ABSTRACT
The gap between modelling techniques for DSP functionality and those for software implementations is widening. This impedes unifying formalisms for analog, digital and software systems. Recovering these opportunities requires declarativity.

A suitable formalism is outlined, based on a mathematical rather than a programming language. Examples show how it unifies continuous and discrete mathematics, from analysis to formal program semantics. The formalism provides crucial advantages in reasoning by calculation about all aspects of SP, and paves the way for software tools of the next generation.

1. INTRODUCTION
1.1. Motivation: the lost grail of declarativity
There is a widening gap between the modelling of SP functionality and of software implementation. The first typically uses well-established formalisms from classical engineering mathematics, but software practices around DSP systems resemble the early days of programming by not using mathematical models for programs or for calculationally deriving programs and their properties [25].

The main cause in DSP is a shift from essentially declarative mature engineering formalisms to “algorithmic thinking” induced by computer implementation, ignoring the declarative mathematical methods for software. This evolution is historically and educationally backward: mathematics evolved from algorithmic concerns (adding numbers) to declarativity (geometry, algebra, analysis), and so does education from grade school to university.

A well-designed declarative DSP language like Silage [18], although now superseded by new concepts, was “a significant improvement over most of its successors”, such as C++ (and SystemC), about which Lamport [21] aptly warns that it may harm the ability to think logically.

Another symptom is very sloppy terminology, blurring the difference between an “algorithm” and an abstract system characteristic to a degree that compares unfavorably with Peyo’s little blue dwarfs calling everything “smurf”.

Indeed, reducing the abstraction level to so-called “executable specifications” wastes valuable opportunities: no programming notation, but only a mathematical one, can achieve the required declarativity [21], especially in DSP.

1.2. Encouraging developments
Encouraging is the growth of hybrid systems formalisms [2, 13, 28], although their style is often too entrenched in traditional logic for linking conveniently to classical engineering mathematics. Here the more practical calculational logic advocated in [14, 16, 8] is better suited.

The need for declarativity has also been emphasized by eminent researchers in the DSP area [23], in the context of making Electrical Engineering and Computer Engineering into a more unified discipline, called ECE [22].

Our own research over the past 15 years is also aimed at unifying EE and CS, starting with mathematical modelling and reasoning. The style in classical engineering is mostly calculational: chaining expressions by relational operators, e.g., equality. In a classic engineering text [12],

\[
F(s) = \int_{-\infty}^{+\infty} e^{-|x|} e^{-i2\pi xs} dx = 2 \int_{0}^{+\infty} e^{-x} \cos 2\pi xs \, dx = 2 \Re \int_{0}^{+\infty} e^{-x} e^{2\pi xs} \, dx = 2 \Re e^{\frac{-1}{2\pi^2 s^2}} = \frac{2}{1 + \frac{1}{\pi^2 s^2}},
\]

every step is based on a clear calculation rule.

By contrast, logical reasoning in everyday practice by mathematicians and engineers is highly informal, and often involves what Taylor [27] calls syncopation, namely using symbols as mere abbreviations of natural language, for instance the quantifier symbols \( \forall \) and \( \exists \) just meaning “for all” and “there exists”, without calculation rules.

We provide a formalism enabling engineers to calculate with quantifiers as fluently as with simple derivatives and integrals. The formalism is free of all defects of common conventions, also those outlined by Lee and Varaiya [23].

The reward is that mathematical models for software become as convenient as those for signals and systems.

1.3. Overview
The language and the calculation rules of the formalism are presented in section 2, representative application examples are given in section 3, conclusions in section 4.
2. THE FORMALISM

2.1. Funmath language design

Poor notation is a stumbling block to formal calculation: if one has to be on guard for the defects, one cannot "let the symbols do the work" [3]. For a critique of typical defects in conventions 'everyone' uses, see [10, 23]. We do not patch defects ad hoc, but generate correct forms by orthogonal combination of just 4 constructs, gaining extra useful forms of expression. The basis is functional. A function $f$ is fully defined by its domain $D$ and its mapping (image definition). Here are the constructs.

Identifiers: nearly any symbol. They are introduced by bindings $i: X \land p$, where $i$ is the (tuple of) identifi(s), $X$ a set and $p$ a proposition. The filter $\land p$ (or with $p$) is optional, e.g., $n: \mathbb{N}$ and $n: \mathbb{Z} \land n \geq 0$ are interchangeable.

