Duration calculus :
A real-time semantic for B

Samuel COLIN[1,2], Georges MARIANO[1], Vincent POIRRIEZ[2]

[1] INRETS*, 20, rue Elisée RECLUS, BP 317 F-59666 Villeneuve d'Ascq Cedex, France
[2] LAMIH**, UMR CNRS 5830, Le Mont Houy, 59313 Valenciennes Cedex 9, France

Abstract. Among the possible approaches for expressing real-time problems with the ***B*** method, two are dominant : the use of the usual ***B*** mechanisms to define temporal constraints on the one hand, and extending ***B*** through another formalism more adapted to the real-time context on the other hand.
We define here a possible temporal semantic for ***B***, by using a temporal logic (the duration calculus), and we illustrate how this extension affects the proof mechanism used to show the soundness of abstract machines.

1 Introduction

For several years, feedback on the use of formal methods in the industrial field has been, in majority, positive. Indeed, for instance, the ***B*** method showed its strength in helping the conception of safety-critical systems (for instance, the famous example of line number 14 of the Parisian subway [BBFM99]).

However, the possibilities offered by the formal methods have to evolve at the same time as industrial needs do. Viable methods for the validation of non-functional constraints appear gradually : among them are the temporally-constrained problems. Indeed, the field of embedded devices and embedded software has great need of methods allowing the designing of solutions including time management, and also, in the case of critical systems, the checking of the *validity* of these solutions.

Such methods already exist, but often show drawbacks that can make the study of some cases difficult :

- Model-checking methods are suitable for the validation of little problems, but when composing those problems to have them interact, the number of cases increase dramatically. There are methods to avoid this problem partially, but they involve often abstract interpretation, which is actually a way of giving semantics to languages.
- Methods using temporal automatas have good compositionality properties, but do not allow the description of the step from the abstract modelisation to computer code.

Then the idea of merging these design methods with formal languages and/or methods ensues, so one can benefit from both sides :

- Easy validation of temporal constraints from the temporal formalism part
- Properties of modularity, compositionality and proximity with computer code from the language part.

In the next sections, we present a method to obtain a formal tool, allowing the checking of both functional and temporal properties of a given problem, by defining a temporal semantic for the ***B*** method. We first describe the formalism used to express this semantic, the duration calculus[3] then we remind the reader of the properties of the ***B*** method. Later on, we describe more in detail a temporal semantic for ***B***, and end up with the possibilities brought by this approach.

2 Duration calculus

From the site ([DCa]) : "The duration calculus is a modal logic for describing and reasoning about the real-time behaviour of dynamic systems, where states change over time and are represented by functions from time (reals) to the Boolean values (0 and 1). It is an extension of Interval Temporal Logic[4] [Dut95], but with continuous time, and uses integrated durations of states as interval temporal variables. Assuming finite variability of state functions, the axioms and rules of the DC constitute a complete logic (relative to Interval Temporal Logic)."

* Institut National de REcherche sur les Transports et leur Sécurité
** Laboratoire d'Automatique, de Mécanique, et d'Informatique industrielles et Humaines
[3] abbreviated as DC from now on.
[4] abbreviated IL from now on

2.1 History

Research on a temporal logic more powerful than "classical" interval temporal logic (see [Dut95]) was initiated by the ProCos[5] project of the ESPRIT[6] program, in the BRA[7] 3104 and 7071 working groups.

This initiative led in 1990 to the paper entitled "A calculus of durations" [ZHR91], which established the foundations of DC. Then, more advanced studies followed, treating such topics as completeness or decidability (see e.g. [HZ97]), then extensions to DC, like DC with infinite time intervals [ZWR95], or higher-order DC [ZGN99], for instance (see e.g. [DCb]).

There are also examples of the use of DC ([Nai99]) through the design of real-time software, as well as proof assistants for DC ([Hei8a,Hei99]). Nowadays, the most active institution in this field is the IIST[8]. Its site [DCa] gathers many links in relationship with DC.

2.2 Classical modelling of real-time problems

[SH01] presents an example of the study of the watertank problem. This case study interests us because it describes well the required steps in any other study of a real-time problem with DC. These steps are :

- Problem variables are defined
- Specifications of the problem are translated into a DC formula we call *Req*
- Design decisions are taken and also translated into a DC formula we call *Des*, such that $Des \Rightarrow Req$
- Design takes place in a dense-time context, thus one may need to make it discreet. In that case, a formula *Cont* must be found, such that $\mathcal{A} \vdash Cont \Rightarrow Des$, $\mathcal{A}$ being a formula stating the behaviour of the environment and the relations between discrete and dense variables.
- Finally, a program verifying the discrete constraints of *Cont* is written.

