Thoughts about the implementation of the Duration Calculus with Coq

Samuel Colin[1,2] Samuel.Colin@inrets.fr,
Vincent Poirriez[2] Vincent.Poirriez@univ-valenciennes.fr,
Georges Mariano[1] Georges.Mariano@inrets.fr

[1] INRETS*, 20, rue Elisée RECLUS, BP 317 F-59666 Villeneuve d'Ascq Cedex, France
[2] LAMIH**, Le Mont Houy, 59313 Valenciennes Cedex 9, France

Abstract. This work is a derivative of studies about the duration calculus , aiming at deciding whether it is sound to use it as an extension logic for a formal method (namely, the "B method"). Indeed, we wanted to know the feasability and the usability, of such a modal logic implemented in a proof assistant. In this paper, two complementary implementations are described, as well as problems inherited from both sides : the proof system for itself, and the tweaking of the proof assistant.

1 Introduction

We will present the reasons that drove us to the writing of Coq libraries for DC (duration calculus), and to that end we'll do a quick presentation of the B method.

The B method, a formal method, allows the development of safe software, from abstract, mathematical specifications, to computer code that is proved correct with regard to those specifications. The steps going from specifications to code are called refinements. The abstract specifications and the refinements have to be proved correct, through the proof of so-called proof obligations, that are formulas expressed with predicate calculus and set theory, generated from the specifications and the refinements.

While this method has convinced the industrial world, it still has limits, e.g. when dealing with problems having temporal constraints. Some examples of application of the B method to time-constrained problems exist (see for example [1, 2]), but the complexity of the generated proof obligations can easily become confusing for both the automatic theorem prover and the operator who must read the formulas having failed with this prover.

Methods involving the extension of the B method also exist ([3]), and we have chosen to study the extension of the logic used by B to Duration Calculus. To do so, we needed a proof tool able to handle both normal B logical formulas, and DC formulas. Coq having several set theory libraries at disposal, we chose it to build a library for DC.

In the section 2 we'll present the duration calculus, then in section 3 the Coq proof assistant. In section 4 we will highlight interesting points about the implementation of DC with Coq, and we'll conclude in sections 5 and 6.

* Institut National de REcherche sur les Transports et leur Sécurité
** Laboratoire d'Automatique, de Mécanique, et d'Informatique industrielles et Humaines

2 Duration Calculus

This section won't present an in-depth description of the Duration Calculus, we will rather focus on peculiar properties, which will be of interest in the other sections.

2.1 History

The Duration Calculus was first presented in [4], as a temporal logic based on IL (Interval Logic) [5]. Ever since, numerous extensions were proposed for DC ([6, 7]), allowing to express more and more complex properties of real-time systems. An in-depth survey of DC and its properties can be found in [8].

2.2 Syntax

Let X_i be a *propositional temporal letter* (interpreted as a boolean function over time intervals), P_i a state variable (interpreted as a boolean-valued function over time), $x, y, \ldots$ global variables (interpreted as real numbers), f_i functions and R_i relation symbols. Usually the functions are the standard arithmetic ones $(+, *)$ and the relations also are the usual ones $(=, \leq)$. The syntax of DC formulas is (functions and relations might be noted with prefix or infix notation, as syntax is not our main concern) :

$$
\begin{aligned}
\text{formula} &::= \text{Atom} \mid \neg\ \text{formula} \mid \text{formula} \vee \text{formula} \mid \text{formula}^\frown\text{formula} \mid \exists x.\text{formula} \\
\text{Atom} &::= \mathbf{true} \mid X \mid \text{R(term,} \ldots \text{, term)} \\
\text{term} &::= x \mid \ell \mid \textstyle\int \text{state} \mid \text{f(term,} \ldots \text{, term)} \\
\text{state} &::= 0 \mid 1 \mid P \mid \text{state} \vee \text{state} \mid \neg\ \text{state}
\end{aligned}
$$

The additions of IL to predicate calculus are the special variable ℓ and the *chop* connector $^\frown$. This connector chops a formula into two formulas representing the valid predicates on the first part of the time interval and the second part, respectively. The ℓ variable represents the length of the current time interval, i.e. the value of ℓ is influenced by the *chop* connector.

