

$\mathrm{P} \leftrightharpoons' \mathrm{M}' \Leftrightarrow$ there exists $\mathrm{M} \in \mathcal{M}$ s.t. $\mathrm{P} \leftrightharpoons \mathrm{M}$ and $\mathrm{M}' = \mathbf{M}(\mathrm{M})$;

thus the new formal method FM′ has the following relevant components

$(\mathcal{EP}, \mathcal{M}', \leftrightharpoons', [\![\_]\!]', \{\mathbf{S}(\mathrm{SP}) \mid \mathrm{SP} \in \mathcal{SPEC}\} \subseteq \mathcal{SPEC}')$.

Proposition 3 *If* $\mathbf{M}$ *is compatible with* $\leftrightharpoons$*, that is*
for all $\mathrm{P} \in \mathcal{EP}$*,* $\mathrm{M}, \overline{\mathrm{M}} \in \mathcal{M}$*, if* $\mathrm{P} \leftrightharpoons \mathrm{M}$ *and* $\mathbf{M}(\overline{\mathrm{M}}) = \mathbf{M}(\mathrm{M})$*, then* $\mathrm{P} \leftrightharpoons \overline{\mathrm{M}}$*,*
then FM′ *is a method and* $\mathbf{S}$ *is a simulation of* FM *by* FM′*.*

PROOF. To see that FM′ is a method we have to prove **1)**, **2)** and **3)** of Sect. 2.4.1.

1) Assume $\mathrm{P}_1 \leftrightharpoons' \mathrm{M}'_1$, $\mathrm{P}_2 \leftrightharpoons' \mathrm{M}'_1$, $\mathrm{P}_2 \leftrightharpoons' \mathrm{M}'_2$; thus there exist M_1, $\overline{\mathrm{M}}_1$ and M_2 s.t. $\mathrm{P}_1 \leftrightharpoons \mathrm{M}_1$, $\mathrm{P}_2 \leftrightharpoons \overline{\mathrm{M}}_1$ and $\mathrm{P}_2 \leftrightharpoons \mathrm{M}_2$, $\mathrm{M}'_1 = \mathbf{M}(\mathrm{M}_1)$, $\mathrm{M}'_1 = \mathbf{M}(\overline{\mathrm{M}}_1)$ and $\mathrm{M}'_2 = \mathbf{M}(\mathrm{M}_2)$. From the hypothesis $\mathrm{P}_2 \leftrightharpoons \mathrm{M}_2$. Because FM is a method $\mathrm{P}_1 \leftrightharpoons \mathrm{M}_2$ and so $\mathrm{P}_1 \leftrightharpoons' \mathrm{M}'_2$.

2) Assume $\mathrm{P} \leftrightharpoons \mathrm{M}'_1$, $\mathrm{P} \leftrightharpoons \mathrm{M}'_2$ and $\mathrm{M}'_1 \in [\![\mathbf{S}(\mathrm{SP})]\!]'$; thus by definition of $\leftrightharpoons'$ there exist M_1, $\mathrm{M}_2 \in \mathcal{M}$ s.t. $\mathrm{P} \leftrightharpoons \mathrm{M}_1$, $\mathrm{P} \leftrightharpoons \mathrm{M}_2$, $\mathrm{M}'_1 = \mathbf{M}(\mathrm{M}_1)$ and $\mathrm{M}'_2 = \mathbf{M}(\mathrm{M}_2)$.

Because $(\mathbf{S}, \mathbf{M})$ is a deep translation $[\![\mathbf{S}(\mathrm{SP})]\!]' = \mathbf{P}(\mathbf{M})([\![\mathrm{SP}]\!])$, and so there exists $\overline{\mathrm{M}}_1 \in [\![\mathrm{SP}]\!]$ s.t. $\mathrm{M}'_1 = \mathbf{M}(\overline{\mathrm{M}}_1)$. By the hypothesis we have that $\mathrm{P} \leftrightharpoons \overline{\mathrm{M}}_1$; because FM is a method (prop. **2)**) $\mathrm{M}_2 \in [\![\mathrm{SP}]\!]$, and so $\mathrm{M}'_2 \in [\![\mathbf{S}(\mathrm{SP})]\!]'$.

