Integrating formal methods by unifying abstractions

Raymond Boute

INTEC, Ghent University, Belgium,
boute@intec.UGent.be,

Abstract. Integrating formal methods enhances their power as an intellectual tool in modelling and design. This holds regardless of automation, but a fortiori if software tools are conceived in an integrated framework.

Among the many approaches to integration, most valuable are those with the widest potential impact and least obsolescence or dependency on technology or particular tool-oriented paradigms. From a practical view, integration by unifying models leads to more uniform, wider-spectrum, yet simpler language design in automated tools for formal methods.

Hence this paper shows abstractions that cut across levels and boundaries between disciplines, help unifying the growing diversity of aspects now covered by separate formal methods and mathematical models, and even bridge the gap between "continuous" and "discrete" systems. The abstractions also yield conceptual simplification by hiding non-essential differences, avoiding repeating the same theory in different guises.

The underlying framework, not being the main topic, is outlined quite tersely, but enough for showing the preferred formalism to express and reason about the abstract paradigms of interest. Three such paradigms are presented in sufficient detail to appreciate the surprisingly wide scope of the obtained unification. The function extension paradigm is useful from signal processing to functional predicate calculus. The function tolerance paradigm spans the spectrum from analog filters to record types, relational databases and XML semantics. The coordinate space paradigm covers modelling issues ranging from transmission lines to formal semantics, stochatic processes and temporal calculi. One conclusion is that integrated formal methods are best served by calculational tools.

Keywords. Abstraction, continuous, discrete, hybrid systems, databases, filters, formal methods, function extension, function tolerance, integration, predicate calculus, quantification, semantics, signal processing, transmission lines, unification, XML.

1 Introduction: motivation, choice of topic and overview

Applying formal methods to complex systems may involve modelling different aspects and views that are often expressed by different paradigms. Insofar as complexity is due mainly to quantitative elements (e.g., size of the state space),

past decades have seen impressive progress in the capabilities of automated tools, where we single out model checking as a representative example [20, 32, 45].

However, when complexity is due to the need for combining different modelling viewpoints [3, 28, 44], tools are too domain-specific, each reflecting a particular paradigm in a way not well-suited towards conceptual combination. Moreover, automation always carries a certain risk of entrenching ad hoc paradigms and poor conceptualizations, the equivalent of legacy software in programming. Both the tool developer and the user tend to preserve the invested effort, and successes with near-term design solutions curtail incentive for innovation. Hence much existing tool support fails to exploit the advantages that formality brings.

Integration is not in the first place a matter of tools, but of careful thinking about concepts, abstractions and formalisms before starting to think about tools.

From an engineering perspective, there is an instructive analogy with automated tools in classical engineering disciplines, such as mechanics and electronics. These disciplines are mainly based on physical phenomena, which are best modelled by methods from (linear) algebra and analysis or calculus. The use of well-known automated tools such as Maple, Mathematica, MATLAB, Mathcad (and more specialized ones such as SPICE) is very widespread, comparatively much more so than the use of tools for formal methods in software engineering. We attritubute this to two major factors, namely

a. The abstractions: for modelling physical phenomena, algebra and analysis have tremendous power of abstraction. For instance, one differential equation can model vastly different phenomena, yielding effortless integration.
b. The formalisms[1]: the notation and rules supported by software tools are those that have proven convenient for human communication and for pencil-and-paper calculations that are essentially *formal*[2]. Still, this only refers to calculating with derivatives and integrals; the logical arguments in analysis are quite informal, causing a severe style breach (addressed below).

So tool design *follows* well-designed abstractions and formalisms. Wide-scope abstractions, formalisms that are convenient for formal calculation by hand (not just when automated) and style continuity are hallmarks of mature integration.

In computing (hardware, software), formal logic has always been the basis for mathematical modelling, and is now supported by good formal tools [40, 42]. Although these tools conceptually have a wider scope than, say, model checking, they do not play the same role as those mentioned for classical engineering.

a. The abstractions: the level where the generality of formal logic is exercised differs from that of algebra and analysis. In fact, there is a strong case for using formal logic in analysis to eliminate the style breach, which is now made possible in an attractive way by advances in calculational logic. However, this

[1] A *formalism* is a language/notation together with rules for symbolic manipulation.
[2] This means manipulating expressions on the basis of their *form*, using precise rules, unlike the common way based on *meaning* (intuitive interpretation). In this way, the shape of the expressions provides guidance in calculations and proofs.

only shows that logic by itself does not constitute the desired wide-spectrum paradigm, but needs to be complemented by other mathematical concepts.

b. The formalisms: logics supported by current tools are by no means convenient for pencil-and-paper calculation or human communication[3]. Here also is a severe style breach: the mathematics used in studying principles and algorithms for the tools themselves is highly informal (e.g., the use of quantifiers and set comprehension reflects all the deficiencies outlined in section 2) and the proofs in even the best treatments are mere plausibility arguments.

Here the tools *impose* the abstractions and formalisms, often quite narrow ones.

As an aside: Lamport [35] correctly observes that, for systems specification, mathematics is more appropriate than program-like notation. The latter misses the power of declarativity (necessary for abstraction and going beyond discrete processes) and convenience for pencil-and-paper calculation (for which it is too verbose). Integrating formal methods and tools makes similar demands.

The theme description for this conference explicitly mentions the following approaches to integrating different viewpoints: creating hybrid notations, extending existing notations, translating between notations, incorporating a wider perspective by innovative use of existing notation.

Of course, these are not mutually exclusive. The approach used here contains some flavour of all, but most emphatically the latter. Referring to aforementioned elements (abstractions and formalisms), it is characterized as follows.

a. A wider perspective is offered by abstractions that unify paradigms from the continuous and the discrete world, often in surprising and inspiring ways. The basis is *functional predicate calculus* [15] and *generic functionals* [16]; the latter is the main layer of mathematical concepts complementing logic.

b. Existing notation is embedded in a general formalism that eliminates ambiguities and inconsistencies, provides useful new forms of expression at no extra cost, and supports formal calculation, also "by hand". It fully eliminates the style breach in classical mathematics as well as in formal methods.

The unification also entails considerable conceptual simplification. We shall see how the concepts captured by our abstractions are usually known only in various different guises, the similarities hidden by different notations, and properties derived separately for each of these guises. Abstractions allow doing the work once and for all. As observed in [5], *Relief [in coping with monumental growth of usable knowledge] is found in the use of abstraction and generalization [using] simple unifying concepts. This process has sometimes been called "compression".* This very effective epistemological process also reduces fragmentation.

Overview The underlying framework is not the main topic here, but a rather terse outline is given in section 2 to provide the wider context.

The main part of the paper deals with abstractions and paradigms, the former being formalized versions of the latter, stripped from their domain-specific

[3] Auxiliary tools translating proofs into text are only a shallow patch, making things worse by adding verbosity, not structure. The real issue is a matter of proof style.

connotations. We select some of the most typical unifying abstractions from which the general approach emerged. The topics chosen uncover insights derived from the continuous world which reveal surprising similarities with seemingly disparate discrete concepts, not vaguely, but in a prescise mathematical form.

For each topic, we start with a modelling aspect of analog systems, extend it in a direct way to a general abstraction, and then show modelling applications in the discrete world of computing. The first topic (section 3) goes from analog adders and modulators in signal processing and automatic control via a function extension operator to predicate calculus. The second one (section 4) goes from analog filter characteristics via a functional generalization of the Cartesian product to record types, databases and XML semantics. The third one (section 5) goes from distributed systems via lumped ones to program semantics.

Related subjects and ramifications are pointed out along the way. We conclude with notes on the advantages of such far-reaching unifications in the theory and practice of formal methods, tool design, and education in CS and EE.

2 The basic formalism as outlined in the Funmath LRRL

This section explains the formalism used in the sequel. Yet, we shall mostly use our syntax in the conservative mode of synthesizing only common and familiar notations. In this way, most of the notation from section 3 onward will appear entirely familiar, unless (with due warning) the extra power of expression is used.

Hence readers uninterested in formalisms may gloss over section 2, and come back later. Others may also be interested in the wider context. Indeed, the formalism is designed to cover both continuous and discrete mathematics in a formal way with (a) a minimum of syntactic constructs (b) a set of rules for formal calculational reasoning (by hand, as in [21, 24–27]). Therefore this section gives a first idea of how item (a) of this rather ambitious goal is achieved, while item (b) is the main topic of a full course [15]. Since the best compact outline is the "Funmath Language Rationale and Reference Leaflet", we reproduce here its main portion, taken verbatim (without frills) from an annex to [15].

