

The Timer Cascade: Functional Modelling and Real Time Calculi

Raymond Boute¹ and Andreas Schäfer²

¹ INTEC, Universiteit Gent, Belgium,
Raymond.Boute@intec.UGent.be

² Department für Informatik, Universität Oldenburg, Germany,
Andreas.Schaefer@informatik.uni-oldenburg.de

Abstract. Case studies can significantly contribute towards improving the understanding of formalisms and thereby to their applicability in practice. One such case, namely a cascade of the familiar 24-hour timers (in suitably generalized form) provides interesting gedanken experiments and illustrations for presenting, illustrating and comparing various formalisms for modelling real-time behaviour of systems.

The timer cascade is first modelled in a general-purpose functional formalism (Funmath) and various properties are derived, including an interesting algebraic monoid structure of timer programs. Then it is described and analyzed in duration calculus, thereby highlighting, similarities and differences in the approach to modelling and reasoning, and also the link between the formalisms.

Future work consists in using this case as a running example for exploring the same issues for other formalisms intended for real time and hybrid systems. The underlying idea is that other authors join this effort and contribute towards extending it, finally arriving at a broad comparative survey of such formalisms.

Index terms — Automata, cascade connection, Duration Calculus, functional description, Funmath, hybrid systems, real time systems, systems modelling, timers

1 Introduction: motivation and overview

Hybrid systems formalisms have become increasingly important for modelling interacting continuous and discrete aspects [2, 9, 16, 23]. Research was especially fruitful in the past two decades, but the very wealth of techniques resulting from these efforts may be a problem for integration into practice. We briefly elaborate.

A basis for comparison is the wide and problem-free integration of mathematical software such as Maple, Mathematica, Matlab and Mathcad throughout all branches of engineering. This is possible because the mathematics is classical (linear algebra, differential and integral calculus etc.) with long-standing notational and calculational conventions. Standard high school and college mathematics suffice for direct use of such software, and engineers educated 50 years ago apply it without further ado, yet quite reliably. Admittedly, use in discrete mathematics is less safe due to errors as pointed out in [19] and remedied in [7].

Appeared in: D.V. Hung and M. Wirsing (Eds.), ICTAC 2005, LNCS 3722, pp. 242-256, Springer-Verlag Berlin Heidelberg 2005

The situation alters drastically as soon as nontrivial elements from logic enter into the picture, as needed for software, digital hardware and hybrid systems. The relevant concepts are neither supported by common mathematical software, nor part of classical engineering background. Computer science students have difficulties with logic [1], and in industry, applications with logic software often requires external support by consultants (private or university researchers).

Quick introductions or trying to learn logic via tools are ill-advised. Habrias [14] aptly warns against using tools without sufficient awareness. Safe use requires a solid background in logic, including understanding as can be fostered only by serious pencil-and-paper problem solving similar to common practice in analysis and algebra. This holds for students, but even more for industrial users.

As mentioned, the wealth of formalisms is a complicating factor. Notational and calculational conventions are far less uniform than in classical mathematics; hence commonality in software support is still remote. Choosing one tool excludes possibly crucial features present in an other one. Given this situation, the (ideal) hybrid systems engineer must master several quite different formalisms, awaiting the emergence of a common framework.

Meanwhile, there is no universal solution, only ways for alleviation.

In particular, case studies provide a good starting point for understanding and comparing formalisms [16]. A widely studied example is the steam boiler [22], which has proved a useful testbed for various systems aspects. However, the crucial aspects to be highlighted are often diluted by other details.

Here we propose a case chosen to be as simple as possible and concentrating on the time aspect in its purest form, while still offering interesting ramifications: the 24 hour timer (somewhat generalized) and timer cascades. This turns out to be very appropriate for studying how time is handled in different formalisms.

An important side issue is how well formalisms “scale down” in the sense that simple systems can be described in a comparably simple way. Indeed, whereas industrial applicability often relies on scaling up (to “large” systems with many details), the intrinsic design quality and intellectual value of a formalism is often characterized by its downscaling potential in the aforesaid sense.

Overview Section 2 informally introduces the timer and the timer cascade. Section 3 provides a formal description in the functional formalism Funmath and illustrates the calculational derivation of interesting algebraic properties. In section 4, similar issues are studied using Duration Calculus (DC). The link between the two is briefly discussed in section 5, followed by an outline of future work and suggestions for contributions by others.

2 The timer cascade: informal introduction

The 24-hour timer is a “common household” device that is plugged into a wall outlet in order to supply power during predetermined time intervals (Fig. 1).

An interesting configuration arose by coincidence when storing a few of these timers, while reducing the volume by inserting them into each other (Fig. 2).