Shorthand: $i := e$ stands for $i := e$. We write $i e$, not $\{e\}$, for singleton sets, using $i$ defined by $e' \in i e \equiv e' = e$.

Identifiers can be variables (in an abstraction as below) or constants (introduced by a definition: def binding).

Application: the default is $f e$ for function $f$ and argument $e$; other conventions may be specified in the binding, e.g., $\equiv$ for infix. Parentheses are never operators, but only support for parsing. The precedence are the usual ones. If $f$ is a function-valued function, $f x y$ stands for $(f x) y$.

Partial application of $* \equiv a \ast b$, with $(a \ast b) = a \ast b = (\ast b) a$. Variadic application is $a \ast \ast b \ast c$ etc., always defined as $F(a, b, c)$ for a suitable elastic extension $F$ of $*$. We write abstractions in synthesizing familiar expressions such as $\sum i: 0 \ldots n. q^i$ and $\{m: \mathbb{Z} \mid m < n\}$.

2.2. Equational and calculational reasoning

The equational style of $(1)$ is generalized to the format

$$e \ R \ e'$$

where the $R$ in successive lines are mutually transitive, for instance $\equiv$, etc. in arithmetic, $\equiv$, $\rightarrow$, etc. in logic.

We write $\lceil_x e$ for substitution, as in $(x \cdot y)^{2}_{x+1} = (z+1) \cdot y$.

2.3. Rules for calculating with propositions and sets

Assume the usual propositional operators $\neg$, $\equiv$, $\rightarrow$, $\land$, $\lor$. A practical proposition calculus needs many rules [16]. Implication $\equiv$ is associative, $\rightarrow$ is not. Parentheses in $p \equiv (q \equiv r)$ are optional, so required in $(p \equiv q) \equiv r$. Embedded in arithmetic [4, 5], logic constants are 0, 1.

Leibniz’s principle is $e = e' \Rightarrow d_{e'}^e = d_{e'}^{e'}$.

For sets, $\equiv$ is the basis. Rules are derived ones, e.g., defining $\cap \equiv \{x \in X \cap Y \mid x \equiv x \in X \land x \equiv x \in Y\}$ and $\times$ by $(x, y) \equiv x \times y \equiv x \in X \land y \equiv y \in Y$. Later we define $\{\{\ldots\}, enabling to prove $y \equiv \{x \equiv X \mid p\} \equiv y \in X \land p^y_{X}$.

2.4. Rules for functions and generic functionals

We omit the design decisions, to be found in [6] and [9]. In what follows, $f$ and $g$ are any functions, $P$ any predicate ($\equiv$-valued function, $B := \{0, 1\}$). $X$ any set, $e$ arbitrary.

Function equality and abstraction Equality is defined by $f = g \Rightarrow D f = D g \equiv (x \in D f \cap D g \Rightarrow f x = g x)$ (Leibniz) and by extensionality for the converse. Abstract encapsulates substitution. Formally: domain axiom $d \in D(v: X \land p . e) \equiv d \in X \land p^{d}_{e}$, mapping axiom $d \in D(v: X \land p . e) \Rightarrow (v: X \land p . e) d = e^{d}_{e}$. Equality is characterized via function equality (exercise).

Generic functionals Goals: (a) removing restrictions in common mathematical functionals, (b) making often-used implicit functionals from signal and systems theory explicit. The idea is defining the result domain judiciously.

Case (a) is illustrated by composition $g \circ f$, commonly requiring $\forall g \subseteq D f$. We define, for any functions:

$$f \circ g \equiv x: D g \land g x \in D f . f (g x) .$$

Note: $D(f \circ g) = \{x: D g \mid g x \in D f\}$.

Case (b) is illustrated by the usual implicit generalization of arithmetic functions to signals, traditionally written $(s + s')(t) = s(t) + s'(t)$. We generalize this by (duplex direct extension ($\uparrow$)) for any functions $*$ (infix), $f$, $g$.

$$f \ast g = x: D f \cap D g \land (f x, g x) \in D(*) . f \ast x g x .$$

Similar is half direct extension: for function $f$, any $e$,

$$f \ast e = f \ast (D f \ast e) \equiv e \ast f = (D f \ast e \ast f) .$$

Simplex direct extension ($\downarrow$) is defined by $\downarrow g = f \circ g$.