Rationale A formal mathematical language is valuable insofar as it supports the design of precise calculation rules that are convenient in everyday practice.

In this sense, mathematical conventions are strong in Algebra and Analysis (e.g., rules for $\int$ in every introductory Analysis text), weaker in Discrete Mathematics (e.g., rules for $\sum$ only in very few texts), and poor in Predicate Logic (e.g., disparate conventions for $\forall$ and $\exists$, rules in most logic texts impractical). This is reflected in the degree to which everyday calculation in these areas can be called "formal", and inversely proportional to the needs in Computer Science.

Entirely deficient are the conventions for denoting sets. Common expressions such as $\{m \in \mathbb{N} \mid m < n\}$ and $\{2 \cdot n \mid n \in \mathbb{Z}\}$ seem innocuous, but exposing their structure as $\{v \in X \mid p\}$ and $\{e \mid v \in X\}$ (with the metavariables below) reveals the ambiguity: $\{n \in \mathbb{N} \mid n \in \mathbb{Z}\}$ matches both. Calculation rules are nonexistent.

Funmath (Functional Mathematics) is not "yet another computer language" but an approach to structure formalisms by conceiving mathematical objects

as functions whenever convenient — which is quite more often than common practice reflects. Four constructs suffice to synthesize most (all?) common conventions without their ambiguities and inconsistencies, and also yield new yet useful new forms of expression, such as point-free expressions. This leaflet covers only syntax and main definitions; calculation rules are the main topic of [15].

Syntax To facilitate adopting this design in other formalisms, we avoid a formal grammar. Instead, we use metavariables: i for a (tuple of) identifiers, and for expressions: v, w: (tuple of) variable(s); d, e: arbitrary; p, q, r: boolean; X, Y: set; f, g: function; P, Q: predicate; F, G: family of functions; S, T: family of sets. By "family of X" we mean "X-valued function". Here are the four constructs.

0. An *identifier* can be any (string of) symbol(s) except markers (binding colon and filter mark, abstraction dot), parentheses ( ), and keywords (**def**, **spec**).
   Identifiers are *declared* by *bindings* $i : X \mathbin{\mathbb{A}} p$, ("$i$ in X satisfying p"). The *filter* $\mathbin{\mathbb{A}} p$ (or **with** p) is optional, e.g., $n : \mathbb{N}$ and $n : \mathbb{Z} \mathbin{\mathbb{A}} n \geq 0$ are the same.
   Definitions, of the form **def** *binding*, declare *constants*, with global scope. Existence and uniqueness are proof obligations, which is not the case for *specifications*, of the form **spec** *binding*. Example: **def** $roto : \mathbb{R}_{\geq 0}$ **with** $roto^2 = 2$. Well-established symbols (e.g., $\mathbb{B}$, $\Rightarrow$, $\mathbb{R}$, $+$, $\sqrt{\ }$) are predefined constants.
1. An *abstraction* (*binding* . *expression*) denotes a *function*. The identifiers declared are *variables*, with local scope. Writing f for $v : X \mathbin{\mathbb{A}} p \,.\, e$, the domain axiom is $d \in \mathcal{D}\, f \;\equiv\; d \in X \wedge p[^v_d]$ and the mapping axiom $d \in \mathcal{D}\, f \Rightarrow f\, d = e[^v_d]$. Here $e[^v_d]$ is e with d substituted for v. Example: $n : \mathbb{Z} \,.\, 2 \cdot n$.
2. A *function application* has the form $f\, e$ in the default *prefix* syntax. When binding a function identifier, dashes can specify other conventions, e.g., $\text{—}\star\text{—}$ for infix. Prefix has precedence over infix. Parentheses are used for overriding precedence rules, *never* as an operator. Application may be partial: if $\star$ is an infix operator, then $(a\star)$ and $(\star b)$ satisfy $(a\star)\, b = a \star b = (\star b)\, a$. *Variadic application*, of the form $e \star e' \star e'' \star e'''$, is explained below.
3. *Tupling*, of the form e, e', e'' (any length n), denotes a function with domain $0..n-1$ and mapping illustrated by $(e, e', e'')\, 0 = e$ and $(e, e', e'')\, 1 = e'$ etc.

Macros can define shorthands in terms of the basic syntax, but few are needed. Shorthands are d^e for $d \uparrow e$ (exponent) and d_e for $d \downarrow e$ (filtering, see below). Sugaring macros are $e \mid v : X \mathbin{\mathbb{A}} p$ for $v : X \mathbin{\mathbb{A}} p \,.\, e$ and $v : X \mid p$ for $v : X \mathbin{\mathbb{A}} p \,.\, v$, and finally $v := e$ for $v : \iota\, e$. The *singleton set injector* ι has axiom $d \in \iota\, e \;\equiv\; d = e$.

Functions A function f is defined by its *domain* $\mathcal{D}\, f$ and its *mapping* (unique image for every domain element). Skipping a technicality [15], equality is axiomatized by $f = g \Rightarrow \mathcal{D}\, f = \mathcal{D}\, g \wedge (e \in \mathcal{D}\, f \cap \mathcal{D}\, g \Rightarrow f\, e = g\, e)$ and its converse, the inference rule $\mathcal{D}\, f = \mathcal{D}\, g \wedge (v \in \mathcal{D}\, f \cap \mathcal{D}\, g \Rightarrow f\, v = g\, v) \vdash f = g$ (new variable v).

Example: the *constant function definer* $^\bullet$ with $X^\bullet\, e = v : X \,.\, e$ (v not free in e); near-trivial, but very useful. Special instances: the *empty* function $\varepsilon := \emptyset^\bullet\, e$ (any e, by equality) and the *one-point function definer* $\mapsto$ with $d \mapsto e = \iota\, d^\bullet\, e$.

Predicates are $\mathbb{B}$-valued functions. Here $\mathbb{B} = \{0, 1\}$, some prefer $\mathbb{B} = \{\text{F}, \text{T}\}$.

Pragmatics We show how to exploit the functional mathematics principle and synthesise common notations, issues that are not evident from mere syntax.

(a) *Elastic operators* originally are functionals designed to obviate common ad hoc abstractors like $\sum_{i=m}^{n}$, $\forall\, v : X$, $\lim_{x \to a}$, but the concept is more general.

The *quantification* operators ($\forall$, $\exists$) are defined by $\forall\, P \;\equiv\; P = \mathcal{D}\, P^{\bullet}\, 1$ and $\exists\, P \;\equiv\; P \neq \mathcal{D}\, P^{\bullet}\, 0$. Observe synthesis of familiar forms in $\forall\, P \;\equiv\; \forall\, x : \mathcal{D}\, P \,.\, P\, x$ and $\forall\, x : \mathbb{R} \,.\, x^2 \geq 0$ but also new forms as in $\forall\, (p, q) = p \wedge q$ and $\exists\, (p, q) = p \vee q$.

For every infix operator $\star$ an *elastic extension* E is designed such that $x \star y = E\, (x, y)$. Evident are $\bigcup$ and $\bigcap$ for $\cup$ and $\cap$ (e.g., $e \in \bigcap\, S \;\equiv\; \forall\, x : \mathcal{D}\, S \,.\, e \in S\, x$), more interesting are $\sum$ for $+$ (see [15]) and the next extensions for $=$ and $\neq$.

The predicate con (*constancy*) with $\mathsf{con}\, f \;\equiv\; \forall\, x : \mathcal{D}\, f \,.\, \forall\, y : \mathcal{D}\, f \,.\, f\, x = f\, y$ and inj (*injectivity*) with $\mathsf{inj}\, f \;\equiv\; \forall\, x : \mathcal{D}\, f \,.\, \forall\, y : \mathcal{D}\, f \,.\, f\, x = f\, y \Rightarrow x = y$ follow the same design principle. Properties are $\mathsf{con}\, (d, e) \;\equiv\; d = e$ and $\mathsf{inj}\, (d, e) \;\equiv\; d \neq e$.