Notice that here, the programming step is the last one, and the language used in [SH01] is simple.

Therefore, our idea is to exploit the fact that, in the ***B*** method, the programming step is strongly connected to the proof step through refinement, so that we obtain a simplification of all these steps. Here follows a quick survey of DC with examples.

2.3 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 are also the usual ones $(=, \leq)$. The syntax of DC formulas is :

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

Let us mention the fact as functions and relations might be noted with prefix or infix notation, as syntax is not our main concern.

ℓ represents the length of the current time interval, and depends on its position in a formula w.r.t. the *chop* connector $\frown$, which chops a formula into two formulas representing the valid predicates on the first part of the time interval and the second part, respectively.

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 the expression of properties on these states and on logical relations between them.

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 following function is not finitely variable over an interval of real numbers :

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

[5] Provably correct systems.
[6] European Strategic Program for Research in Information Technology.
[7] Basic Research Action.
[8] International Institute for Software Technology, affiliated with the United Nations University.

2.4 DC examples

Some examples are inspired from [HZ97]:

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 the first part of the time interval is 10 time units long, and the second part is 5 time units long.
3. $\mathbf{true} \frown (\phi \frown \mathbf{true})$ 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".

2.5 Proof system

We will only underline in this section some hard points of the proof system of [HZ97], on which are based some of the implementations described in section 2.7. Now, for some definitions beforehand:

Definition 1. A DC formula is called *rigid* if it does not contain any state variable, propositional letter or ℓ symbol. It is otherwise called *flexible*

Definition 2. A DC formula is called *chop free* if the $\frown$ does not occur in the formula

Definition 3. The term θ is *free for x in* ϕ if x does not 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 instantiation. The axioms of the proof system are distributed between those coming from IL, and those coming from DC. For example:

Example 1. Some IL axioms:

$\ell \geq 0$	The length of a time interval can not 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 2. 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 those 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 do not hold hypotheses, as in sequent calculus, for example. Thus some of them will not be valid if coded "as is" in a prover (these problems have already been solved in the particular case of [Hei99]).

For example:

Example 3. Some DC inference rules (inherited from IL inference rules):

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

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

2.6 DC with iteration

Many extensions to DC have been proposed to deal with the fact that formulas are only considered over the current interval (see 2.1). One of them is DC with iteration[9], which allows to reason on an arbitrary number of time subintervals. The next section is a more detailed presentation of DC*.

An addition to DC Some additions to the proof system of DC are done :

- A logical connector *, which chops an interval an arbitrary (possibly zero) number of times.
- Three axioms:

  DC_1^* $\ell = 0 \Rightarrow \phi^*$
  DC_2^* $(\phi^* \frown \phi) \Rightarrow \phi^*$
  DC_3^* $(\phi^* \wedge \psi \frown \mathbf{true}) \Rightarrow$
  $(\psi \wedge \ell = 0 \frown \mathbf{true}) \vee ((\phi^* \wedge \neg\psi \frown \phi) \wedge \psi) \frown \mathbf{true}$

  The two first axioms describe the properties of the iteration connector, as could be expected from a repetition operator. The DC_3^* axiom is needed to make deductions like $\phi \Rightarrow \psi \vdash \phi^* \Rightarrow \psi^*$. Informally, this axioms helps the characterising of the possible cases for an interval in which a formula ψ holds in its prefix and for a finite number of sub-intervals the formula ϕ holds.
- Some additional notations : $\begin{array}{l} \phi^+ \mathrel{\widehat{=}} \phi \frown (\phi^*) \\ \phi^0 \mathrel{\widehat{=}} \ell = 0 \\ \phi^k \mathrel{\widehat{=}} \underbrace{\phi \frown \ldots \frown \phi}_{k \text{ times}} \text{ for } k > 0 \end{array}$
- An inference rule :

  ω: if $\forall k < \omega, H((\lceil\!\lceil S \rceil\!\rceil \vee \lceil\!\lceil \neg S \rceil\!\rceil)^k)$ then $H(\mathbf{true})$

DC* is preferred over DC when hybridising with a programming language, due to the iteration connector, which allows a more accurate representation of the loops (like the *while* control structure).

Decidable subset In [GH99], a subset of DC* has been proven decidable :

$$\phi ::= \ \ell = 0 \mid \lceil\!\lceil S \rceil\!\rceil \mid a \leq \ell \mid \ell \leq a \mid (\phi \vee \phi) \mid (\phi \wedge \phi) \mid (\phi \frown \phi) \mid \phi^*$$

The formulas of this subset are called *simple*. We will denote this subset SDC*. This subset is interesting, as the decision procedure is easy to write (an example can be found in [GH99] for a subclass of the simple formulas). Therefore, when describing a temporal semantic for a programming language, it is important to stay as close as possible to such decidable subsets, in order to avoid problems of provability.