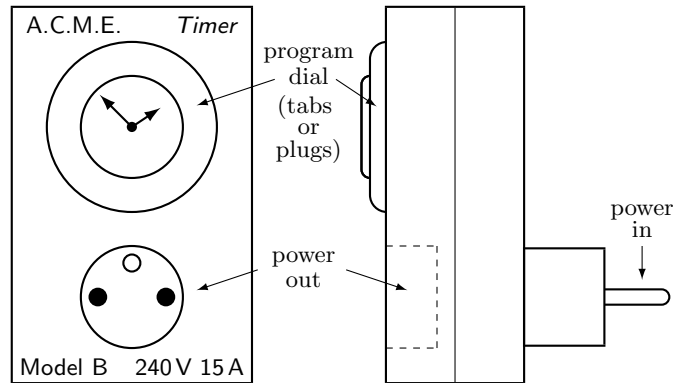


Fig. 1. A 24-hour timer

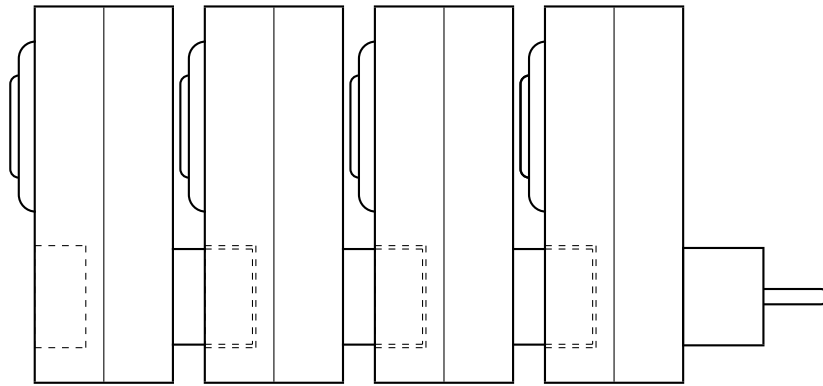


Fig. 2. A timer cascade

This immediately raises the question what would be the behavior of the resulting cascade, and what would be the best way to describe and analyze it. The idea to use this as a testbed for real time formalisms came up during a session at ICTAC 2004 in Guiyang, where several such formalisms were presented.

We make some basic assumptions explicit. Depending on the kind of timer, the “power on” intervals are programmed by pushing tabs or inserting plugs (for the analog variant with a timing motor) or via pushbuttons and a small screen (for the digital variant with electronic clock). Some digital variants support programs for longer periods (week, month) and have a battery that preserves the program during power failures. However, the battery also keeps the timer going during power out intervals, making the behavior of cascades uninteresting.

Hence our abstract model follows the analog variant: removal of power does not erase the program (which is mechanical) but pauses the timer. We also make the model generic by supporting infinitely long programs. This is done WLOG, since a finite program can be modelled by a periodic infinite one. Conversely, a

cascade of 24-hour timers can realize certain programs with longer periods, but such “practical” application is not envisaged since digital timers support longer programs in a less challenging way. Here we only want interesting behavior.

3 Functional modelling of the timer cascade

3.1 The formalism used

By *formalism* we mean a language (or notation) together with formal manipulation rules. In this section we shall use *Funmath* (*Functional mathematics*).

The language of Funmath [4] consists of only 4 constructs: *identifier*, *application*, *abstraction* and *tupling*. These suffice to synthesize common mathematical conventions while removing all defects (ambiguities, inconsistencies) and to support new and very useful styles of expression, in particular point-free ones.

The calculation rules of Funmath [5] equip all these forms of expression, including those that are rather loose in conventional mathematics, with a precise formal basis for symbolic manipulation, “making the symbols do the work”. This means that calculation is guided by the shape of the expressions [11, 13].

The two main elements are: (i) a *functional predicate calculus* [5, 7], enabling engineers to calculate with predicates and quantifiers as fluently as they have learned for derivatives and integrals; (ii) *concrete generic functionals* [5, 6], providing similar fluency with higher order functions (functionals), with the point-free style, and with smooth transition between styles.

Here we use Funmath mostly in the “conservative mode” of synthesizing conventions familiar to readers with modest mathematical background and no prior acquaintance with our formalism. The references provide further detail.

3.2 Modelling the (abstract) timer and the timer cascade

Conventions We do not model power inputs and outputs as AC waveforms, but as binary signals taking the values 0 (“off”) and 1 (“on”). Signals are themselves functions of time. We assume the time domain $\mathbb{T} := \mathbb{R}_{\geq 0}$ and value domain $\mathbb{B} := \{0, 1\}$, which is a subset of \mathbb{R} . We prefer this over $\{F, T\}$ for various reasons. Adherents of $\{F, T\}$ can adapt the sequel via a *characteristic function* $c: \{F, T\} \rightarrow \{0, 1\}$ with $c_F = 0$ and $c_T = 1$ (or simply $c := (F, T)^-$ in Funmath).

Timer model Our first signal space is the set of \mathbb{B} -valued functions (predicates)

$$\text{Sig} := \{P : \mathbb{T} \rightarrow \mathbb{B} \mid P \text{ is p.c.}\} \quad . \quad (1)$$

The usual notion of piecewise continuity over a closed interval is assumed generalized to possibly infinite intervals: a function is piecewise continuous (p.c.) over an interval iff in every finite closed subinterval it has at most a finite number of discontinuities, and left and right limits exist at each discontinuity (plus right limit at the start and left limit at the end of each of the subintervals). If the interval of interest is not stated explicitly, it is taken to be the domain of

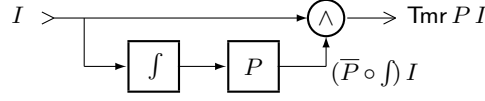


Fig. 3. Signal flow model of a timer

State space model A large class of systems [17] is modelled by a *state function* stf and an *output function* out relating state s , input i , output u by

$$D s t = \text{stf}(s t, i t) \quad \text{and} \quad u t = \text{out}(s t, i t) \quad (4)$$

where $D s$ is the derivative of s . E.g., for linear circuits these functions are of the form $\text{stf}(s t, i t) = a t \cdot s t + b t \cdot i t$ etc. or similar matrix expressions in case there are several state, input or output variables.