The *(function) range* operator $\mathcal{R}$ has axiom $e \in \mathcal{R}\, f \;\equiv\; \exists\, x : \mathcal{D}\, f \,.\, f\, x = e$. Using $\{\text{—}\}$ as a synonym for $\mathcal{R}$ synthesizes set notations such as $\{m : \mathbb{N} \mid m < n\}$ and $\{2 \cdot n \mid n : \mathbb{Z}\}$. We never abuse $\in$ for binding, so $\{n : \mathbb{N} \mid n \in \mathbb{Z}\}$ has no ambiguity. Expressions like $\{e, e', e''\}$ also have their usual meaning. Rules are derived via $\exists$. We use $\mathcal{R}$ in defining the *function arrow* $\rightarrow$ by $f \in X \rightarrow Y \;\equiv\; \mathcal{D}\, f = X \wedge \mathcal{R}\, f \subseteq Y$. For the *partial arrow*, $f \in X \nrightarrow Y \;\equiv\; \mathcal{D}\, f \subseteq X \wedge \mathcal{R}\, f \subseteq Y$.

Variadic function application is alternating an infix operator with arguments. We *uniformly* take this as standing for the application of a matching elastic operator to the argument list. Examples: $p \wedge q \wedge r \;\equiv\; \forall\, (p, q, r)$ and $e = e' = e'' \;\equiv\; \mathsf{con}\, (e, e', e'')$. An example of a new opportunity is $e \neq e' \neq e'' \;\equiv\; \mathsf{inj}\, (e, e', e'')$.

Traditional ad hoc abstractors have a "range" attached to them, as in $\sum_{i=m}^{n}$. Elastic operators subsume this by the domain of the argument. This *domain modulation* principle is supported by the generic *function/set filtering* operator $\downarrow$ defined by $f_P = x : \mathcal{D}\, f \cap \mathcal{D}\, P \wedge P\, x \,.\, f\, x$ and $X_P = \{x : X \cap \mathcal{D}\, P \mid P\, x\}$.

(b) *Generic functionals* [16] extend often-used functionals to *arbitrary* functions by lifting restrictions. For instance, function inversion f^{-} traditionally requires $\mathsf{inj}\, f$ and composition $f \circ g$ traditionally requires $\mathcal{R}\, g \subseteq \mathcal{D}\, f$. We discard all restrictions on the argument functions by defining the domain of the result function such that its image definition is free of out-of-domain applications, e.g., $f \circ g = x : \mathcal{D}\, g \wedge g\, x \in \mathcal{D}\, f \,.\, f\, (g\, x)$. For the main generic functionals, see [16].

3 From signal processing to predicate calculus: the function extension paradigm

a. Introduction of the paradigm by example First, operations defined on instantaneous values are extended formally to signals, in order to express the behaviour of memoryless components. Next, we generalize this by including domain information to obtain a generic functional applicable to all kinds of functions. Thirdly, we illustrate its use in a functional variant of predicate calculus.

(i) The *starting point* is the description of the *behaviour* of certain systems in terms of *signals*, i.e., functions of type $\mathbb{T} \rightarrow A$ (or $\mathcal{S}_A$) where $\mathbb{T}$ is a time domain and A a set of instantaneous values.

In communications engineering [19] and automatic control [23], the simplest basic blocks are memoryless devices realizing arithmetic operations. Usually the extension of arithmetic operators to signals done implicitly by writing $(x+y)\,t = x\,t + y\,t$. However, we shall see it pays off using an explicit *direct extension* operator $\widehat{\ }$, e.g., for signals x and y with $(x \mathbin{\widehat{+}} y)\,t = x\,t + y\,t$.

(ii) The *generalization step* consists in making $\widehat{\ }$ *generic* [16], i.e., applicable to *all* infix operators $\star$ and all functions f and g by suitably defining $f \mathbin{\widehat{\star}} g$. The criterion for suitability, as motivated in [14], is that the domain for $f \mathbin{\widehat{\star}} g$ must be defined such that the image definition does not contain any out-of-domain applications. It is easy to see that this requirement is satisfied by defining

$$\begin{aligned} &x \in \mathcal{D}\,(f \mathbin{\widehat{\star}} g) \;\equiv\; x \in \mathcal{D}\,f \cap \mathcal{D}\,g \wedge (f\,x, g\,x) \in \mathcal{D}\,(\star) \\ &x \in \mathcal{D}\,(f \mathbin{\widehat{\star}} g) \Rightarrow (f \mathbin{\widehat{\star}} g)\,x = (f\,x) \star (g\,x). \end{aligned} \tag{1}$$

A noteworthy example is *equality*: $(f \mathbin{\widehat{=}} g) = x : \mathcal{D}\,f \cap \mathcal{D}\,g \,.\, f\,x = g\,x$, hence $f \mathbin{\widehat{=}} g$ is a predicate on $\mathcal{D}\,f \cap \mathcal{D}\,g$.

(iii) The *particularization step* to applications in predicate and quantifier calculus uses the fact that our predicates are *functions* taking values in $\{0, 1\}$. We shall also use the *constant function specifier* $^\bullet$: for any set X and any e,

$$\mathcal{D}\,(X^\bullet e) = X \quad \text{ and } \quad x \in X \Rightarrow (X^\bullet e)\,x = e. \tag{2}$$

Our *quantifiers* $\forall$ and $\exists$ are predicates over predicates: for any predicate P,

$$\forall\,P \;\equiv\; P = \mathcal{D}\,P^\bullet 1 \quad \text{ and } \quad \exists\,P \;\equiv\; P \neq \mathcal{D}\,P^\bullet 0. \tag{3}$$

These simple definitions yield a powerful algebra with dozens of calculation rules for everyday practical use [15]. Here we mention only one *theorem* illustrating the role of $\widehat{\ }$, namely $\forall\,P \wedge \forall\,Q \Rightarrow \forall\,(P \mathbin{\widehat{\wedge}} Q)$. Here is a calculational proof.

$$\begin{aligned} \forall\,P \wedge \forall\,Q \equiv\;& \langle \text{Def. } \forall \rangle \quad P = \mathcal{D}\,P^\bullet 1 \wedge Q = \mathcal{D}\,Q^\bullet 1 \\ \Rightarrow\;& \langle \text{Leibniz} \rangle \quad \forall\,(P \mathbin{\widehat{\wedge}} Q) \;\equiv\; \forall\,(\mathcal{D}\,P^\bullet 1 \mathbin{\widehat{\wedge}} \mathcal{D}\,Q^\bullet 1) \\ \equiv\;& \langle \text{Def. } \widehat{\ } \rangle \quad \forall\,(P \mathbin{\widehat{\wedge}} Q) \;\equiv\; \forall\,x : \mathcal{D}\,P \cap \mathcal{D}\,Q \,.\, (\mathcal{D}\,P^\bullet 1)\,x \wedge (\mathcal{D}\,Q^\bullet 1)\,x \\ \equiv\;& \langle \text{Def. } ^\bullet \rangle \quad \forall\,(P \mathbin{\widehat{\wedge}} Q) \;\equiv\; \forall\,x : \mathcal{D}\,P \cap \mathcal{D}\,Q \,.\, 1 \wedge 1 \\ \equiv\;& \langle \forall\,(X^\bullet 1) \rangle \; \forall\,(P \mathbin{\widehat{\wedge}} Q) \;\equiv\; 1 \end{aligned}$$

This theorem has a conditional converse: $\mathcal{D}\,P = \mathcal{D}\,Q \Rightarrow \forall\,(P \mathbin{\widehat{\wedge}} Q) \Rightarrow \forall\,P \wedge \forall\,Q$.

b. Some clarifying remarks on predicates It is not always customary in logic to view propositions (formulas) as boolean expressions, or predicates as boolean functions. However, this view is common in programming languages, and gives booleans the same status as other types. In fact, we make a further unification with the rest of mathematics, viewing booleans as a restriction of arithmetic to $\{0, 1\}$, as in [13]. In a similar approach, Hehner [30] prefers $\{-\infty, +\infty\}$. We chose $\{0, 1\}$ because it merges with modulo 2 arithmetic, facilitates counting and obviates characteristic functions in combinatorial and word problems.

Introducing the constants $\{0, 1\}$ (or $\{\text{F}, \text{T}\}$) in traditional logic is somewhat confusing at first, because $p \equiv 1$ is exchangeable with p. This seeming difficulty disappears, however, be pondering the associativity of $\equiv$ in $p \equiv (p \equiv 1)$.

Another issue, raised also by Lamport [35, page 14] is that $x > 1$, taken out of context, can be either a *formula* depending on x, or a (provable) *statement* depending on the hypotheses. In metamathematics, one often uses $\vdash$ to make explicit that provability is meant. However, as Lampson notes, in regular (yet careful) mathematical discourse, symbols like $\vdash$ are not used since the intent (formula versus statement) is clear from the context.