The additions of DC to IL are represented by the duration operator $\int$ and the state expressions. These have the expressive power of propositional logic, and the duration operator allows to express properties on these states and on logical relations between them.

2.3 Semantics

The most direct way to interpret DC formulas is to do so over time intervals. Let the following interpretations functions and definitions domains be:

- Time, usually represented by the real numbers $\mathbb{R}$
- TimeInterval = $\{(b, e) | b, e \in \text{Time} \wedge b \leq e\}$
- Val = term $\rightarrow$ TimeInterval $\rightarrow \mathbb{R}$
- ValState = state $\rightarrow$ Time $\rightarrow \{0, 1\}$
- $\mathcal{T}$: ValState

- $\mathcal{V}$: Val
- I : formula $\rightarrow ((\text{Val} \times \text{ValState}) \times \text{TimeInterval}) \rightarrow \{true, false\}$

For readability reasons, in the description of I, $\mathcal{V}$ and $\mathcal{T}$ are implied. The same remark applies for the description of $\mathcal{V}$ and $\mathcal{T}$.

$$
\begin{array}{ll}
I(X)([b,e]) & = X_I([b,e]) \\
I(R(\theta_1,\ldots,\theta_n))([b,e]) & = R(c_1,\ldots,c_n) \text{ where } c_i = \mathcal{V}(\theta_i)([b,e]) \\
I(\neg\phi)([b,e]) & = \neg I(\phi)([b,e]) \\
I(\phi_1 \vee \phi_2)([b,e]) & = I(\phi_1)([b,e]) \vee I(\phi_2)([b,e]) \\
I(\exists x.\phi)([b,e]) & = I(\phi_{\mathcal{V}'})([b,e]) \text{ where } \mathcal{V}(y) = \mathcal{V}'(y) \text{ for } y \neq x \\
I(\phi_1 \frown \phi_2)([b,e]) & = \exists m.(I(\phi_1)([b,m]) \wedge I(\phi_2)([m,e])) \text{ for } m \in [b,e] \\
I(\mathbf{true})([b,e]) & = \text{true}
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{V}(x)([b,e]) & = x \\
\mathcal{V}(\ell)([b,e]) & = e - b \\
\mathcal{V}(f(\theta_1,\ldots,\theta_n))([b,e]) & = f(c_1,\ldots,c_n) \text{ where } c_i = \mathcal{V}(\theta_i)([b,e]) \\
\mathcal{V}(\int S)([b,e]) & = \int_b^e \mathcal{T}(S)(t)dt
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{T}(0)(t) & = 0 \\
\mathcal{T}(1)(t) & = 1 \\
\mathcal{T}(S \vee T)(t) & = \begin{cases} 0 \text{ if } \mathcal{T}(S)(t) = 0 \text{ and } \mathcal{T}(S)(t) = 0 \\ 1 \text{ otherwise} \end{cases} \\
\mathcal{T}(\neg S)(t) & = 1 - \mathcal{T}(S)(t) \\
\mathcal{T}(P)(t) & = P_{\mathcal{T}}(t)
\end{array}
$$

A proviso is added for the state variables, which are interpreted as functions over time : for the functions to be integrable, they need to be *finitely variable* over the considered time interval. For example, the function :

$$P_{\mathcal{T}}(t) = \begin{cases} 0 \text{ if } t \text{ is irrational} \\ 1 \text{ otherwise} \end{cases}$$

2.4 Examples

Some examples are inspired from [8]:

1. Let the state variables Gas and $Flame$ be the expressions of the event "gas is produced" and "flame exists", respectively. Then this DC formula states that during the non-zero time interval, each time gas is produced, the flame must be present :

$$\int(Gas \Rightarrow Flame) = \ell \wedge \ell > 0$$

2. The formula $\ell = 10 \frown \ell = 5$ states that in the first part of the time interval is 10 time units long, and the second part 5 time units long.