A timer is not linear (due to the way in which it depends on P), but fits into the generic model of (4) as follows, the state being the integrator output.

$$D s t \equiv I t \quad \text{and} \quad U t \equiv I t \wedge P(s t) \quad . \quad (5)$$

For an n -stage cascade, the state s is an n -tuple (of integrator outputs) with $\forall k : \square n . (D s_k t \equiv i_k t) \wedge (u_k t \equiv i_k t \wedge p_k(s_k t)) \wedge (n \neq 0 \Rightarrow i_k t = u_{k-1} t)$ and $I, U = i_0, u_{n-1}$. For the *block*: $\square n = \{j : \mathbb{N} \mid j < n\}$. The state space model is

$$\begin{aligned} \forall k : \square n . D s_k t &\equiv I t \wedge \forall j : \square k . p_j(s_j t) \\ U t &\equiv I t \wedge \forall j : \square n . p_j(s_j t) \quad . \end{aligned} \quad (6)$$

The calculation is based on logic only; linearity neither holds nor is assumed.

Convention Since $\mathbb{B} = \{0, 1\}$, we replace \wedge by \cdot , so $\text{Tmr } P I t = I t \cdot P(f I t)$. For (5), this yields $U t = P(s t) \cdot I t$, whereas (6) can be written

$$\forall k : \square n . D s_k t = I t \cdot \prod_{j=0}^{k-1} p_j(s_j t) \quad \text{and} \quad U t = I t \cdot \prod_{j=0}^{n-1} p_j(s_j t) \quad . \quad (7)$$

Algebraic properties: program composition and the program monoid Since the behavior of a timer is fully characterized by its program, we look for operators on programs in order to reduce reasoning to programs only. Specifically, we wish to study timer cascades via two-argument operators on programs.

A cascade of 2 timers with programs P and P' has behavior $\text{Tmr } P \circ \text{Tmr } P'$. The question is: can we calculate a program Q such that $\text{Tmr } Q = \text{Tmr } P \circ \text{Tmr } P'$ or, equivalently, an operator $\triangleright : \text{Sig}^2 \rightarrow \text{Sig}$ satisfying the following condition?

$$\text{DESIGN REQUIREMENT FOR } \triangleright : \quad \text{Tmr}(P \triangleright P') = \text{Tmr } P \circ \text{Tmr } P' \quad (8)$$

Algebraic derivation Clearly, a timer plugged into a non-interrupted outlet reflects its own program at the output. Formally, for the constant signal $\underline{1} := \mathbb{T} \bullet 1$

and any $P : \text{Sig}$ we calculate $\text{Tmr } P \underline{1} t = P(\int \underline{1} t) \cdot \underline{1} t = P t \cdot 1 = P t$ (omitting the obvious justifications), hence

$$\text{Tmr } P \underline{1} = P \quad . \quad (9)$$

So, $\text{Tmr } P = \text{Tmr } P' \Rightarrow \langle \text{Leibniz} \rangle \text{Tmr } P \underline{1} = \text{Tmr } P' \underline{1} \Rightarrow \langle \text{Tmr } P \underline{1} = P \rangle P = P'$, from which we conclude the injectivity of Tmr . Therefore the inverse Tmr^- satisfies $\text{Tmr}^- (\text{Tmr } P) = P$ for any $P : \text{Sig}$ and, by the preceding reasoning, an explicit formula for Tmr^- is $\text{Tmr}^- b = b \underline{1}$ for any behavior b in $\mathcal{R} \text{Tmr}$, the range of Tmr .

If programs P and P' satisfy $\text{Tmr } P \circ \text{Tmr } P' \in \mathcal{R} \text{Tmr}$ (hypothesis) and we impose on $\triangleright : \text{Sig}^2 \rightarrow \text{Sig}$ the design requirement $\text{Tmr} (P \triangleright P') = \text{Tmr } P \circ \text{Tmr } P'$ (for any P, P'), then

$$\begin{aligned} P \triangleright P' &= \langle \text{Tmr}^- (\text{Tmr } P) = P \rangle \text{Tmr}^- (\text{Tmr} (P \triangleright P')) \\ &= \langle \text{Dsgn. requirement} \rangle \text{Tmr}^- (\text{Tmr } P \circ \text{Tmr } P') \\ &= \langle \text{Hyp.}, \text{Tmr}^- b = b \underline{1} \rangle (\text{Tmr } P \circ \text{Tmr } P') \underline{1} \\ &= \langle \text{Definition } \circ \rangle \text{Tmr } P (\text{Tmr } P' \underline{1}) \\ &= \langle \text{Tmr } P \underline{1} = P \rangle \text{Tmr } P P' \end{aligned}$$

This yields an explicit formula for \triangleright namely $P \triangleright P' = \text{Tmr } P P'$, depending on the condition $\text{Tmr } P \circ \text{Tmr } P' \in \mathcal{R} \text{Tmr}$. Next we verify that it is always satisfied.