Finally, note that our functional $\forall$ does not "range" over variables but is a predicate (boolean function) over predicates, as in $\forall P$. The familiar variables enter the picture when P is an abstraction of the form $x : X \,.\, p$, where p is a formula, so $\forall x : X \,.\, p$ has familiar form and meaning. Using *functionals* (like $\forall$) rather than *ad hoc* abstractors (like $\forall x$) is the essence of our elastic operators.

c. Final remarks on direct extension More recently, the basic concept of direct extension also appears in programming. In the program semantics of Dijkstra and Scholten [21], operators are assumed extended implicitly to structures, e.g., the arithmetic $+$ extends to structures, as in $(x + y).n = x.n + y.n$. This applies even to equality, i.e. if x and y are structures, then $x = y$ does not denote equality of x and y but a function with $(x = y).n \equiv x.n = y.n$. The concept of *polymorphism* in the graphical programming language LabVIEW [6] designates a similar implicit extension. Implicit extension is reasonable in a restricted area of discourse, but it is overly rigid for general practice. An explicit operator offers more flexibility and allows generalization according to our design principles by specifying the result types.

We mention also that *function composition* ($\circ$) is made generic according to the same requirement by defining, for any functions f and g,

$$\begin{aligned} &x \in \mathcal{D}\,(f \circ g) \;\equiv\; x \in \mathcal{D}\,g \wedge g\,x \in \mathcal{D}\,f \\ &x \in \mathcal{D}\,(f \circ g) \Rightarrow (f \circ g)\,x = f\,(g\,x). \end{aligned} \tag{4}$$

The *(simplex) direct extension* ($\overline{=}$) for extending single argument functions can now be defined by $\overline{f}\,g = f \circ g$.

Observe also that, since tuples are functions, $f \circ (x, y) = f\,x, f\,y$. This property subsumes the "map" operator for functional programming.

All these operators entail a rich collection of algebraic laws that can be expressed in point-free form, yet preserve the intricate domain refinements (as can be verified calculationally). Examples are $f \circ (g \circ h) = (f \circ g) \circ h$ and $\overline{h \circ g} = \overline{h} \circ \overline{g}$ and $(\widehat{\star}) = \overline{(\star)} \circ (\&)$. Elaboration is beyond the scope of this paper.

4 From analog filters to record types: the function tolerance paradigm

a. Introduction of the paradigm by example Starting with the usual way of specifying the frequency/gain characteristic of a RF filter, we formalize the concept of *tolerance* for functions and generalize it to arbitrary sets. The resulting generic functional, when particularized to discrete domains, subsumes the familiar Cartesian product with a functional interpretation. Combination with enumeration

types expresses record types (in a way quite different from, but "mappable" to, the formulation with projection or selector functions used in Haskell). It is sufficiently general for capturing all types necessary to describe abstract syntax, directory structures, and XML documents. Again we proceed in three steps.

(i) The *starting point* is the specification of analog filter characteristics, for instance gain as a function of frequency. For continuous systems, accuracy of measurements and tolerances on components are an important issue. To extend this notion to functions in a formal way, it suffices to introduce a *tolerance function* T that specifies, for every value x in its domain (e.g., frequency), the set $T\,x$ of allowable values (e.g., for the filter gain). More precisely, we say that a function f (e.g., a specific filter characteristic) *meets* the tolerance T iff

$$\mathcal{D}\,f = \mathcal{D}\,T \wedge \forall\, x : \mathcal{D}\,f \cap \mathcal{D}\,T \,.\, f\,x \in T\,x.$$

This principle, illustrated in Fig. 1, provides the setting for the next two steps.

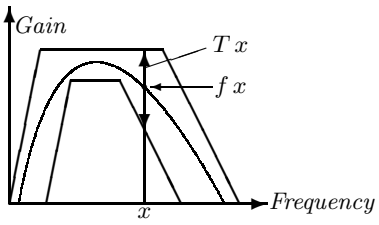

Fig. 1. A bandpass filter characteristic

(ii) The (small) *generalization step* is admitting *any* (not just "dense") sets for $\mathcal{D}\,T$. This suggests defining an operator $\bigtimes$ such that, if T is a set-valued function, $\bigtimes T$ is the set of functions meeting tolerance T:

$$f \in \bigtimes T \;\equiv\; \mathcal{D}\,f = \mathcal{D}\,T \wedge \forall\, x : \mathcal{D}\,f \cap \mathcal{D}\,T \,.\, f\,x \in T\,x \tag{5}$$

Observe the analogy with the definition of function equality:

$$f = g \;\equiv\; \mathcal{D}\,f = \mathcal{D}\,g \wedge \forall\, x : \mathcal{D}\,f \cap \mathcal{D}\,g \,.\, f\,x = g\,x.$$

With the axiom $x = y \;\equiv\; x \in \iota\, y$ for the *singleton set injector* ι, this yields calculationally $f = g \;\equiv\; f \in \bigtimes(\iota \circ g)$, hence $\bigtimes$ can also specify *exactly*.

(iii) *Particularization step:* Instantiate (5) with $T := A, B$ (two sets). Then

$$f \in \bigtimes(A, B) \;\equiv\; \mathcal{D}\,f = \mathbb{B} \wedge f\,0 \in A \wedge f\,1 \in B$$

by calculation. Hence $\bigtimes(A, B) = A \times B$, the usual Cartesian product (considering tuples as functions). This also explains the notation $\bigtimes$ and the name *generalized functional Cartesian product* (abbreviated *funcart* product).

As usual, $\bigtimes$ defines variadic shorthand for $\times$, as in $A \times B \times C = \bigtimes(A, B, C)$. Applied to abstractions, as in $\bigtimes a : A \,.\, B\,a$, it covers so-called *dependent types* [29],

in the literature often denoted by ad hoc abstractions like $\prod_{a\in A} B\,a$. We also introduce the suggestive shorthand $A \ni a \to B\,a$ for $\bigtimes a : A\,.\,B\,a$, which is especially convenient in chained dependencies, e.g. $A \ni a \to B\,a \ni b \to C\,a\,b$.

b. Important properties In contrast with ad hoc abstractors like $\prod_{a\in A} B\,a$, the operator $\bigtimes$ is a genuine functional and has many useful algebraic properties. Most noteworthy is the inverse. By the axiom of choice, $\bigtimes T \neq \emptyset \equiv \forall\, x : \mathcal{D}\,T\,.\,T \neq \emptyset$. This also characterizes the bijectivity domain of $\bigtimes$ and, if $\bigtimes T \neq \emptyset$, then $\bigtimes^-(\bigtimes T) = T$. For the usual cartesian product this implies that, if $A \neq \emptyset$ and $B \neq \emptyset$, then $\bigtimes^-(A \times B) = A, B$, hence $\bigtimes^-(A \times B)\,0 = A$ and $\bigtimes^-(A \times B)\,1 = B$. Finally, an explicit image definition is

$$\bigtimes^- S = x : Dom\,S\,.\,\{f\,x \mid f : S\} \tag{6}$$

for any nonempty S in the range of $\bigtimes$, where $Dom\,S$ is the common domain of the functions in S (extracted, e.g., by $Dom\,S = \bigcap f : S\,.\,\mathcal{D}\,f$).

In fact, the funcart operator is the "workhorse" for typing all structures unified by functional mathematics [12, 13]. Obviously, $A \to B = \bigtimes(A \bullet B)$, so it covers all "ordinary" function types as well.

c. Aggregate data types and structures Let $\square n = \{m : \mathbb{N} \mid m < n\}$ for n in $\mathbb{N}' := \mathbb{N} \cup \iota\,\infty$. For any set A and n in $\mathbb{N}'$, define $A \uparrow n$ (or A^n) by $A \uparrow n = \square n \to A$, hence $A^n = \bigtimes(\square n \bullet A)$, the n-fold product. We also define $A^* = \bigcup n : \mathbb{N}\,.\,A^n$.

Apart from sequences, the most ubiquitous aggregate data type are *records* in the sense of PASCAL [34].

One approach for expressing records functionally is using *selector functions* corresponding to the field labels, where the records themselves appear as arguments. We have explored this alternative some time ago in a different context [8], and it is also currently used in Haskell [33]. However, it does not make records themselves into functions and has a rather heterogeneous flavor.

Therefore our preferred alternative is the $\bigtimes$ operator from (5), whereby records are defined as *functions* whose domain is a set of field labels constituting an *enumeration type*. For instance

$$Person := \bigtimes(name \mapsto \mathbb{A}^* \cup age \mapsto \mathbb{N}),$$

where *name* and *age* are elements of an enumeration type, defines a function type such that the declaration $employee : Person$ specifies $employee\,name \in \mathbb{A}^*$ and $employee\,age \in \mathbb{N}$. The syntax can be made more attractice by defining, for instance, an elastic type definition operator Record with $\mathsf{Record}\,F = \bigtimes(\bigcup F)$, so we can write $Person := \mathsf{Record}\,(name \mapsto \mathbb{A}^*, age \mapsto \mathbb{N})$.