3. **true**$^\frown(\phi^\frown$**true**$)$ states that the ϕ formula is valid in some sub-interval. This special construction is also noted $\Diamond\phi$, and is comparable with the $\Diamond$ one can find in other temporal logics.
4. Similarly, the formula $\neg\Diamond(\neg\phi)$ is noted $\Box\phi$, and is interpreted as : "for any time sub-interval, the ϕ formula is valid".

Now an example of the semantics of DC, over a given time interval $[b,e]$, with $e > b$:

Example 1. Let's suppose that *Gas* is a state whose value is 1 all over the interval, and $Flame$ a state whose value is 0 in the first half of the [e,b] time interval, 1 in the second. Intuitively, it means that the gas is leaking, before it is set on fire :

$$
\begin{aligned}
&I(\textstyle\int(Gas \Rightarrow Flame) = \ell \wedge \ell > 0)([b,e]) \\
&\quad\equiv\; I(\textstyle\int(Gas \Rightarrow Flame) = \ell)([b,e]) \wedge I(\ell > 0)([b,e]) \\
&\quad\equiv\; \mathcal{V}(\textstyle\int(Gas \Rightarrow Flame))([b,e]) = \mathcal{V}(\ell)([b,e]) \wedge \mathcal{V}(\ell)([b,e]) > \mathcal{V}(0)([b,e]) \\
&\quad\equiv\; \textstyle\int_b^e \mathcal{T}((Gas \Rightarrow Flame)(t)dt)([b,e]) = e-b \wedge e-b > 0 \\
&\quad\equiv\; \textstyle\int_b^e \mathcal{T}((\neg Gas \vee Flame)(t)dt)([b,e]) = e-b \wedge \text{true} \\
&\quad\equiv\; \tfrac{e-b}{2} = e-b
\end{aligned}
$$

Because $\neg Gas$ and $Flame$ are both 0 in the first half of the interval. The obtained formula is false, as $e > b$. So the given states *Gas* and $Flame$ don't fulfil the requirement.

2.5 Proof system

We will only underline in this section some hard points of the proof system of [8], on which we have based the implementation described in section 4. Now for some definitions beforehand:

Definition 2. A DC formula is called *rigid* if it doesn't contain any state variable, propositional letter or ℓ symbol. It is otherwise called *flexible*

Definition 3. A DC formula is called *chop free* if the $^\frown$ doesn't occur in the formula

Definition 4. The term θ is *free for x in* ϕ if x doesn't occur freely in ϕ within the scope of the quantified variable y, y occurring in θ

This last definition is used later in a side-condition to address the problem of variable instanciation.

The axioms of the proof system are distributed between the ones coming from IL, and the ones coming from DC. For example:

Example 5. Some IL axioms:

$\ell \geq 0$	The length of a time interval can't be negative
$\phi^\frown\psi \Rightarrow \phi$ if ϕ is rigid	If a rigid formula is valid on a part of an interval, it is also valid on the whole interval, as it is not influenced by temporal variables or symbols

Example 6. Some DC axioms:

$\int 1 = \ell$	The "always true" state lasts the whole time interval
$\int S_1 = \int S_2$ if $S_1 \Leftrightarrow S_2$ holds in propositional logic	Equivalent states have the same duration

Some inference rules are added, and the ones inherited from predicate calculus are modified.

Two noticeable things about the proof system is that:

1. Side-conditions might require non-trivial analysis of the involved formulas
2. Inference rules doesn't hold hypotheses, as in sequent calculus, for example. Thus some of them won't be valid if coded "as is" in a prover (these problems have already been solved in [9], see section 4 for more information).