Analytic verification It suffices proving that \triangleright defined by $P \triangleright P' = \text{Tmr } P P'$ satisfies the design requirement (8). Before doing so, observe that, when generalizing Sig to $\text{Sig} := \{f : \mathbb{R} \rightarrow \mathbb{R} \mid f \text{ is p.c.}\}$ and \int and Tmr accordingly while maintaining the image definition $\text{Tmr } f g x = f(\int g x) \cdot g x$, everything done since replacing \wedge by \cdot remains valid, because the proofs nowhere relied on any restriction to \mathbb{B} .

THEOREM $\text{Tmr} (\text{Tmr } f g) = \text{Tmr } f \circ \text{Tmr } g$ for any signals f, g .

PROOF The successive domains are clearly equal. Also, for $h : \text{Sig}$ and $x : \mathbb{T}$,

$$\begin{aligned} \text{Tmr} (\text{Tmr } f g) h x &= \langle \text{Def. Tmr} \rangle \text{Tmr } f g (\int h x) \cdot h x \\ &= \langle \text{Def. Tmr} \rangle f (\int g (\int h x)) \cdot g (\int h x) \cdot h x \\ &= \langle \text{Def. Tmr} \rangle f (\int g (\int h x)) \cdot \text{Tmr } g h x \\ &= \langle \text{Lemma} \rangle f (\int (\text{Tmr } g h) x) \cdot \text{Tmr } g h x \\ &= \langle \text{Def. Tmr} \rangle \text{Tmr } f (\text{Tmr } g h) x \\ &= \langle \text{Def. } \circ \rangle (\text{Tmr } f \circ \text{Tmr } g) h x \end{aligned}$$

Thus far, the lemma justifying $\int g (\int h x) = \int (\text{Tmr } g h) x$ is “wishful thinking”, guided by the shape of $\text{Tmr } g h x$ to enable the next step. Now we prove it.

LEMMA $\int f \circ \int g = \int (\text{Tmr } f g)$ for p.c. f and g .

PROOF We shall invoke some properties for the derivative D , namely

- (i) Fundamental theorem of calculus: $D(f f) = f \mid \mathcal{D}(D(f f))$ for p.c. f .
 - (ii) Leibniz’s rule: $D(f \circ g) = D f (g x) \cdot D g x$ provided the derivatives are p.c..
 - (iii) Delegation of equality to derivative: $f = g \equiv f 0 = g 0 \wedge D f = D g$ (idem).
- In applying (iii), $(\int f \circ \int g) 0 = \int (\text{Tmr } f g) 0$ is trivial since $\int f 0 = 0$ for p.c. f .

For the derivatives, the domains exclude undefined points and discontinuities do not affect the integral [3, p. 311]. The images for x in the domain obey

$$\begin{aligned}
D(f \circ g) x &= \langle \text{Leibniz's rule} \rangle D(f) (fg) \cdot D(g) x \\
&= \langle \text{Fundam. thm.} \rangle f(g) x \cdot g x \\
&= \langle \text{Definition Tmr} \rangle \text{Tmr} f g x \\
&= \langle \text{Fundam. thm.} \rangle D(f(\text{Tmr} f g)) x
\end{aligned}$$

We call $\int(\text{Tmr} f g)$ the *timer integral* for obvious reasons.

Algebraic properties of program composition Having fulfilled all proof obligations, we can now assert that \triangleright defined (for f and g in the generalized Sig) by

$$\text{DEFINITION OF } \triangleright : f \triangleright g = \text{Tmr} f g \quad (10)$$

satisfies

$$\text{HOMOMORPHISM: } \text{Tmr}(f \triangleright g) = \text{Tmr} f \circ \text{Tmr} g \quad . \quad (11)$$

Recall also that Tmr is injective. We now derive some properties.

(a) The operator \triangleright is associative. Indeed,

$$\begin{aligned}
f \triangleright (g \triangleright h) &= \langle \text{Defin. } \triangleright \rangle \text{Tmr} f (\text{Tmr} g h) \\
&= \langle \text{Defin. } \circ \rangle (\text{Tmr} f \circ \text{Tmr} g) h \\
&= \langle \text{Prop. (11)} \rangle \text{Tmr}(f \triangleright g) h \\
&= \langle \text{Defin. } \triangleright \rangle (f \triangleright g) \triangleright h
\end{aligned}$$

(b) The operator \triangleright has $\underline{1} := \mathbb{R} \bullet 1$ as left and right identity. Indeed,

$$\begin{aligned}
(\underline{1} \triangleright f) x &= \langle \text{Defin. } \triangleright \rangle \text{Tmr} \underline{1} f x \\
&= \langle \text{Def. Tmr} \rangle \underline{1} (\int f t) \cdot f x \\
&= \langle \text{Defin. } \underline{1} \rangle f x \\
(f \triangleright \underline{1}) x &= \langle \text{Def. Tmr} \rangle f (\int \underline{1} x) \cdot \underline{1} x \\
&= \langle \text{Defin. } \underline{1} \rangle f (\int \underline{1} x) \\
&= \langle \int \underline{1} x = x \rangle f x
\end{aligned}$$

This makes Sig a monoid under \triangleright and Tmr an injective monoid homomorphism.