3) Assume $\mathrm{M} \in [\![\mathrm{SP}']\!]' = [\![\mathbf{S}(\mathrm{SP})]\!]' = \mathbf{P}(\mathbf{M})([\![\mathrm{SP}]\!])$, thus there exists $\overline{\mathrm{M}}$ s.t. $\mathrm{M} = \mathbf{M}(\overline{\mathrm{M}})$ with $\overline{\mathrm{M}} \in [\![\mathrm{SP}]\!]$. Because FM is a method there exists P s.t. $\mathrm{P} \leftrightharpoons \overline{\mathrm{M}}$, and so $\mathrm{P} \leftrightharpoons' \mathrm{M}$.

Now we show that $(\mathbf{M}, \mathbf{S})$ is a simulation of FM by FM$'$. Let $\mathrm{SP} \in \mathcal{SPEC}$,
$\mathcal{I}'([\![\mathbf{S}(\mathrm{SP})]\!]') = \{\mathrm{P} \mid \exists\, \mathrm{M}' \in [\![\mathbf{S}(\mathrm{SP})]\!]', \mathrm{P} \leftrightharpoons' \mathrm{M}'\} =$
$\{\mathrm{P} \mid \exists\, \mathrm{M}' \in [\![\mathbf{S}(\mathrm{SP})]\!]', \mathrm{M} \in \mathcal{M} \text{ s.t. } \mathrm{M}' = \mathbf{M}(\mathrm{M}), \mathrm{P} \leftrightharpoons \mathrm{M}\} =$
$\{\mathrm{P} \mid \exists\, \mathrm{M} \in \mathcal{M}, \mathrm{M}' \in \mathbf{M}([\![\mathrm{SP}]\!]) \text{ s.t. } \mathrm{M}' = \mathbf{M}(\mathrm{M}), \mathrm{P} \leftrightharpoons \mathrm{M}\} =$
$\{\mathrm{P} \mid \exists\, \mathrm{M} \in \mathcal{M}, \overline{\mathrm{M}} \in [\![\mathrm{SP}]\!] \text{ s.t. } \mathbf{M}(\overline{\mathrm{M}}) = \mathbf{M}(\mathrm{M}), \mathrm{P} \leftrightharpoons \overline{\mathrm{M}}\} =$
(from the hypothesis) $\{\mathrm{P} \mid \exists\, \overline{\mathrm{M}} \in [\![\mathrm{SP}]\!], \mathrm{P} \leftrightharpoons \overline{\mathrm{M}}\} = \mathcal{I}[\![\mathrm{SP}]\!]$. $\square$

6 Conclusions

We started with some general remarks on the permanent controversy on the role of 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.

Not being the time nor our experience 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 within a method.

The use of the proposed pattern for presenting formal methods allows to enlighten many aspects, frequently kept implicit also in author presentations. Moreover singling out what is the formal part of a method could also be of help in teaching the formalities.

Notice however that a method is nicely presented using our pattern does not mean that is surely a good and valuable one; in other words, a well-presented pattern is a necessary but not a sufficient condition for the value of a method.

The proposed pattern has been applied to handle important issues, namely compositionality and simulation of methods, showing that only at the method level we have sufficient tools for an analysis relevant to the practical use.

Although of preliminary character, we believe that some of the ideas can be exploited in other different contexts and perhaps generalized as a useful conceptual tool. Moreover we hope to have shown that rxploring methodological aspects is a subject of interesting investigation itself. We will welcome useful comments, constructive criticism and suggestions.

Someone may wonder why the mathematics used in this paper is so simple (more or less set theory), and why we do not need more sophisticated mathematical tools, for example, category theory, which has been used for the meta presentation of logical specification formalisms, for example adopting the institution framework. First of all we want the formalities related to formal METHODS to be the simplest possible (thus models should be described in set theoretic way, the structure on formal models, specifications, ... is given in terms of functions). Moreover at the moment that has been enough; it may be that going on we need to use more complex mathematical tools.

However it would have been rather easy to rephrase everything in a more sophisticated setting, for example, turning the various classes of entities into categories (so composition would be modelled by limits in some diagram and we should have functors around instead of functions). But, further studies are needed to see if that could give some advantages (e.g., more compact and elegant ways to present the parts of the pattern, easier way to get results).

6.1 Future work

We think that the idea of using a pattern for describing in a organized way formal methods should be tested on other activities different from "to specify"; the next candidates are "to verify the correctness of a development step" (e.g., from requirement to design, from design to code) and "to validate some artifacts" (e.g., specifications, code). Furthermore we need to build a sensible library of instantiations of the various patterns.