Observe the use of *function merge* $\cup$. A full discussion of this operator [16] is beyond the scope of this paper. However, it suffices to know that, if $f \mathbin{©} g$, then $\mathcal{D}\,(f \cup g) = \mathcal{D}\,f \cup \mathcal{D}\,g$ and $x \in \mathcal{D}\,f \Rightarrow (f \cup g)\,x = f\,x$ (similarly for g). *Compatibility* for functions is defined by $f \mathbin{©} g \equiv \forall\, x \in \mathcal{D}\,f \cap \mathcal{D}\,g\,.\,f\,x = g\,x$.

As mentioned, other structures are also defined as functions. For instance, *trees* are functions whose domains are *branching structures*, i.e., sets of sequences

describing the path from the root to a leaf in the obvious way. This covers any kind of branch labeling. For instance, for a binary tree, the branching structure is a subset of $\mathbb{B}^*$. Classes of trees are characterized by restrictions on the branching structures. The $\times$ operator can even specify types for leaves individually.

Aggregates defined as functions inherit all elastic operators for which the images are of suitable type. For instance, $\sum s$ sums the fields or leaves of any number-valued record, tree or other structure s.

d. Application to relational databases Database systems are intended to store information and present a convenient interface to the user for retrieving the desired parts and for constructing and manipulating "virtual tables" containing precisely the information of interest in tabular form.

Code	Name	Instructor	Prerequisites
CS100	Basic Mathematics for CS	R. Barns	
MA115	Introduction to Probability	K. Jason	MA100
CS300	Formal Methods in Engineering	R. Barns	CS100, EE150
...	...	...	

A *relational database* presents the tables as relations. One can view each row as a tuple, and a collection of tuples of the same type as a relation.

However, in the traditional nonfunctional view of tuples, components can be accessed only by a separate indexing function using natural numbers. This is less convenient than, for instance, the column headings. The usual patch consists in "grafting" onto the relational scheme so-called *attribute names* corresponding to column headings. Disadvantages are that the mathematical model is not purely relational any more, and that operators for handling tables are ad hoc.

Viewing the table rows as *records* in functional sense as before allows embedding in a more general framework with useful algebraic properties and inheriting the generic operators. For instance, the table shown can be declared as $GCI : \mathcal{P}\ CID$, a set of *course information descriptors* whose type is defined by

$$\textbf{def}\ CID := \mathsf{Record}\,(\mathsf{code} \mapsto Code, \mathsf{name} \mapsto \mathbb{A}^*, \mathsf{inst} \mapsto Staff, \mathsf{prrq} \mapsto Code^*).$$

Since in our formalism table rows are functions, queries can be constructed by functionals. As an example, we show how this is done for the most subtle of the usual query constructs in database languages, the *("natural") join*.

We define the operator $\bowtie$ combining tables S and T by uniting the domains of the elements (i.e., the field names), but keeping only those records for which the same field name in both tables have the same contents, i.e., only *compatible* records are combined. In other words,

$$S \bowtie T = \{s \cup t \mid (s,t) : S \times T \wedge s \mathbin{©} t\}$$

e. Application to XML documents The following is a simple example of a DTD (Document Type Definition) and a typical instance (here without attributes).

```
                                        <?xml version = "1.0"?>
<!ELEMENT book (title, author*,year?>   <book>
<!ELEMENT title (#PCDATA)>              <title>Trees</title>
<!ELEMENT author (#PCDATA)>             <author>V. Green</author>
<!ELEMENT year (#PCDATA)>               <year>2003</year>
                                        </book>
```

The DTD semantics and the instance can be expressed mathematically as follows, with minor tacit simplifications (a complete discussion is given in [46]).

$$\mathsf{booktype} = \mathsf{Record}\,(\mathtt{title} \mapsto \mathbb{A}^*, \mathtt{author} \mapsto (\mathbb{A}^*)^*, \mathtt{year} \mapsto \iota\,\mathsf{unspecified} \cup \mathbb{A}^*)$$
$$\mathsf{bookinst} = \bigcup(\mathtt{title} \mapsto \text{“}\mathtt{Trees}\text{”}, \mathtt{author} \mapsto \tau\ \text{“}\mathtt{V.\ Green}\text{”}, \mathtt{year} \mapsto \text{“}2003\text{”})$$

The operator τ, for expressing sequences of length 1, is defined by $\tau\, x = 0 \mapsto x$.

5 From transmission lines to program semantics: the coordinate space paradigm

a. Introduction of the paradigm by example Starting with the well-known telegraphists' equation for transmission lines, we impose a structuring on the parameters involved (voltage, curent, time, space), and show how discretization of the space coordinate covers the usual models for lumped circuits and the notion of *state* in formal semantics. As mentioned earlier, we do this in three steps.

(i) The *starting point* is the typical modelling of dynamical systems in physics and engineering. The example chosen is the simplest model of a transmission line consisting of two wires, with a load at one end, as depicted in Fig. 2. Voltage v

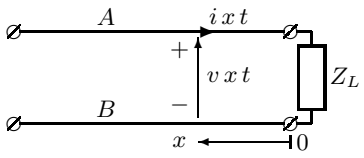

Fig. 2. Introducing the coordinate space paradigm

and current i, with the conventions shown, are functions of type $\mathbb{S} \to \mathbb{T} \to \mathbb{U}$ where $\mathbb{S}$ is the *spatial coordinate* space (say, $\mathbb{R}_{\geq 0}$, for the distance from the load), $\mathbb{T}$ is the *temporal coordinate* space ($\mathbb{R}_{\geq 0}$ or $\mathbb{R}$, as desired) and $\mathbb{U}$ is the *instantaneous value* space (voltage, current, typically $\mathbb{R}$). With these conventions, $v\,x\,t$ and $i\,x\,t$ denote the voltage and the current at location x at time instant t.

Our formulation as higher-order functions (Currying) facilitates defining integral transforms, e.g. for a lossless line, in terms of incident and reflected wave:

$$v\,x\,t = \overrightarrow{v}\,0\,(t + x/c) + \overleftarrow{v}\,0\,(t - x/c)$$
$$V\,x\,f = \overrightarrow{V}\,0\,f \cdot e^{j \cdot k_f \cdot x} + \overleftarrow{V}\,0\,f \cdot e^{-j \cdot k_f \cdot x}$$

Here $V : \mathbb{S} \to \mathbb{R} \to \mathbb{C}$ is the Fourier transform of v with $V\,x\,f = \mathcal{F}\,(v\,x)\,f$, and $k_f = 2 \cdot \pi \cdot f/c$. However, it is the time domain formulation $\mathbb{S} \to \mathbb{T} \to \mathbb{U}$ that provides the setting for the following two steps.

(ii) The (small) *generalization step* consists in admitting *any* (not only dense) coordinates, e.g. refining the above model by introducing a set $W := \{A, B\}$ of names for the wires and new functions for the potential and the current.

$$v'\_ : W \to \mathbb{R}_{\geq 0} \to \mathbb{T} \to \mathbb{V} \text{ related to } v \text{ by } v'_A\,x\,t - v'_B\,x\,t = v\,x\,t$$
$$i'\_ : W \to \mathbb{R}_{\geq 0} \to \mathbb{T} \to \mathbb{I} \text{ related to } i \text{ by } i'_A\,x\,t = i\,x\,t \wedge i'_B\,x\,t = -(i\,x\,t)$$

Discrete coordinates are used in *systems semantics* [10, 11] to express the semantics of a language for describing *lumped* adirectional systems [12].

The order of appearance of the space and time coordinates is a matter of convenience. In electronics, one often uses the more "neutral" Cartesian product, writing $v : \mathbb{S} \times \mathbb{T} \to \mathbb{V}$ and $v\,(x, t)$. Higher-order functions support 2 variants:

- The *signal space* formulation: a *signal* is a function from $\mathbb{T}$ to $\mathbb{U}$, and quantities of interest are described by functions of type $\mathbb{S} \to \mathbb{T} \to \mathbb{U}$.
- The *state space* formulation: a *state* is a function from $\mathbb{S}$ to $\mathbb{U}$, and quantities of interest are described by functions of type $\mathbb{T} \to \mathbb{S} \to \mathbb{U}$.