3.4 Conclusions

It is clear that Tmr and \triangleright have many algebraic properties, about which only the tip of the iceberg has been explored.

One of the issues deserving further investigation is the periodicity of periodic programs and (as a gedanken experiment) program synthesis by cascades of periodic programs (which model the behavior of finite programs).

4 Modelling using Duration Calculus

Duration Calculus (DC) [10, 15] is an interval temporal logic. It incorporates the integral operator and is thus able to reason about durations of system states. This is a particular convenient feature for modelling and reasoning about the timer cascade as each timer is in fact a stop-watch. As in the previous section we investigate how a cascade of two timers can be expressed using only one timer. Although DC is equipped with a powerful proof system, we put emphasis on automatic verification using model-checking techniques. Modelling and the automatic verification are performed on a more concrete level than in the functional modelling with Funmath.

4.1 Duration Calculus

The behavior of systems is described by time-dependent variables, so called observables which have in most cases finite domains. For each timer in the cascade we use two observables `power_in` and `power_out`. The observable `power_in` models that the timer is connected to current and `power_out` models that it supplies current at its output. As we use only boolean observables in this example the semantics of an observable is a function of type `Sig` thus $\mathcal{I}(X) : \mathbb{T} \rightarrow \mathbb{B}$. For the integrals to exist, we further require the functions to be piecewise constant.

Boolean combinations of observables, so called state assertions are used to specify the state of the system for a certain point in time.

Duration Calculus is interpreted over time intervals. Therefore DC terms associate a real number to each interval. An integral operator can be applied to state assertions in order to measure its duration. Furthermore DC provides global rigid variables and the special symbol ℓ , denoting the length of the interval.

Formally, the set of DC terms is defined by the following EBNF

$$\theta ::= x \mid f(\theta_1, \dots, \theta_n) \mid \int P \mid \ell$$

where x denotes a global time-independent variable, f an n -ary function symbol and P a state assertion. As usual, the value of a rigid variable is determined by a valuation \mathcal{V} . In addition to first order quantifiers and boolean connectives, Duration Calculus uses a special modality \frown called “chop”. A formula $F \frown G$ is true on an interval, iff this interval can be partitioned into two subintervals, such that F holds on the first part and G holds on the second part. Formally, DC formulas are generated from the following EBNF

$$F ::= \neg F \mid F_1 \wedge F_2 \mid F_1 \frown F_2 \mid p(\theta_1, \dots, \theta_n) \mid \forall x.F$$

As usual, the other logical connectives can be derived as abbreviations. Additionally, we introduce the following abbreviations, to denote the point interval,

$$\square \stackrel{df}{=} \ell = 0$$

To denote that the state assertion P is true almost everywhere on a non-point interval, we use

$$[P] \stackrel{df}{=} \int P = \ell \wedge \ell > 0$$

The modalities \diamond^{DC} , \square^{DC} and \square_0^{DC} are derived by

$$\diamond^{DC} F \stackrel{df}{=} true \frown F \frown true \quad \square^{DC} F \stackrel{df}{=} \neg \diamond^{DC} \neg F \quad \square_0^{DC} F \stackrel{df}{=} \neg (\neg F \frown true)$$

The modality \diamond^{DC} reads as “on some subinterval”, \square^{DC} as “on every subinterval” and \square_0^{DC} as “on every subinterval starting at point zero”.

4.2 Modelling the timer cascade

As mentioned in the introduction, we employ two boolean observables `power_in` and `power_out` to model the state of one timer in the cascade. Additionally, we use the auxiliary observable `pass` to denote whether current can pass through the timer or not. We use the index i to indicate the i -th timer. For each timer we use three parameters,

- $cycle_i$, the cycle time of the i -th timer,
- $start_i$ the start time of the i -th timer,
- $stop_i$ the stop time of the i -th timer.

We specify the behavior of a timer cascade using the following DC formulas.

If the duration of `power_ini` is below the start value, `pass` has to be false, i.e.

$$\square_0^{DC} ((\int \text{power_in}_i \text{ mod } cycle_i < start_i) \Rightarrow true \frown [\neg \text{pass}_i])$$

If the value is between start and stop, `pass` is true.

$$\square_0^{DC} ((start_i \leq \int \text{power_in}_i \text{ mod } cycle_i \leq stop_i) \Rightarrow true \frown [\text{pass}_i])$$

Above the stop value, the observable `pass` has to be false again.

$$\square_0^{DC} ((\int \text{power_in}_i \text{ mod } cycle_i > stop_i) \Rightarrow true \frown [\neg \text{pass}_i])$$

If power can pass through the timer and it is connected to current, the outlet is powered.

$$[] \vee [(\text{power_in}_i \wedge \text{pass}) \Leftrightarrow \text{power_out}_i]$$

The observables `power_out` and `power_in` of two consecutive timers are connected.

$$[] \vee [\text{power_in}_{i+1} \Leftrightarrow \text{power_out}_i]$$

As the first timer in the cascade should always be connected to the power supply, we assume

$$[] \vee [\text{power_in}_0]$$

The behavior of the complete cascade is specified by the DC formula TC which is defined to be the conjunction of all the formulas given above.