For example:

Example 7. Some DC inference rules (inherited from IL inference rules):

$$\frac{\forall x.\phi(x)}{\phi(\theta)} \text{ if } \theta \text{ is free for } x \text{ in } \phi(x) \text{ and } \begin{cases} \text{either } \theta \text{ is rigid} \\ \text{or } \phi(x) \text{ is chop free} \end{cases}$$

$$\frac{\phi \Rightarrow \psi}{(\phi ^\frown \Phi) \Rightarrow (\psi ^\frown \Phi)}$$

3 The Coq proof assistant

3.1 Presentation

Paraphrasing the Coq reference manual (see [10]), "Coq is a proof assistant for higher-order logic, allowing the development of computer programs consistent with their formal specification".

Coq's logical language is based on the *Calculus of Inductive Constructions*, a variety of type theory, which allows manipulations of higher order terms in a consistent framework, ensured by type-checking of formulas. Still citing the Coq reference manual, "It is possible to understand the Calculus of Inductive Constructions at a higher level, as a mixture of predicate calculus, inductive predicate definitions presented as typed PROLOG, and recursive function definitions close to the language ML".

One of the noticeable properties of Coq (even if not useful for the comprehension of next section) is its ability to extract programs from proofs, endorsing hereby the Curry-Howard isomorphism.

3.2 Description

Coq may be used interactively, in a console toplevel or in an editor with a dedicated mode (e.g. ProofGeneral for Emacs), or with the Coq compiler (producing a compiled file containing the proved theorems and the corresponding lambda-terms, to speed up the development of complex proofs requiring lots of lemmas).

Syntax As Coq is first meant to be used interactively, it provides a natural feeling for building proofs. The allowed terms of Coq can be subdivided into three categories:

1. The *vernacular* terms : these are the commands that allow one to add definitions, telling Coq we want to prove a theorem, or changing Coq's behaviour.

   Example 8. Some vernacular commands :
   - `Theorem ModusPonens:(A,B:Prop)(A /\ (A -> B)) -> B.` tells Coq we want to prove the formula $\forall A, B(A \wedge (A \Rightarrow B) \Rightarrow B)$.
   - `Definition excluded_middle := (A:Prop) A \/ ~A.` allows to associate the excluded middle formula to a variable named `excluded_middle`.
   - `Quit.` allows to quit Coq's toplevel

2. The tactics : these are the commands used *during* the proof of a theorem, to specify what kind of rule we want to use, e.g. introduction rules, elimination rules, apply a theorem,etc. There are also higher level tactics used to describe complex but repetitive proof commands.

   Example 9. Some tactics commands :
   - `Intros x P.` tells Coq to apply introduction rules to the current goal, and naming the obtained hypotheses x and P.
   - `Repeat Left.` allows to choose in a goal the leftmost innermost term. For example it would produce the goal P_1 if applied to the goal $((\ldots(P_1 \vee P_2)\ldots \vee P_{n-1}) \vee P_n)$.

3. The grammar redefinition language : it allows the user to define its own grammar for new terms or definitions he introduced, and even for complex tactics. There are example of it in section 4

3.3 Examples

Let's show a proof example with Coq :

Example 10. This example is a possible proof for the formula (A, B, C being propositions) : $\forall A, B, C(A \wedge B) \vee (A \wedge C) \Rightarrow A$

```
Theorem easyproof:(A,B,C:Prop)(A /\ B) \/ (A /\ C) -> A.
Intros.
Elim H.
Intros.
Elim H0.
Intros.
Assumption.
Intros.
Elim H0.
Intros.
Assumption.
Qed.
```