Here $\mathbb{U}$ denotes the universe of values of interest (voltages in our example).

(iii) The *particularization step* to program semantics now simply consists in defining the spatial coordinate space $\mathbb{S}$ to be the set of identifiers introduced in the declarations as variables, $\mathbb{T}$ to be a suitable discrete space for the locus of control (in the syntax tree of the program) and $\mathbb{U}$ the value space specified in the declarations. We define the *state space* $S := \mathbb{S} \to \mathbb{U}$ or, as a refinement expressing dependence on the type declared for each variable, $S := \mathbb{S} \ni v \to \mathbb{U}_v$.

For instance, if `x`, `y` and `z` are declared as variables, then $\mathbb{S} = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$. We can express the effect of a given command by an equation for the dynamical function $st : \mathbb{T} \to S$ to express the relationship between the state at the (abstract) time t of execution and the state at time $next\,t$ after execution. E.g., for the assignment

‘`z := x + y`’

the function $st : \mathbb{T} \to \mathbb{S} \to \mathbb{U}$ satisfies the equation[4]

$$st\,(next\,t)\,v = (v = \mathtt{z})\,?\,st\,t\,\mathtt{x} + st\,t\,\mathtt{y} \mid st\,t\,v. \tag{7}$$

We can eliminate v by two general-purpose axiliary operators . *Function overriding* ($\oslash$) is defined as follows. For any functions f and g, the domain of $f \oslash g$ is given by $\mathcal{D}\,(f \oslash g) = \mathcal{D}\,f \cup \mathcal{D}\,g$, and the image for any x in this domain by $(f \oslash g)\,x = x \in \mathcal{D}\,f\,?\,g\,x \mid g\,x$. The operator $\mapsto$ allows writing a function whose domain consists of the single element d, having image e, as $d \mapsto e$. As expected, $((d \mapsto e) \oslash g)\,x = (x = d)\,?\,e \mid g\,x$ for any x in $\iota\,e \cup \mathcal{D}\,g$. Hence (7) becomes

$$st\,(next\,t) = (\mathtt{z} \mapsto st\,t\,\mathtt{x} + st\,t\,\mathtt{y}) \oslash st\,t. \tag{8}$$

[4] The conditional of the form $c\,?\,b \mid a$, read: "if c then b else a", is self-explanatory.

Since $st\,(next\,t)$ depends on t only via $st\,t$ (this is the case in general, not just in this example), one can conveniently eliminate time by expressing the effect of every command as a *state transformation*, viz. a function of type $S \to S$. We combine these functions into one by including the command as a parameter and defining a *meaning function* $\mathcal{C} : C \to S \to S$, where C is the set of commands. For instance, the semantics of our assigment is expressed by

$$\mathcal{C}\ \text{`z := x + y'}\ s = (\mathtt{z} \mapsto s\,\mathtt{x} + s\,\mathtt{y}) \oslash s. \tag{9}$$

This is the familiar formulation of denotational semantics [39, 48].

Dependence of the locus of control on data can be conveniently expressed by adapting a simple technique from hydrodynamics (relating paths to streamlines) to abstract syntax trees (not elaborated here), whereas environments can be formulated as coordinate transformation.

b. Other applications along the way Viewing variables as coordinate values raises the question: are they really *variables*? From the unifying viewpoint (and in denotational semantics) they are not: only the *state* varies with time!

This is clarified by the following example, which also illustrates the ramifications for hardware description languages. Consider the digital device (a gate) and the analog device (an operational amplifier) from Fig. 3.

What is the role of the labels $x\,y\,z$ and $x'\,y'\,z'$ on the terminals?

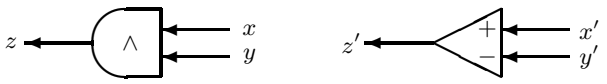

Fig. 3. Devices with labelled terminals

Many choices are possible, for instance

- Names of terminals (just labels in an alphabet).
- Instantaneous values, e.g., boolean ones in $z = x \wedge y$, real ones in $z' = x' - y'$.
- Signals (time functions), e.g., of type $\mathbb{T} \to \mathbb{B}$ in $z\,t = x\,t \wedge y\,t$ and of type $\mathbb{T} \to \mathbb{R}$ in $z'\,t = x'\,t + y'\,t$.

Observe that the interpretations are mutually *in*compatible, e.g., the possibility of $x = y$ as *values* conflicts with the obvious fact that $x \neq y$ as *terminals*. Furthermore, using, for instance, x' and y' as function names may require letting $x' = \sin$ for one *gedanken* experiment (or real experiment) and $x' = \cos$ in another. Such "context switching" looks more like assignment in imperative programming than mathematics. Although often harmless, it limits expression.

The way to support all these views without conflict is the coordinate paradigm. In addition to the space coordinate space $\mathbb{S}$, the time coordinate space $\mathbb{T}$ and the instantaneous value space $\mathbb{U}$, we consider an *experiment index space* Z, which supports distinguishing between experiments, but can often be left implicit. As before, we have the signal space formulation $sg : Z \to \mathbb{S} \to \mathbb{T} \to \mathbb{U}$ and the state space formulation $st : Z \to \mathbb{T} \to \mathbb{S} \to \mathbb{U}$.

In the example of Fig. 3, names of terminals figure as space coordinates. For instance, using the signal space formulation and leaving Z implicit, we make the following typical conventions:

- for the AND-gate: $\mathbb{S} = \{x, y, z\}$ and $\mathbb{T} = \mathbb{N}$ and $\mathbb{U} = \mathbb{B}$
- for the op amp: $\mathbb{S}' = \{x', y', z'\}$ and $\mathbb{T}' = \mathbb{R}$ and $\mathbb{U}' = \mathbb{R}$

The two families of signals $s : \mathbb{S} \to \mathbb{T} \to \mathbb{U}$ and $s' : \mathbb{S}' \to \mathbb{T}' \to \mathbb{U}'$ satisfy respectively

$$s\, z\, n = s\, x\, n + s\, y\, n \quad \text{and} \quad s'\, z'\, t = s'\, x'\, t + s'\, y'\, t. \tag{10}$$

or, equivalently, using *direct extensions*: $s\, z = s\, x \mathbin{\widehat{\wedge}} s\, y$ and $s'\, z' = s'\, x' \mathbin{\widehat{+}} s'\, y'$. Direct extension is a topic for later; all that needs to be understood now is that it "extends" an operator $\star$ over instantaneous values to an operator $\widehat{\star}$ over signals by $(f \mathbin{\widehat{\star}} g)\, t = f\, t \star g\, t$. This is such standard practice in communications and control engineering [19, 23] that the extension operator $\widehat{(\ )}$ is usually left implicit.

c. Defining stochastic processes with the coordinate paradigm Consider the signal space formulation

$$sg : Z \to \mathbb{S} \to \mathbb{T} \to \mathbb{U}$$

and assume a probability measure on Z is defined

$$Pr : \mathcal{P}\, Z \to [0, 1].$$

Then Z (seen as the index set for all possible experiments) will be called a *sample description space* as in [43].

The transpose[5], of sg, namely sg^T, is of type $\mathbb{S} \to Z \to \mathbb{T} \to \mathbb{U}$ and will be called a *family of stochastic processes*. For every x in $\mathbb{S}$ there is one such process $sg^T x$, of type $Z \to \mathbb{T} \to \mathbb{U}$. The distribution function, assuming $\mathbb{U} := \mathbb{R}$, is then defined in the following fashion, where the image definition is the familiar one from the theory of stochastic processes [36, 41].

$$\textbf{def}\ F_{\_} : \mathbb{S} \to \mathbb{T} \to \mathbb{U} \to [0, 1]\ \textbf{with}\ F_x\, t\, \eta = Pr\, \{\zeta : Z \mid sg\, \zeta\, x\, t \leq \eta\}$$

For hardwired systems, $\mathbb{S}$ is the set of (real or conceptual) terminals, and the model coincides with the classical one. In program semantics, letting $\mathbb{S}$ be the set of variables and defining a probability measure on Z yields *stochastic semantics*.

c. Functional temporal calculus We show how the formulation from [9] fits into the coordinate space paradigm. Again letting $sg : Z \to \mathbb{S} \to \mathbb{T} \to \mathbb{U}$ where Z is an index set for all possible experiments, we call the transpose sg^T a *family of temporal variables*. Each temporal variable $sg^T x$ (for given x in $\mathbb{S}$) is a function of type $Z \to \mathbb{T} \to \mathbb{U}$. A *temporal operator* is a function of type

$$\mathbb{O} := (Z \to \mathbb{T} \to \mathbb{U}) \to (Z \to \mathbb{T} \to \mathbb{W})$$

[5] Transposition is another generic operator not discussed here; for the special case $f : A \to B \to C$, the transpose f^T is of type $B \to A \to C$ and satisfies $f^T\, b\, a = f\, a\, b$.

i.e. from temporal variables to temporal variables. We consider two kinds of temporal operators.