4.3 Refinement

Duration Calculus can be used to describe systems at several levels of detail in different phases of the design process. Especially, it can be used to establish a refinement relationship between a more abstract specification and a more concrete implementation level.

In this section we investigate how a single abstract timer of cycle time $cycle^A$ with program start $start^A$ and stop $stop^A$ where $start^A \leq stop^A$ can be implemented by two concrete timers having the same shorter cycle-time $cycle^C$.

To derive an implementation, we introduce the following abbreviation

$$\Delta^A \stackrel{df}{=} stop^A - start^A$$

denoting the length of the program. At first, we compute how many cycles of the concrete the cascade has to wait until the program should start. We denote by div and rem the result of the division and the remainder respectively of the start time $start^A$ by the cycle time $cycle^C$ of the implementation, i.e.

$$start^A = div \cdot cycle^C + rem$$

such that $0 \leq rem < cycle^C$. We can now implement the abstract timer by a cascade of two concrete timers using the program

$$\begin{aligned} start_0^C &\stackrel{df}{=} rem \\ stop_0^C &\stackrel{df}{=} rem + \frac{cycle^C}{m} \\ start_1^C &\stackrel{df}{=} div \cdot \frac{cycle^C}{m} \\ stop_1^C &\stackrel{df}{=} div \cdot \frac{cycle^C}{m} + \Delta^A \end{aligned}$$

for $m = \frac{cycle^A}{cycle^C} \in \mathbb{N}$ with the additional constraint that $\frac{cycle^C}{m} > \Delta^A$. The program of timer 0 must have a duration of $\frac{cycle^C}{m}$ to ensure that both timers are in zero position after the first timer has completed m cycles. During the first div cycles of timer 0, $power_out_1^C$ is not activated. Only after $div \cdot cycle^C + rem$ time units $power_out_1$ becomes true for Δ^A time units. This is ensured by the extra condition $\frac{cycle^C}{m} > \Delta^A$. It is not always possible to find an implementation of the abstract timer, by two concrete ones. For example if the duration Δ^A of the abstract timer is greater than the cycle time of the concrete timers, it is impossible to find an implementation. The definition above does not yield a valid program in these cases, as the value of $stop_1^C$ exceeds the cycle time. Nevertheless, if all time bounds are below the cycle time, we get a correct implementation of the abstract specification. This is to be verified formally.

Let TC^A denote the specification of the abstract timer and TC^C be the specification of the concrete implementation, then we have to show the refinement requirement e.g.

$$TC^C \wedge TC^A \Rightarrow [power_out_1^C \Leftrightarrow power_out^A].$$

4.4 Verification

Duration Calculus is equipped with numerous proof rules to facilitate this kind of proofs. As we have shown such a calculation by hand in the previous section, we concentrate on the application of tools here. DC is decidable for discrete time domain – and undecidable for continuous time domain. A model-checker called DCValid [18] is available, so we employ this tool for the verification of the refinement requirement. As DCValid does not allow arbitrary computation, we verify the refinement of one 24-hour timer by a cascade of two 12h timers. Henceforth, we assume our two systems are defined by the following parameters.

$$\begin{aligned} start^A &= 15, & stop^A &= 17, & cycle^A &= 24, \\ start_0^C &= 3, & stop_0^C &= 9, & & \\ start_1^C &= 6, & stop_1^C &= 8, & cycle^C &= 12. \end{aligned}$$

As DCValid does not incorporate calculation of remainders, this calculation has to be eliminated. To this end, we introduce 3 fresh observables $zero^A, zero_1^C$, and $zero_2^C$ to mark all points on which the respective timer is in zero position. We specify that $zero$ has to be true for one time unit after the timer having $power_in$ activated for its cycle time. To this end, we define lower and upper bound for $zero$ by the following DC formulas.

$$\begin{aligned} &\neg \diamond^{DC} ((([zero] \wedge [\neg zero]) \wedge (\int power_in < cycle)) \wedge [zero]) \\ &\neg ((([\neg power_in] \vee []) \wedge [power_in \wedge \neg zero]) \wedge true) \\ &\neg \diamond^{DC} ([zero] \wedge \ell > 1) \\ &\neg \diamond^{DC} (([zero] \wedge [\neg zero]) \wedge \int power_in > cycle) \end{aligned}$$

As we use *discrete* DC for automatic verification, we need not to specify a lower bound on the duration of $zero$ as a phase $[zero]$ cannot have a duration below one time unit. Having introduced these auxiliary observables, we can modify the specification. Instead of looking at all intervals starting at the beginning and calculating the measure of $power_in$ modulo the cycle time, we can just measure the amount of time $power_in$ is true since the last phase on which $zero$ holds.