The proofs are done the top-down way. This corresponds to the following proof tree (where inference steps are annotated with the tactics commands) :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{H,H_0,H_1:A,H_2:B\vdash A}\,\texttt{Assumption.}}{H,H_0\vdash A\Rightarrow B\Rightarrow A}\,\texttt{Intros.}}{H,H_0:A\land B\vdash A}\,\texttt{Elim H0.}}{H\vdash A\land B\Rightarrow A}\,\texttt{Intros.} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{H,H_0,H_1:A,H_2:C\vdash A}\,\texttt{Assumption.}}{H,H_0\vdash A\Rightarrow C\Rightarrow A}\,\texttt{Intros.}}{H,H_0:A\land C\vdash A}\,\texttt{Elim H0.}}{H\vdash A\land C\Rightarrow A}\,\texttt{Intros.}}{H:(A\land B)\lor(A\land C)\vdash A}\,\texttt{Elim H.}}{\vdash (A\land B)\lor(A\land C)\Rightarrow A}\,\texttt{Intros}$$

`(x:sort), /\, \/, ->, ~`	These are the symbols for universal quantification (for a variable `x` of sort `sort`), conjunction, disjunction, implication, negation respectively. More generally connectors of the usual logic are represented by visually similar ASCII symbol.
`Intros`	Put the premisses of the goal in the hypotheses
`Elim H`	Apply an elimination rule for the given formula *H*
`Assumption`	Attempts to solve the current goal by telling Coq the goal is also present in the current hypotheses
`Qed`	Ends the proof and saves the generated proof term.

An additional reason that made us choose Coq, was the disponibility of a library of definitions and theorems for real numbers, as DC can be used as well in the domain of integers as in the domain of real numbers.

4 Two paths towards an implementation of DC with Coq

The proof system of DC presented in [8] isn't a sequent-style system, and the ℓ variable is context-dependent w.r.t. the $\frown$ connector. Thus the inference rules and axioms of this proof system might raise incompatibilities with the ones already present in Coq (see section 4.2). That's why we have defined two approaches so as to implement DC's proof system in Coq.

Despite those different approaches, in each case grammar redefinitions had been done, in order to ease the proof process. For example, the *diamond* meaning "for some sub-interval" (see 2.4) has been given the ASCII symbol <>.

Side-conditions involving the analysis of the formula, also had been coded with inductive definitions or functions.

We will focus in the following, on the IL part of the DC implementations, as DC-specific axioms and inference rules didn't bring much problems.

Also notice that problems we'll speak about have been solved in [9], but Isabelle/HOL being a meta-logic, it's thus easier to build a full logic in it than coping peculiarities of already existent logic one can base the implementation on. In this point of view, the shallow-embedded Coq implementation of DC in section 4.1 is comparable to the Isabelle/HOL one.

4.1 Shallow-embedded implementation

In this implementation, we simply added the missing connectors of DC with their correct type, and defined the properties of these connectors through axioms and inference rules, as described in [8].

The development libraries have been divided by logical system and by functionality : there are an IL axioms library, an IL syntax definition and an IL theorems library, and based on that, a DC axioms library, a DC syntax definition library and a DC theorems library. Hence we can say that the shallow-embedded implementation is *modular*, as one can develop a DC extension (e.g. [11]) without having to know the internals of the DC library.

Example 11. Here are the definitions of the connectors, along with grammar and syntax redefinitions. `R` is the type of real numbers, `Prop` the type of propositions, and the quotes around the definition of `point` helps Coq's parser knowing that it requires the axioms and definitions of the real numbers library.

```
Parameter l:R.
Parameter chop:Prop->Prop->Prop.
Definition point:=``l == R0``.
Definition sometime:=[P:Prop](chop True (chop P True)).
Definition always:=[P:Prop]~(sometime ~P).

Grammar constr constr5:=
  chop [constr5($c1) "^^" constr5($c2)] -> [ (chop $c1 $c2) ].

Syntax constr level 5 :
  chop [ $t1 ^^ $t2 ] -> [ [<v 0> $t1:L "^^" $t2:L ] ].

Grammar constr constr2:=
  timepoint ["[[]]"] -> [ point ]
 |sometime ["<>" constr2($c)] -> [ (sometime $c) ]
 |always ["[]" constr2($c)] -> [ (always $c) ].


Syntax constr level 2 :
  timepoint [ point ] -> [ [<v 0> "[[]]"] ]
 |sometime [ (sometime $t) ] -> [ [<h 0> "<>" $t:L] ]
 |always [ (always $t) ] -> [ [<h 0> "[]" $t:L] ].
```