Filtering $(\downarrow)$ introduces or eliminates arguments:

$$f \downarrow P \equiv x: D f \cap D P \land P . f x .$$

We extend $\downarrow$ to sets: $x \equiv (X \downarrow P) \equiv x \equiv X \cap D P \land P . x$. Writing $a \downarrow b$ for $a \downarrow b$ and using partial application, we get formal rules for useful shorthands like $f_{<n}$ and $\mathbb{R}_{>0}$.

For the common restriction $(\downarrow)$: $f \downarrow X = f \downarrow (X \ast 1)$.

A very important use of generic functionals is supporting the point-free style, i.e., without referring to domain points. The elegant algebraic flavor is illustrated next.
2.5. Rules for predicate calculus and quantifiers

Axioms, forms of expression For any predicate $P$,
\[
\forall P \equiv P = D P \cdot 1 \quad \exists P \equiv P \not\equiv D P \cdot 0
\]  
(7)

Letting $P$ be an abstraction $v : X \cdot p$, yields the familiar form $\forall v : X \cdot p$, as in $\forall x : \mathbb{R} \cdot x^2 \geq 0$. Algebraic laws are most elegantly stated in point-free form. Each has a pointwise (familiar-looking) form using an abstraction.

Derived rules All follow from (7) and function equality. A practical collection is derived in [8, 10]. Here we give only some examples, starting by expressing $f = g$ as
\[
f = g \equiv D f = D g \land \forall x : D f \cap D g \cdot f x = g x
\]  
(8)

Another example is duality (generalizing De Morgan law)
\[
\forall P \equiv \exists (\neg P) \quad \neg (\forall v : X \cdot p) \equiv \exists v : X \cdot \neg p
\]  
(9)

Here are the main distributivity laws. All have duals.

<table>
<thead>
<tr>
<th>Name of the rule</th>
<th>Point-free form</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distribut. $\forall \lor \land$</td>
<td>$\forall (P \land Q) \equiv \forall P \land \forall Q$</td>
</tr>
</tbody>
</table>
| One-point rule   | $\forall P \equiv \neg (\exists v : D P 
\rightarrow P e)$ |
| Trading          | $\forall (P \lor Q) \equiv \forall (Q \Rightarrow P)$ |
| Transposition    | $\forall (\forall \circ R) \equiv \forall (\forall \circ R^T)$ |

2.6. Wrapping up the rule package for function(al)s

Function range We define the range operator $R$ by
\[
e \in R \iff \exists x : D f \cdot f x = e
\]  
(10)

A consequence is the composition rule $\forall P \Rightarrow \forall (P \circ f)$ and $D P \subseteq R f \Rightarrow (\forall (P \circ f) \equiv \forall P)$; in pointwise form $\forall (y : R f) \equiv \forall (x : D f \cdot p^y_{f x})$ ("dummy change").

Set comprehension We define $\{x \mid \text{finitely interchangable with } R\}$. This yields defect-free set notation: expressions like $\{2, 3, 5\}$ and $\text{Even} = \{2 \cdot m \mid m \in \mathbb{Z}\}$ have familiar form and meaning, and all desired calculation rules follow from predicate calculus via Eq. (10). In particular, we can prove $y \in \{v : X \mid p\} \equiv y \in X \cap p^y_{X} \quad (\text{exercise})$.

Function typing The familiar function arrow ($\rightarrow$) is defined by $f : X \rightarrow Y \equiv D f = X \land R f \subseteq Y$. A more refined type is the Functional Cartesian Product ($\times$):
\[
f : T \times T \equiv D f = D T \land \forall x : D f \cdot f x \in T
\]  
(11)

where $T$ is a set-valued function. Note $X \times (X, Y) = X \times Y$ and $X \times (X \times Y) = X \rightarrow Y$. We use $X \rightarrow x - Y$ as shorthand for $X : x \cdot Y$, where $y$ may depend on $x$.

3. Examples

Here we illustrate the very wide scope of the formalism.