A *temporal mapping* $g : \mathbb{O}$ is defined such that $g\, v\, \zeta\, t = f\,(v\, \zeta\, t)$ for some $f : \mathbb{U} \to \mathbb{W}$. It can be used to model the input/output behaviour of a *memoryless* system. Typical temporal mappings are *direct extensions* of arithmetic ($+$, $-$ etc.), propositional ($\wedge$, $\vee$ etc.) and other operators of interest.

A *temporal combinator* is a temporal operator that is not a temporal mapping. Typical temporal combinators are the *next* ($\bigcirc$), *always* ($\Box$) and *sometime* ($\Diamond$) operators defined such that, for every temporal variable $v : Z \to \mathbb{T} \to \mathbb{U}$, every $\zeta : \mathbb{Z}$ and every $t : \mathbb{T}$ (where $\mathbb{T}$ is assumed ordered) by

$$\begin{array}{ll}
\bigcirc\, v\, \zeta\, t = v\, \zeta\, (\mathit{next}\, t) & (\text{e.g., } \mathit{next}\, t = t + 1 \quad \text{for} \quad \mathbb{T} := \mathbb{N}) \\
\Box\, v\, \zeta\, t = \forall\, t' : \mathbb{T}_{\geq t} \,.\, v\, \zeta\, t' & (\text{here } \mathbb{U} = \mathbb{W} = \mathbb{B}) \\
\Diamond\, v\, \zeta\, t = \exists\, t' : \mathbb{T}_{\geq t} \,.\, v\, \zeta\, t' & (\text{here } \mathbb{U} = \mathbb{W} = \mathbb{B})
\end{array}$$

Various choices for $\mathbb{T}$ (discrete, continuous, partial orderings, "branching") and matching definitions are possible, depending on the aspect to be modelled. For instance, for discrete $\mathbb{T}$ with a *next* operator and $t' \geq t \;\equiv\; \exists\, n : \mathbb{N} \,.\, t' = \mathit{next}^n\, t$, clearly $\Box\, v\, \zeta\, t = \forall\, n : \mathbb{N} \,.\, v\, \zeta\, (\mathit{next}^n\, t)$.

Since all these operators are defined as higher-order functions, they can be used in the point-free style with respect to the dummies in Z and $\mathbb{T}$, resulting in expressions of the form (omitting $\widehat{\ }$)

$$\Box\, (w \Rightarrow \bigcirc\, w) \Rightarrow w \Rightarrow \Box\, w.$$

Hence we can eliminate all explicit reference to time, but also refer to time when systems modelling requires doing so.

The formulas without reference to time are formally identical to those in certain variants of temporal logic [37]. Temporal logic is more abstract in the sense that a given variant may have several models, whereas temporal calculus is a single model in itself. On the other hand, for certain applications pertaining to concrete systems, working within a concrete model may be necessary to keep an explicit relationship with other system aspects. For instance, it is shown in [9] how the *next* operator $\bigcirc$ can be directly related to the z-transform, an important technique in discrete signal processing and control systems engineering [23, 36].

An extension of the functional temporal calculus with a collection of auxiliary functionals [17] is currently being applied to formally specify so-called *patterns* in Bandera [22], a system for modelling concurrent programs written in Java.

6 Concluding remarks

We have shown how suitable abstractions unify concepts in very disparate fields.

As one referee aptly observed, the mathematics often seems far removed from the area of discourse (e.g., without our explanation, $\bigtimes T$ seems unrelated to analog filters) and reasoning amounts to "playing with mathematics".

This is related to a phenomenon that Wigner calls the "unreasonable effectiveness of mathematics" [47]. For instance, the "mechanical" differential equation $m \cdot \mathrm{D}^2\, x + k \cdot x = 0$ (for mass m, spring constant k) and the "electrical" one $L \cdot \mathrm{D}^2\, i + i/C = 0$ (for inductor L, capacitor C) are both captured by the form $a \cdot \mathrm{D}^2\, f + c \cdot f = 0$. The particulars of the domain of discourse disappear, and one can reason mathematically without distraction by irrelevant concerns.

The "unreasonable effectiveness of mathematics" is directly useful to formal methods. Indeed, rather than designing different methods particular to various application domains (reflecting their idiosyncrasies), unifying models remove irrelevant differences and suggest more generic formal methods whereby the designer can concentrate on exploiting the reasoning power of mathematics.

Admittedly, this goes against the grain of some trends in tool design advocating "being as close as possible to the language of the application domain". However, it has taken physics a few thousand years to realize the advantages of the opposite way: translating the concepts from the application domain into mathematics (Goethe notwithstanding). Computer science is considerably younger but need not wait a thousand years: once the example has been set, the learning time can be shortened. In fact, much useful mathematics is already there.

Anyway, unification by abstraction provides an important intellectual asset, but when it is also applied to the design of automated tools to support formal methods, it can lead to considerably more commonality and wider scope.

Regarding the preferable *style* of such tools, consider the following calculation examples taken from two typical engineering textbooks, namely [7] and [18].

$$
\begin{aligned}
&\frac{1}{n}\sum_{\mathbf{x}} p^n(\mathbf{x}|\theta) l_n(\mathbf{x}) \\
&\leq \frac{1}{n}\sum_{\mathbf{x}} p^n(\mathbf{x}|\theta)[1 - \log q^n(\mathbf{x})] \\
&= \frac{1}{n} + \frac{1}{n} L(\mathbf{p}^n; \mathbf{q}^n) + H_n(\theta) \\
&= \frac{1}{n} + \frac{1}{n} d(\mathbf{p}^n, \mathcal{G}) + H_n(\theta) \\
&\leq \frac{2}{n} + H_n(\theta)
\end{aligned}
\qquad\qquad
\begin{aligned}
F(s) &= \int_{-\infty}^{+\infty} e^{-|x|} e^{-i2\pi x s} dx \\
&= 2\int_0^{+\infty} e^{-x} \cos 2\pi x s \; dx \\
&= 2 \ \mathrm{Re} \int_0^{+\infty} e^{-x} e^{i2\pi x s} dx \\
&= 2 \ \mathrm{Re} \ \frac{-1}{i2\pi s - 1} \\
&= \frac{2}{4\pi^2 s^2 + 1}.
\end{aligned}
$$

The style is *calculational*, and in the second example even purely *equational*, since only equality occurs. Not surprisingly, this is also the most convenient style to formally manipulate expressions in the various unifying system models.

By contrast, tools for logic nearly all use variants of classical formal logic, which amounts to quite different styles such as "natural" reasoning, sequent calculus, tableaux and so on. These styles are acceptable for internal representation but are not suited (nor systematically used) for hand calculation. As pointed out in [26], this seriously hampers their general usefulness. We could add the comment that *the best choice of language or style for any automated tool is one whose formal manipulation rules are convenient for hand calculation as well.*

An aside observation is the following Turing-like test for integration: well-integrated formal methods can formally describe not only target systems but also the concepts and implementations of various tools in a way that is convenient for exposition, *formal* reasoning and proving properties about these tools.

Work by Dijkstra [21], Gries [24–27] and others shows conclusively that *calculational logic* meets this criterion. Its style is similar to the preceding engineering calculation examples, with $\equiv$ and $\Rightarrow$ as logical counterparts of $=$ and $\leq$. Thereby formal logic becomes a practical tool for everyday use, which explains why it has has found wide acceptance in the computing science community during the recent years. Its usefulness would even gain from automated support but, as pointed out in [38], considerable work remains to be done in this direction.

While developing our unifying models, we found calculational logic to merge "seamlessly" with classical algebra and analysis (as used in more traditional physics-based engineering models), thereby closely approximating Leibniz's ideal. The resulting common ground not only increases the scope, but also considerably lowers the threshold for introduction in industry. This should perhaps be a major consideration in designing tools to support formal engineering methods.

Such a tool could have a core based on substitution and equality (Leibniz's rule of "equals for equals") and including function abstraction, surrounded by a layer of propositional calculus, generic functionals [16] and functional predicate calculus [15], and a second layer implementing mathematical concepts as developed in this paper for unifying system models.