With these syntax redefinitions, we can write axioms two ways, for example:

```
Axiom chop_assoc:(p,q,r:Prop)(chop (chop p q) r) <-> (chop p (chop q r)).
```

which is equivalent to:

```
Axiom chop_assoc:(p,q,r:Prop)((p ^^ q) ^^ r) <-> (p ^^ (q ^^ r)).
```

As Coq inference rules are hard-coded in its core, the inference rules for DC are defined through axioms.

Example 12. For example, the necessitation rule of DC. After having defined the axioms, we also define tactics so the user has the impression to use an inference rule instead of a simple axiom.

```
Axiom necessitation_left:(p,q:Prop)p -> ~(~p ^^ q).
Axiom necessitation_right:(p,q:Prop)p -> ~(q ^^ ~p).

Tactic Definition NecessitationLeft:=Apply necessitation_left.
Tactic Definition NecessitationRight:=Apply necessitation_right.
```

The specific side-conditions of DC are coded by inductive definitions:

Example 13. The rigidity side-conditions is (not all the induction cases are represented):

```
Inductive rigid:Prop->Prop:=
 | rig_true : (rigid True)
 | rig_false : (rigid False)
 | rig_chop : (p,q:Prop) (rigid p)/\(rigid q) -> (rigid (chop p q))
 | rig_imp : (p,q:Prop) (rigid p)/\(rigid q) -> (rigid (p -> q))
 | rig_and : (p,q:Prop) (rigid p)/\(rigid q) -> (rigid (p /\ q))
...
```

The cases when a formula is not rigid do not belong to the inductive definition, so as when trying to prove the rigidity of a formula, the inference system will be blocked. Then axioms involving rigidity are written e.g. as follows:

```
Axiom rigid_chop_left:(p,q:Prop)(rigid p)->(p ^^ q)->p.
```

Then the theorems are proved the normal way in Coq.

Example 14. The Coq proof of $\Box(A \Rightarrow B) \Rightarrow (\Box A \Rightarrow \Box B)$

```
Theorem always_distr_implies:(A,B:Prop)[](A->B) -> ([]A -> []B).
Unfold always; Unfold sometime.
Intros A B alw_A_B alw_A.
Unfold not; Intros som_nB.
Apply alw_A_B.
Monotony; Intros nB__True.
Monotony; Intros nB.
Unfold not; Intros A_B.
Apply alw_A.
Monotony; Intros nB__True2.
Monotony; Intros nB2.
Tauto.
Qed.
```

Note that the proviso of term freedom for variables ("The term θ is *free for x in* ϕ" in the section 2.5) wasn't necessary to define, as this proviso actually prevents abusive scoping of newly introduced variables. Indeed Coq is aware of variables bindings at any level.

Unfortunately, problems that can be easily solved in [9] can't have an easy solution here : as we don't define the whole logical system "from scratch", we are forced to deal with the already present logical connectors and inference rules.

Let's take for example the problem of equality, described in [9, p.21]. Usually one term can be replaced by another if they are equal. But this is not true for Duration calculus, for example:

Example 15.

$$\frac{}{\ell = 3, \ell = 3 \Rightarrow \ell = 2 ^\frown \ell = 1 \vdash 3 = 3 \Rightarrow 3 = 2 ^\frown 3 = 1}$$

This problem is addressed by constraining the equality with an "always" operator, but this can't be done for the shallow-embedded DC in Coq : the equality is already defined (this is Leibniz's one), and redefining it would be contrary to a shallow-embedding approach.