3.1. Examples in analysis and continuous systems

Analysis: calculation replacing syncopation We show how traditional proofs that are tedious by syncopation [27] are done calculationally. The example is adjacency [20]. Since predicates of type $\mathbb{R} \rightarrow \mathbb{B}$ yield more elegant formulations than sets (of type $\mathbb{P} \mathbb{R}$), we define the predicate transformer $\text{ad} : (\mathbb{R} \rightarrow \mathbb{B}) \rightarrow (\mathbb{R} \rightarrow \mathbb{B})$ and the predicates $\text{open} : (\mathbb{R} \rightarrow \mathbb{B}) \rightarrow \mathbb{B}$ with
\[
\text{ad} P v \equiv \forall v : \mathbb{R} \cdot P v \Rightarrow \exists v : \mathbb{R} \cdot \exists r : \mathbb{R} \cdot |x - v| < \varepsilon
\]  
open $P v \equiv \forall v : \mathbb{R} \cdot P v \Rightarrow \exists r : \mathbb{R} \cdot \forall x : \mathbb{R} \cdot |x - v| < \varepsilon \Rightarrow P x$
\[
\text{closed } P = \text{open } P
\]  

We prove the closure property $\text{closed } P \equiv \text{ad } P = P$.

Properties of systems Signals over $A$ are functions of type $S_A := T \rightarrow A$ for time domain $T$. A system is a function $s : S_A \rightarrow S_B$. Assume $T$ additive and $A = B = \mathbb{C}$.

Let $\sigma$ be the shift operator with $\sigma \cdot x = x(t + \tau)$. We characterize time invariance by $\forall t : T, s \circ \sigma t = \sigma \circ s$ and linearity by $\forall z : C, \forall c : C, (c \cdot s \cdot z) = c \cdot (z \cdot s)$. We show that the response of a linear and time-invariant system to the parametrized exponential $E_{\tau} : \mathbb{C} \rightarrow \mathbb{C}$ with $E_{\tau} t = e^{\tau t}$ satisfies $s E_{\tau} = s E_0 \cdot \tau \cdot E_{\tau}$. We start by calculating $s E_{\tau} (t + \tau)$ in order to exploit all properties:
\[
s E_{\tau} (t + \tau) = \begin{cases} \text{(Definition } \sigma) \sigma \cdot (s E_{\tau}) t \\ \text{(Time inv. } s) (s \sigma E_0) t \\ \text{(Property } E_{\tau}) s \cdot (E_\tau \cdot s E_0) t \\ \text{(Linearity } s) (E_{\tau} \cdot s^2 E_0) t \\ \text{(Definition } \sigma) E_{\tau} \cdot s E_0 \cdot t \\ \end{cases}
\]  

Substituting $t := 0$ yields $s E_{\tau} = (s E_0 \cdot \tau \cdot E_{\tau}) \tau$ and hence, by function equality, $s E_{\tau} = s E_0 \cdot \tau \cdot E_{\tau}$.

3.2. Functions as a unifying paradigm in SP

Sequences are rarely viewed consistently as functions in DSP, which often even leads to inadequate conventions as pointed out in [23], e.g., denoting a sequence by $x[n]$. Continuous and discrete SP can be unified by always defining sequences as functions, making them inherit the
rich collection of generic functionals. This issue is discussed in [9], including application examples to formal semantics and calculational reasoning for SP-related languages such as LabVIEW. Here we provide only the basics and a different example, namely, about automata.

**Sequences as functions** Define $\square n = \{ m : N \mid m < n \}$ for any $n : N'$, where $N' := N \cup \{ \infty \}$. The set of sequences of length $n$ over a set $A$ is defined by $A^n \uparrow n \rightarrow A$, with shorthand $A^n$. Note: $A^0 = \epsilon$ and $A^\infty = \{ \} \rightarrow A$. Also, $A^* = \bigcup n : N . A^n$ and $A^+ = \bigcup n : N_0 . A^n$ and $A^e = \bigcup n : N ' . A^n$. Finally, recall $\tau a = 0 \rightarrow a$.

We define the *shift* ($\sigma$) for any nonempty sequence $x$ by $\sigma x = n : N \{ n \mid x (n - 1) \}, x (n + 1) \}$. Concatenation is $+ +$, e.g., $(7, e) + + (3, d) = 7, e, 3, d, a$ and $x < a = x + + a$.

**Causal systems** We define *prefix ordering* $\leq$ on $A^*$ by $x \leq y \equiv \exists z : A^* . y = x + + z$. A system $s : A^* \rightarrow B^*$ is *sequential* iff $s \leq y \Rightarrow s x \leq s y$. This captures the notion of causal (better: “non-anticipatory”) behavior.

Function $r : (A^*)^2 \rightarrow B^*$ is a *residual behavior* (rb) function for $s$ iff $s (x + + y) = s x + + r (x, y)$.