Part of this rationale also underlies B [2]. Differences are our use of generic functions, the functional predicate calculus, and the application to "continuous" mathematics. Similarly, the scope of inification in Hoare and Jifeng's Unified Theories of Programming [31] is the discrete domain of programming languages.

The concepts presented are also advantageous in education. Factoring out common aspects avoids unnecessary replication, while stimulating the ability to think at a more abstract level. As a fringe benefit, this creates additional room for other topics, which is necessary in view of the rapid technological developments and the limited time available in most curricula, or even the reduction in time as imposed by the Bachelor/Master reform throughout Europe.

References

1. Chritiene Aarts, Roland Backhouse, Paul Hoogendijk, Ed Voermans, Jaap van der Woude, *A relational theory of data types*. Report, Eindhoven University (1992)
2. Jean-Raymond Abrial, *B-Book*. Cambridge University Press (1996)
3. Rajeev Alur, Thomas A. Henzinger, Eduardo D. Sontag, eds., *Hybrid Systems III*, LNCS 1066. Springer-Verlag, Berlin Heidelberg (1996)
4. Henk P. Barendregt, *The Lambda Calculus — Its Syntax and Semantics*, North-Holland (1984)
5. Hyman Bass, "The Carnegie Initiative on the Doctorate: the Case of Mathematics", *Notices of the AMS, Vol. 50*, No. 7, pp. 767–776 (Aug. 2003)
6. Robert H. Bishop, *Learning with LabVIEW*. Addison Wesley Longman (1999)

7. Richard E. Blahut, *Theory and Practice of Error Control Codes*. Addison-Wesley (1984)
8. Raymond T. Boute, "On the requirements for dynamic software modification", in: C. J. van Spronsen, L. Richter, eds., *MICROSYSTEMS: Architecture, Integration and Use* (Euromicro Symposium 1982), pp. 259-271. North Holland (1982)
9. Raymond T. Boute, "A calculus for reasoning about temporal phenomena", *Proc. NGI-SION Symposium 4*, pp. 405–411 (April 1986)
10. Raymond T. Boute, "System semantics and formal circuit description", *IEEE Transactions on Circuits and Systems, CAS–33*, 12, pp. 1219–1231 (Dec. 1986)
11. Raymond T. Boute, "Systems Semantics: Principles, Applications and Implementation", *ACM TOPLAS 10*, 1, pp. 118–155, (Jan. 1988)
12. Raymond T. Boute, "Fundamentals of Hardware Description Languages and Declarative Languages", in: J. P. Mermet, ed., *Fundamentals and Standards in Hardware Description Languages*, pp. 3–38, Kluwer (1993)
13. Raymond T. Boute, *Funmath illustrated: A Declarative Formalism and Application Examples*. Computing Science Institute, University of Nijmegen (July 1993)
14. Raymond T. Boute, "Supertotal Function Definition in Mathematics and Software Engineering", *IEEE Transactions on Software Engineering, Vol. 26*, No. 7, pp. 662–672 (July 2000)
15. Raymond T. Boute, *Functional Mathematics: a Unifying Declarative and Calculational Approach to Systems, Circuits and Programs — Part I: Basic Mathematics*. Course text, Ghent University (2002)
16. Raymond T. Boute, "Concrete Generic Functionals: Principles, Design and Applications", in: Jeremy Gibbons, Johan Jeuring, eds., *Generic Programming*, pp. 89–119, Kluwer (2003)
17. Raymond Boute, Hannes Verlinde, "Functionals for the Semantic Specification of Temporal Formulas for Model Checking", in: Hartmut König, Monika Heiner, Adam Wolisz, eds., *FORTE 2003 Work-in-Progress Papers*, pp. 23–28. BTU Cottbus Computer Science Reports (2003).
18. Ronald N. Bracewell, *The Fourier Transform and Its Applications*, 2nd ed, McGraw-Hill (1978)
19. Ralph S. Carson, *Radio Communications Concepts: Analog*. Wiley (1990)
20. Edmund M. Clarke, O. Gromberg, D. Peled, *Model Checking*. MIT Press (2000)
21. Edsger W. Dijkstra, Carel S. Scholten, *Predicate Calculus and Program Semantics*. Springer (1990)
22. Matthew B. Dwyer, John Hatcliff, *Bandera Temporal Specification Patterns*, `http://www.cis.ksu.edu/santos/bandera/Talks/SFM02/02-SFM-Patterns.ppt`, tutorial presentation at ETAPS'02 (Grenoble) and SFM'02 (Bertinoro), 2002.
23. Gene F. Franklin, J. David Powell, Abbas Emami-Naeini, *Feedback Control of Dynamic Systems*. Addison-Wesley (1986)
24. David Gries, "Improving the curriculum through the teaching of calculation and discrimination", *Communications of the ACM 34*, 3, pp. 45–55 (March 1991)
25. David Gries, Fred B. Schneider, *A Logical Approach to Discrete Math*. Springer (1993)
26. David Gries, "The need for education in useful formal logic", *IEEE Computer 29*, 4, pp. 29–30 (April 1996)
27. David Gries, "Foundations for Calculational Logic", in: Manfred Broy, Birgit Schieder, eds., *Mathematical Methods in Program Development*, pp. 83–126. Springer NATO ASI Series F158 (1997)
28. Robert L. Grossman, Anil Nerode, Anders P. Ravn, Hans Rischel, eds., *Hybrid Systems*, LNCS 736. Springer-Verlag, Berlin Heidelberg (1993)

29. Keith Hanna, Neil Daeche, Gareth Howells, "Implementation of the Veritas design logic", in: Victoria Stavridou, Tom F. Melham, Raymond T. Boute, eds., *Theorem Provers in Circuit Design*, pp. 77–84. North Holland (1992)
30. Eric C. R. Hehner, *From Boolean Algebra to Unified Algebra*. Internal Report, University of Toronto (June 1997, revised 2003)
31. C. A. R. Hoare, He Jifeng, *Unifying Theories of Programming*. Prentice-Hall (1998)
32. Gerard Holzmann, *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley (2003)
33. Paul Hudak, John Peterson, Joseph H. Fasel, *A Gentle Introduction to Haskell 98*. `http://www.haskell.org/tutorial/` (Oct. 1999)
34. Kathleen Jensen, Niklaus Wirth, *PASCAL User Manual and Report*. Springer (1978)
35. Leslie Lamport, *Specifying Systems*, Addison-Wesley (2002).
36. Edward A Lee, David G. Messerschmitt, *Digital Communication* (2nd ed.). Kluwer (1994)
37. Zohar Manna, Amir Pnueli, *The Temporal Logic of Reactive and Concurrent Systems — Specification*. Springer (1992)
38. Panagiotis Manolios, J. Strother Moore, "On the desirability of mechanizing calculational proofs", *Information Processing Letters, Vol. 77*, No. 2–4, pp. 173–179, (2001)
39. Bertrand Meyer, *Introduction to the Theory of Programming Languages*. Prentice Hall (1991)
40. Sam Owre, John Rushby, Natarajan Shankar, "PVS: prototype verification system", in: P. Kapur, ed., *11th Intl. Conf. on Automated Deduction*, pp. 748–752. Springer Lecture Notes on AI, Vol. 607 (1992)
41. Athanasios Papoulis, *Probability, Random Variables and Atochastic Processes*. McGraw-Hill (1965)
42. Lawrence C. Paulson, *Introduction to Isabelle*, Computer Laboratory `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/dist/docs.html`, University of Cambridge, (Feb. 2001)
43. Emanuel Parzen, *Modern Probability Theory and Its Applications*. Wiley (1960)
44. Frits W. Vaandrager, Jan H. van Schuppen, eds., *Hybrid Systems: Computation and Control*, LNCS 1569. Springer (1999)
45. Moshe Y. Vardi, Pierre Wolper, "An automata-theoretic approach to automatic program verification", *Proc. Symp. on Logic in Computer Science*, pp. 322-331 (June, 1986)
46. Hannes Verlinde, *Systematisch ontwerp van XML-hulpmiddelen in een functionele taal*. M.Sc. Thesis, Ghent University (2003)
47. Eugene Wigner, "The Unreasonable Effectiveness of Mathematics in the Natural Sciences", *Comm. Pure and Appl. Math. Vol. 13*, No. I, pp. 1–14 (Feb. 1960) `http://nedwww.ipac.caltech.edu/level5/March02/Wigner/Wigner.html`
48. Glynn Winskel, *The Formal Semantics of Programming Languages: An Introduction*. MIT Press (1993)