Recently several works are appearing on the topic of combining formalisms and methods or heterogeneous formalisms and methods; we plan to see if our pattern presentation of formal methods may help to explore when and how formalisms and methods may be combined.

We believe that to go on to define patterns for many activities we need to find out an "organized (more or less formal)" way to describe processes happening along the time, as for properly presenting guidelines (which in activity different from "to give a specification" may be the most relevant component of the pattern), representing development processes and the (partial) execution of them (for the activities taking care of development process).

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 M. Cerioli. Relationships between Logical Frameworks. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification*, number 655 in Lecture Notes in Computer Science, pages 126–143. Springer Verlag, Berlin, 1993.

[3] 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.

[4] E. Astesiano and G. Reggio. Formally-Driven Friendly Specifications of Concurrent Systems: A Two-Rail Approach. Technical Report DISI–TR–94–20, DISI – Università di Genova, Italy, 1994. Presented at ICSE'17-Workshop on Formal Methods, Seattle, April 1995.

[5] E. Astesiano and G. Reggio. Labelled Transition Logic: An Outline. Technical Report DISI–TR–96–20, DISI – Università di Genova, Italy, 1996.

[6] E. Astesiano and G. Reggio. Formalism and Method. In M. Bidoit and M. Dauchet, editors, *Proc. TAPSOFT '97*, number 1214 in Lecture Notes in Computer Science, pages 93–114. Springer Verlag, Berlin, 1997.

[7] 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.

[8] D. Bjorner, S. Kousoube, R. Noussi, and G. Satchok. Michael Jackson's Problem Frames: Towards Methodological Principles of Selecting and Applying Formal Software Development Techniques and Tools. In M.G. Hinchey and Liu ShaoYing, editors, *Proc. Intl.Conf. on Formal Engineering Methods, Hiroshima, Japan, 12-14 Nov.1997*, pages 263–270. IEEE CS Press, 1997.

[9] M. Broy and C. Jones. Editorial. *Formal Aspects of Computing*, 8(1–2), 1996.

[10] 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.

[11] M. Chandy and J. Misra. *Parallel Program Design: a Foundation*. Addison-Wesley, 1988.

[12] G. Costa and G. Reggio. Specification of Abstract Dynamic Data Types: A Temporal Logic Approach. *T.C.S.*, 173(2):513–554, 1997.

[13] 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.

[14] 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.

[15] A. Finkelstein, J. Kramer, and B. Nuseibeh, editors. *Software Process Modelling and Technology*. John Wiley & Sons, 1994.

[16] 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.

[17] W. Wayt Gibbs. Software's Chronic Crisis. *Scientific American*, (9):72–81, 1994.

[18] J.A. 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.

[19] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, London, 1985.

[20] 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.

[21] 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.

[22] I.S.O. ISO 8807. Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. IS, International Organization for Standardization, 1989.

[23] M. Jackson. *Software Requirements & Specifications: a lexicon of practice, principles and prejudices*. Addison-Wesley, 1995.

[24] Luqi. Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development. In *Proc. of "1994 Montery Workshop"*. U.S. Naval Postgraduate School - Monterey California, 1994.

[25] R. Milner. *Communication and Concurrency.* Prentice Hall, London, 1989.

[26] N.F.M. Roozenburg and J. Eekels. *Product design: Fundamentals and Methods.* Wiley, 1995.

[27] 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.

[28] D. Sannella and A. Tarlecki. Specifications in an Arbitrary Institution. *Information and Computation*, 76, 1988.

[29] J. Sommerville. *Software Engineering: Third Edition.* Addison-Wesley, 1989.

[30] 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.

[31] C. Stirling. Modal and Temporal Logics. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. 2*, pages 477–563. Clarendon Press, Oxford, 1992.

[32] H. van Vliet. *Software Engineering: Principles and Practice.* John Wiley & Sons, 1993.

[33] R.J. Wieringa. *Requirements Engineering: Frameworks for Understanding.* Wiley, 1996. Available at `http://www.cs.utwente.nl/~roelw/RE1.ps`.

[34] M. Wirsing. Algebraic Specifications. In J. van Leeuwen, editor, *Handbook of Theoret. Comput. Sci.*, volume B, pages 675–788. Elsevier, 1990.

[35] 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.