Other similar problems involve the $^\frown$ operator and ℓ (and, by extension, the duration operator, as one of the axioms states that $\int 1 = \ell$).

One way to solve such a problem, is to exploit the ability of Coq to allow one to define "plugins" so to redefine tactics and inference rules through the language Coq is written in, but this solution is a difficult one, and makes us lose the advantages deep-embedding could offer us (i.e. using already present logical connectors and inference rules without worrying).

4.2 Deep-embedded

In this implementation, all operators involved in the logic (even the ones coming from predicate calculus) are redefined. One could compare this approach to the Isabelle/HOL's one [9], as we here use Coq mostly as an inference engine. The main advantage of this approach is the absence of conflict between the implemented proof system and the proof system of the tool itself.

Example 16. Definition of a formula:

```
Inductive Formula:Type :=
 | FTrue : Formula
 | FLetter : Name -> Formula
 | FNot : Formula -> Formula
 | FOr : Formula -> Formula -> Formula
 | FExists : (DCTerm -> Formula) -> Formula
 | FChop : Formula -> Formula -> Formula
 | Flt : DCTerm -> DCTerm -> Formula
 | Feq : DCTerm -> DCTerm -> Formula
.
```

Then the validity of a formula, is stated by proving : `plvalid`*formula*. `plvalid` is an interpretation function defined by axioms. One could write an interpretation function using the original semantic of DC, i.e. interval numbers.

Then with the help of syntax redefinition allowed by Coq, one can write the formulas two ways:

Example 17. 1. `Axiom pos_interval:%''l>=0''%.`
2. `Axiom pos_interval:(plvalid (Fge length (RVal R0))).`

The special functions needed for the checking of some side-conditions are also coded inductively, but with functions this time:

Example 18. In this example, due to the nature of the existencial quantification, we do an instanciation so we can analyse further the formula.

```
Fixpoint chop_free [f:Formula]:Prop:=
Cases f of
| FTrue => True
| (FLetter _) => True
| (FNot g) => (chop_free g)
| (FOr g h) => (chop_free g) /\ (chop_free h)
| (FExists z) => (chop_free (z (RVal R0)))
| (FChop _ _) => False
| (Flt u v) => True
| (Feq u v) => True
end.
```

Then, when a proof requires to state the rigidity of a formula, one simply has to make a simplification to find out if the formula is rigid or not (`True` of `False` after the simplification, respectively).

So, even if the deep embedded approach is still at early stages, we already have positive results for this implementation:

1. The inductive definitions of formulas gives us an easier definition of side-conditions
2. The grammar and syntax redefinitions help us to have a readable system
3. The ability to interpret formulas over miscellaneous paradigms gives us the possibility to prove all the modifications of the proof system we could make for e.g. the deep-embedded implementation

But there are also drawbacks:

1. The system is not easy to extend : there are many flavours of DC out there (e.g. [6, 7, 11]), and having a static definition for the shape of formulas makes a slight modification having repercussions all over the system, grammar redefinitions and side-condition functions.
   Contrary to the shallow-embedded implementation, extending DC here requires adding the new connectors in the inductive definition above, adding axioms and

modifying interpretation functions possibly all over the library, making this deep-embedded implementation a much less modular one than the shallow-embedded one.

Note that this remark would also be true for any implementation “from scratch” (see [9]).

2. Having to define all axioms and inference rules for well-known logical operators from the beginning can be a source of bugs. Indeed, in a shallow-embedded implementation, we can make the decision to trust the definition of already present connectors. Moreover building such an implementation is time-consuming.