**THEOREM:** $s$ is sequential iff it has an rb function.

Proof: starting from the sequentiality side, $\forall (x, y) : (A^*)^2 . x \leq y \Rightarrow s x \leq s y$

1. Define $\leq y \equiv \exists z : A^* . y = x + + z$.
2. Define $\leq y \equiv \exists z : A^* . y = x + + z$.
3. Define $\leq y \equiv \exists z : A^* . y = x + + z$.
4. Define $\leq y \equiv \exists z : A^* . y = x + + z$.
5. Define $\leq y \equiv \exists z : A^* . y = x + + z$.

We used the *function comprehension axiom*: given any relation $R : X \times Y \rightarrow B$, then $\forall x : X . \forall y : Y . R (x, y)$ iff $\exists f : X \rightarrow Y . \forall x : X . R (x, f x)$.

**Application: derivatives and primitives** For sequential systems, we define the derivative operator $D$ by $D s \varepsilon = \varepsilon$ and $s (x < a) = s x + + D s (x < a)$, so $D s (x < a) = r (x, a)$. Properties are shown next, with a striking analogy in analysis (with respective D-and I-operators, of course).

Finally, $\{ x : A^* . r (x, y) \mid x : A^* \}$ is the state space, on which automata realizing $s$ can be defined (exercise).

### 3.3. Modelling programs

**Program equations** Define the state $s$ as the tuple made of the program variables, and $S$ its type. Let’s denote the state before and $s'$ after executing a command; $c = c_1^e$ and $c' = c_2^e$; also, $s, s'$ abbreviates $s : S, s : S$. Let $C$ be the set of commands.

We define $R_{c} : C \rightarrow S^2 \rightarrow \mathbb{B}$ and $T_{c} : C \rightarrow S \rightarrow \mathbb{B}$ such that the effect of command $c$ is described by two equations: $R_{c} (s, s')$ for state change, $T_{c} s$ for termination. Example: for Dijkstra’s *guarded command* language [14],

<table>
<thead>
<tr>
<th>Command $c$</th>
<th>State change $R_{c} (s, s')$</th>
<th>Termination $T_{c} s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v := e$</td>
<td>$s' = (s' _t)^e$</td>
<td>$T_c' s$</td>
</tr>
<tr>
<td>$c'_1$</td>
<td>$\exists s . R_{c_2} (s, s) \wedge R_{c_3} (s, s')$</td>
<td>$\exists i : I . b_i \wedge T_{c_1} s$</td>
</tr>
</tbody>
</table>

For $\text{skip}$: $R_{\text{skip}} (s, s') \equiv s = s$ and $T_{\text{skip}} s = 1$.

For $\text{abort}$: $R_{\text{abort}} (s, s') \equiv 0$ and $T_{\text{abort}} s = 0$. For iteration: do $b \rightarrow c \odot d$ is defined to have the same effect as $\text{if } \neg b \rightarrow \text{skip} \quad b \rightarrow c' \quad \text{do } b \rightarrow c' \odot d$.

**Hoare semantics** *Hoare triple* describes all possible computations for command $c$ (i.e., those in $(S^2)^R$) starting in a state satisfying $A : S \rightarrow B$ (antecedent) and terminating in a state satisfying $P : S \rightarrow B$ (postcondition) by a predicate of type $(S \rightarrow B) \times C \times (S \rightarrow B) \rightarrow B$ defined next for partial correctness (12) and total correctness (13), using $T_{\text{end}} : C \rightarrow (S \rightarrow B) \rightarrow B$ for termination (14).

$\{ A \} c \{ P \} \equiv \forall s . \forall s' . R_{c} (s, s') \Rightarrow A' s \Rightarrow P' s' , \quad (12)$

$\{ A \} c \{ P \} \equiv \forall s . A s \Rightarrow W_{\text{a}} , P s , \quad (15)$

$\{ A \} c \{ P \} \equiv \forall s . A s \Rightarrow W_{\text{a}} , P s , \quad (16)$

To obtain explicit formulas, we calculationally transform (12) and (13) to match the shape of (15) and (16):
Matching yields (uniqueness being easy to show)

\[ \text{Wla}_c \ P s \equiv \forall s'. R_c(s, s') \Rightarrow P s', \]  \hspace{1cm} (17)

\[ \text{Wla}_c \ P \equiv \text{Wla}_c \ P \land T_c. \]  \hspace{1cm} (18)

Note the striking similarity of the calculations with those in the analysis example: everything is predicate calculus.