Every interval starting with a phase on which $zero$ is true and the measure of $power_in$ is below the start of the timer, on the end of the interval $pass$ does not hold.

$$\begin{aligned} &\Box^{DC} ((([zero] \wedge \ell = 1 \wedge ([\neg zero] \vee [])) \\ &\quad \wedge (\int power_in \leq start)) \\ &\quad \Rightarrow (true \wedge [\neg pass])) \end{aligned}$$

If the measure is between $start$ and $stop$ the interval must end in a phase satisfying $pass$.

$$\begin{aligned} &\Box^{DC} ((([zero] \wedge \ell = 1 \wedge ([\neg zero] \vee [])) \\ &\quad \wedge (\int power_in > start \wedge \int power_in \leq stop)) \\ &\quad \Rightarrow (true \wedge [pass])) \end{aligned}$$

If the measure is above *stop* it has to end in $\neg\text{pass}$.

$$\begin{aligned} \Box^{DC} &(((\text{zero}] \wedge \ell = 1 \wedge (\neg\text{zero}] \vee [])) \wedge (\int \text{power.in} > \text{stop})) \\ &\Rightarrow (\text{true} \wedge [\neg\text{pass}]) \end{aligned}$$

Using this definition, the refinement requirement can be automatically verified. DCValid takes 4.16 seconds on a 1.8 GHz Athlon XP 2200+ machine to verify the validity. We employed manual optimisations exploiting the fact, that power.in^A and power.in_2^C are always true and therefore instead of calculating the measure $\int \text{power.in}^A$ and $\int \text{power.in}_2^C$ respectively, one can use the length of the interval directly.

4.5 Conclusion

We presented how a specification of a timer cascade can be formalised in Duration Calculus. As DC incorporates the \int -operator, it allows natural modelling of stop watches and henceforth the whole timer cascade. Duration Calculus can be used in various stages of the design process. So we presented how an abstract timer cascade can be refined and how the correctness of the refinement can be automatically verified.

5 Final remarks and future work

5.1 Linking formalisms

Linking formalisms in a clear, formal way always contributes to a better understanding of all formalisms involved.

A promising approach to linking Duration Calculus as used in section 4 with the functional approach as used in section 3 is similar to the one used for linking R. Dijkstra's *Computation Calculus* [12] to Calculational Semantics in [8].

Within the scope of this paper, only an outline can be given. Define the set of intervals over a totally ordered time domain \mathbb{T} by $\mathcal{I} := \{[a, b] \mid a, b: (\mathbb{T}^2)_{\leq}\}$. Various styles of DC can be defined in Funmath. Here are two of them.

- Interval style: predicates of type $\text{IP} := \mathcal{I} \rightarrow \mathbb{B}$ (predicates on intervals)
- Computation style: predicates of type $\text{CP} := \mathcal{C} \rightarrow \mathbb{B}$ where the set of *computations* is defined by $\mathcal{C} := \bigcup I: \mathcal{I}. I \rightarrow \mathbb{S}$, given a suitable state space \mathbb{S} .

In this outline, we concentrate on “chop” (\frown), the pivotal operator in DC.

- Interval style: \frown has type $\text{IP}^2 \rightarrow \text{IP}$, map $(P \frown Q) I \equiv \exists t: I. P I_{\leq t} \wedge Q I_{\geq t}$
- Computation style: type $\text{CP}^2 \rightarrow \text{CP}$, map $(P \frown Q) \gamma \equiv \exists t: \mathcal{D} \gamma. P \gamma_{\leq t} \wedge Q \gamma_{\geq t}$

Note: *filtering* (\downarrow) is defined for any set S by $S \downarrow P = \{x: S \cap \mathcal{D} P \mid P x\}$ and for any function f by $\mathcal{D} f_P = \{x: \mathcal{D} f \cap \mathcal{D} P \mid P x\}$ with $\forall x: \mathcal{D} f_P. f_P x = f x$; in both cases P is any predicate. Abbreviating $a \downarrow b$ as a_b (and $a \uparrow b$ as a^b), together with so-called *partial application* (as in $\leq t$) explains the notation formally.

Crucial remark Parameters like I and γ appear only in basic definitions and calculations where axioms of the axiomatic formulations of DC are derived as theorems. In subsequent use, the formulas can be written in exactly the same form as in the axiomatic formulations, and calculations are “point-free”. The difference between interval style and computation style then becomes hidden.

For instance, associativity of “chop”, namely $(P \frown Q) \frown R = P \frown (Q \frown R)$, is easily proven from either definition using functional predicate calculus.

Another example: let \diamond be defined in the interval style by³ $\diamond P I \equiv \exists P_{\subseteq I}$. Defining $\mathbf{T} := \mathcal{I} \bullet 1$ (“1 for any interval”), one proves similarly $\diamond P = \mathbf{T} \frown P \frown \mathbf{T}$.

While this is only an outline, it captures the flavor of the approach.

5.2 Future work

Obviously, the most immediate task is the complete elaboration of the link between the functional and the DC models of the timer.

However, this paper is only a first step in a more ambitious effort towards a broad comparative survey of formalisms for real time and hybrid systems. To this effect, we shall study several other formalisms in a similar way, elaborating for each two examples: one that highlights its strong points (dependent on the formalism), and the timer cascade (the same running example for all). Most importantly, links and the possibility of a common framework will be investigated. Another issue is the interaction between tools supporting various formalisms.