5 Perspectives

What made us stop our work, besides other developments, in each one of the implementation is:

- The conflicts caused by the temporal logical connector $\frown$ and the special variable ℓ with the inference rules of the proof system, for the shallow-embedded implementation
- The time-consuming task of defining the whole proof system from the beginning, for the deep-embedded implementation

In [9], the former is solved by the modification of the axioms and rules causing those conflicts : the equality is redefined, the DC-specific inference rules are modified to take in account that the inference system is a sequent-style one. Moreover, those modifications are not proved with pen and paper, but are proved with an earlier implementation of DC with PVS [12]. In short, modifications for the implementation of a proof system in a proof tool are proved with another proof tool.

This is where the deep-embedded implementation can help us : we can use it to prove modifications of the proof system that would solve the problems of the shallow-embedded implementation.

An interesting track to solve those problems, is to consider ℓ no more as a variable (because of its peculiar properties), but rather as predicates over values of the chosen numeric domain (real numbers usually) with the adequate axioms.

Example 19. E.g., with *interval_le* stating that the current time interval is lower than or equal to some value, we define the axioms relating this predicate with “normal” order relations:

$interval\_le(x) \wedge x \leq y \Rightarrow interval\_le(y)$

6 Conclusion

As explained in section 1, this work is an effort to have a proof tool for both normal logic and DC at disposal.

We built two implementations, a shallow-embedded and a deep-embedded one, having in mind different uses for them : the former would be used as a proof tool allowing one to reason on formulas and specifications made with DC, and the latter would allow one to reason on the DC proof system itself, e.g. to prove the equivalence of two interpretations of DC, or find decidability results.

The shallow-embedded implementation has shown us problems already faced in [9] with solutions that are not easy to apply with Coq, and the deep-embedded implementation, whereas long to define, can help us modify the proof system so as to solve these problems. So the same proof tool is used both to implement a proof system and to prove properties of this proof system.

7 Thanks

I would like to thank Vincent Poirriez for his enlightening remarks, Georges Mariano, the team of the Coq project for making such a powerful proof tool.

References

1. Lano, K.: Specifying reactive systems in B AMN. LNCS **1212** (1997) 242–275
2. Treharne, H., Schneider, S.: Capturing timing requirements formally in AMN. Technical Report CSD-TR-99-06, Royal Holloway, Department of computer science, Egham, Surrey TW20 0EX, England (1999)
3. Hammad, A., Julliand, J., Mountassir, H., Okalas Mossami, D.: Expression en B et raffinement des systèmes réactifs temps réel. In: AFADL'2003. (2003) 211–225
4. Zhou, C., Hoare, C., Ravn, A.: A calculus of durations. In: Information Processing Letters. Volume 10(5). (1991) 269–276
5. Dutertre, B.: On first order interval temporal logic. Technical Report CSD-TR-94-3, University of London, Department of computer science, Egham, Surrey TW20 0EX, England (1995)
6. Zhou, C., Wang, J., Ravn, A.: A duration calculus with infinite intervals. In Reichel, H., ed.: Fundamentals of Computing Theory. Volume 965 of LNCS. Springer-Verlag, Lübeck, Germany (1995) 16–41
7. Zhou, C., Guelev, D., Naijun, Z.: A higher-order duration calculus. Technical Report 167, UNU/IIST, P.O.Box 3058, Macau (1999)
8. Hansen, M., Zhou, C.: Duration calculus, logical foundations. In: Formal Aspects of Computing. Volume 9. (1997) 283–330
9. Heilmann, S.T.: Proof Support for Duration Calculus. Phd-thesis, Department of Information Technology, Technical University of Denmark (1999)
10. : Coq (1989-2003) http://coq.inria.fr.
11. Guelev, D., Hung, D.: Completeness and decidability of a fragment of duration calculus with iteration. Technical Report 163, UNU/IIST, P.O. Box 3058, Macau (1999)
12. Skakkebæk, J.U.: A Verification Assistant for a Real-Time Logic. Phd-thesis, Department of Computer Science, Technical University of Denmark (1994) Also available as Technical Report ID-TR: 1994-150.