From (17) and (18) we can calculate properties that in the literature are always given as postulates [14]. The same holds for the results obtained by substituting the program equations for the various language constructs [11].

\[ \text{Wla}_c \ P s \equiv P s, \]  \hspace{1cm} \text{Wla}_c \ P \equiv \text{Wla}_c \ P \land T_c.

Computation sequences and program semantics

The final example shows how a model for R. Dijkstra’s Computation Calculus (CC) [15] can be expressed and properties derived using operators for sequences which we originally developed for reasoning about signals and systems.

Preliminaries: for sequences in \( A^\infty \), define take \( (j) \) by \( x \[ n \equiv x_{<n} \) and drop \( (j) \) by \( x \[ n \equiv \sigma^n x \) for any \( x : A^\infty \) and \( n : \#(x + 1) \). For predicates with common domain \( X \), define relation \( \subseteq \) by \( P \subseteq \forall x : X. P x \equiv \forall x : X. P x \).

In CC, computations are elements of \( C := S^+ \cup S^\infty \) for state space \( S \). Specifications and behaviors are expressed by computation predicates of type \( CP := C \rightarrow B \), e.g. \( \tau \) for final states. \( \gamma \) for initial states.

\( (C', C'' : CP) \) is defined for arbitrary \( C', C'' : CP^2 \) and \( \gamma : C \) by

\[ (C'; C'') \gamma \equiv (\# \gamma = \infty \land C' \gamma \lor \exists n : D \gamma. C' \gamma (n + 1) \land C'' \gamma (n) \). \]

Predicate calculus shows that composition is associative and that the predicate 1 : CP with \( \# 1 \equiv \# \gamma = 1 \) is a 2-sided unit element. We give composition precedence over \( \lor \) and \( \land \). Hence \( C \cap C' \cap C'' = C \cap (C' \cap C'') \).

States are represented by sequences of length 1, e.g.,

\[ \gamma_\infty = \tau \gamma_{\infty - 1} \] if \# \gamma \neq \infty for final states. State predicates are predicates of type \( SP := \{ P : CP \mid P \subseteq 1 \} \).

Predicate calculus shows that

(i) \( (P : T) \gamma \equiv P \gamma_0 \),

(ii) \( (T : P) \gamma \equiv \# \gamma \neq \infty \Rightarrow P \gamma_0 \),

(iii) \( (P; C = P : T \cap C \land (iv) C ; C \equiv C \cap T : P \) for any \( P : SP \) and \( C : CP \).

For the eternity predicate \( E := T ; F \) and the bounded predicate \( B := \exists E \), clearly \( E \gamma \equiv \# \gamma = \infty \) and \( B \gamma \equiv \# \gamma \neq \infty \).

CC defines, for any \( A \subset P \) in \( SP \) en \( C \) in \( CP \),

\[ \{ A \} C [P] \equiv A : C \subseteq T ; P \]  \hspace{1cm} (19)

\[ [A] C [P] \equiv A : C \subseteq B ; P \]  \hspace{1cm} (20)

A first calculation example is bringing \( \{ A \} C [P] \) into the form \( \{ A \} C \{ P \} \land T \), where \( T \) is to be discovered.

Hence \( \{ A \} C [P] \equiv \{ A \} C \{ P \} \land T \). Clearly \( A : C \subseteq B \) is the desired termination formula.

A calculation example spanning across theories is “reverse engineering” to find systems equations, i.e., abstract variants of program equations, capturing CC. So we calculate \( R_\Rightarrow : CP \rightarrow S^2 \rightarrow B \) en \( T_\Rightarrow : CP \rightarrow S \rightarrow B \) such that

\[ \{ A \} C \{ P \} \equiv \forall (s', s) : S^2. R_\Rightarrow(s, s') \Rightarrow P (\tau s') \]  \hspace{1cm} (21)

\[ A : C \subseteq B \equiv \forall s : S. A (\tau s) \Rightarrow T_\Rightarrow(s). \]  \hspace{1cm} (22)

These are adaptations of (12, 14) for SP. Calculating:

\[ A : C \subseteq B \equiv (\text{Prop. (i)}) \]  \hspace{1cm} (19)

\[ (\text{Prop. (ii)}) \]  \hspace{1cm} (17)

Apart from the evident scientific ramifications, the approach provides a unified basis for education in ECE, as advocated in [22].

Together with the many examples given, this provides ample evidence that future software tools for DSP specification and design should be based on declarative mathematical formalisms, not programming notations.
5. REFERENCES