We hope that other researchers join this effort, most conveniently by providing a brief outline of their preferred formalism and two examples as described. For those who are interested, we will prepare a more extensive discussion of the kind of specifications and verification obligations that would be most helpful.

References

1. Vicki L. Almstrum, “Investigating Student Difficulties With Mathematical Logic”, in: C. Neville Dean, Michael G. Hinchey, eds., *Teaching and Learning Formal Methods*, pp. 131–160. Academic Press (1996)
2. Rajeev Alur, Thomas A. Henzinger, Eduardo D. Sontag, eds., *Hybrid Systems III*, LNCS 1066. Springer-Verlag, Berlin Heidelberg (1996)
3. Robert G. Bartle, *The Elements of Real Analysis*. Wiley, New York (1964)
4. Raymond T. Boute, *Funmath illustrated: A Declarative Formalism and Application Examples*. Declarative Systems Series No. 1, Computing Science Institute, University of Nijmegen (1993)
5. Raymond Boute, *Functional Mathematics: a Unifying Declarative and Computational Approach to Systems, Circuits and Programs — Part I: Basic Mathematics*. Course text, Ghent University (2002)
6. Raymond T. Boute, “Concrete Generic Functionals: Principles, Design and Applications”, in: Jeremy Gibbons, Johan Jeuring, eds., *Generic Programming*, pp. 89–119, Kluwer (2003)

³ In the functional predicate calculus, quantifiers are predicates over predicates, viz., $\forall P \equiv P = \mathcal{D} P \bullet 1$ and $\exists P \equiv P \neq \mathcal{D} P \bullet 0$. We write $S \bullet e$ for the constant function mapping all elements of set S to e . Note: $\exists P_{\subseteq I} \equiv \exists J : \mathcal{I}. J \subseteq I \wedge P J$ for P in \mathbb{P} .

7. Raymond Boute, “Functional declarative language design and predicate calculus: a practical approach”, to appear in *ACM Trans. Prog. Lang. and Syst.*
8. Raymond Boute, “Calculational semantics: deriving programming theories from equations by functional predicate calculus”, to appear in *ACM Trans. Prog. Lang. and Syst.*
9. J. Buck, S. Ha, E. A. Lee, D. G. Messerschmitt, “Ptolemy: a framework for simulating and prototyping heterogeneous systems”, *International Journal of Computer Simulation*, spec. issue on Simulation Software Development (Jan. 1994)
10. Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *IPL*, 40(5):269–276, 1991.
11. Edsger W. Dijkstra, “Under the spell of Leibniz’s dream”, *EWD1298* (April 2000).
12. Rutger M. Dijkstra, “Computation calculus: Bridging a formalization gap”, in: Johan Jeuring, ed., *Mathematics of Program Construction*, pp. 151–174. LNCS 1422, Springer (1998)
13. David Gries, “The need for education in useful formal logic”, *IEEE Computer* 29, 4, pp. 29–30 (April 1996)
14. Henri Habrias and Sébastien Faucou, “Linking Paradigms, Semi-formal and Formal Notations”, in: C. Neville Dean and Raymond T. Boute, eds., *Teaching Formal Methods*, pp. 166–184, Springer LNCS 3294 (Nov. 2004)
15. M. R. Hansen and Zhou Chaochen. *Duration Calculus: A Formal Approach to Real-Time Systems*. EATCS: Monographs in Theoretical Computer Science. Springer, 2004.
16. Thomas Krilavičius, “Bestiarium of Hybrid Systems” (draft, Mar. 2005) <http://wwwhome.cs.utwente.nl/~krilaviciust/publications/bestiarium.pdf>
17. Edward A. Lee and Pravin Varaiya. *Structure and Interpretation of Signals and Systems*. Addison-Wesley (2003)
18. P.K. Pandya, Specifying and deciding quantified discrete-time duration calculus formulae using dvalid. Technical report, Tata Institute of Fundamental Research, 2000.
19. William Pugh, “Counting Solutions to Presburger Formulas: How and Why”, *ACM SIGPLAN Notices* 29, 6, pp. 121–122 (June 1994)
20. Andreas Schäfer, “Combining Real-Time Model-Checking and Fault Tree Analysis”, in: D. Mandrioli and K. Araki and S. Gnesi, eds., *FM 2003: 12th International FME Symposium*. LNCS 2805, Springer (2003) <http://csd.Informatik.Uni-Oldenburg.DE/pub/Papers/as03.pdf>
21. Andreas Schäfer, “A Calculus for Shapes in Time and Space”, in: Z. Liu and K. Araki, eds., *Proc. ICTAC 2004*. LNCS 3407, Springer (2005)
22. Graeme Smith, “Specifying Mode Requirements of Embedded Systems”, in: Michael Oudshoorn, ed., *ACSC2002*, pp. 251–257 (Jan.–Feb. 2002). Also: <http://www.itee.uq.edu.au/~smith/papers/acsc2002.pdf>
23. Frits W. Vaandrager, Jan H. van Schuppen, eds., *Hybrid Systems: Computation and Control*, LNCS 1569. Springer-Verlag, Berlin Heidelberg (1999)