2.7 Proof support

The interested reader can find some of the tools presented here at [DCa]. There are many tools based on model-checking methods to prove validity of DC formulas ([Pan01] is such a tool), but we focus in this section on proof assistants (whether they be automatic or not), because the ***B*** method relies on a theorem prover. Then, any extension to ***B*** should be done with having in mind the availability of this extension in the chosen theorem prover. Moreover, some proofs happen to be hard enough not to be proved automatically, then requiring user interaction with the prover.

There are two ways to include DC support in a general-purpose proof assistant : with a shallow-embedding implementation (adding the missing axioms to the theorem prover) or a deep-embedding implementation (all rules of DC are rewritten). Let us present some of them :

PC/DC Based on PVS[10], this library used deep-embedding for its implementation. While proving the soundness of DC rules is made easier, doing proofs "the syntactical way" may be difficult.See [Hei8a] for more details.

[9] Noted DC* from now on
[10] Prototype Verification System.

Isabelle/DC The method used here is shallow-embedding[11]. This systems allows easy proofs of DC, but the drawback appears at the conception step : Isabelle/HOL is a meta-engine, thus all calculi DC depend on have to be written again. See [Hei99] for a more in-depth view of the design and use of this system.

Coq We have also developed libraries for Coq (presented in [CPM03]). These libraries illustrate both shallow-embedding and deep-embedding approaches. While the former is used to actually make proofs, the latter can be used to manipulate and reason about the formalism .

3 B method

We will suppose the reader is familiar with the B method or formal methods of the same kind (*Z*, for instance). We will simply recall its greatest characteristics here.

The B method is based on the following formalisms : predicate calculus with set theory, generalised substitutions, and refinement. The B machines are made up of clauses, the most important ones being the *INVARIANT* and the *OPERATIONS*. The soundness of a B machine is checked with logical formulas, called *proof obligations*, built with the *INVARIANT* and either an operation or the initialisation clause of the machine.

We are particularly interested in this building rules, as they are based on Hoare's triples and weakest precondition calculus : Let *pre* and *post* be logical formulas, and S a substitution, then $\{pre\}[S]\{post\}$ is a valid triple if *pre* establishes the weakest precondition of S w.r.t. *post*. This weakest precondition, noted in B $[S]post$, is calculated with the rules in figure 1 (GSL is an abbreviation for "generalised substitutions language"). Then, we can prove the $\{pre\}[S]\{post\}$ is valid by proving the logical formula $pre \Rightarrow [S]post$.

In the ***B*** method, the postcondition is simply replaced by the machine *INVARIANT*. Similarly, the refinement of an operation is proved to be correct if, when called within a (possibly) weaker precondition, it establishes the same result as its abstraction. The reader can refer to [Abr96] for more details.

GSL	$[GSL]P$	description
$skip$	P	"Do nothing" substitution
$x := E$	$P[E/x]$	All the occurrences of x are replaced by E
$g \vert S$	$g \wedge [S]P$	Precondition
$g \Longrightarrow S$	$g \Rightarrow [S]P$	Guard
$S;T$	$[S]([T]P)$	Sequence
$S [] T$	$[S]P \wedge [T]P$	Bounded choice
$@x.S$	$\forall x [S]P$	Unbounded choice
$S \Vert T$	simplified with rewriting rules	Non-deterministic substitution
WHILE C DO S VARIANT V INVARIANT I	$\forall x(I \wedge P \Rightarrow [S]I)$ $\forall x(I \Rightarrow V \in \mathbb{N})$ $\forall x(I \wedge P \Rightarrow [n := V][S](V < n))$ $\forall x(I \wedge \neg P \Rightarrow R)$	

Fig. 1. Calculus of the weakest precondition

3.1 Temporally constrained problems with "classical" B

The use of the set theory allows the developer to adapt the proof to its framework. Therefore it is possible to express models with temporal needs. In [Lan98], a communication protocol between a SmartCard and its reader is modelled in B. A component with temporal constraints is also described in [TS99]. Unfortunately, there are still some limitations :

– In [Lan98], temporal constraints do not include hard temporal needs, i.e. quantified time intervals, but rather constraints on the order of the steps of the protocol.

[11] actually, due to the particularity of Isabelle/HOL, which a meta-engine, this embedding is also denoted as *external*.

– In [TS99], the model calls a clocks it updates itself, and does not allow the “triggering” of the different operations, although it allows to check the operations function correctly if triggered during specific time intervals. Moreover, with this approach, proof obligations can become complex, even more if they are composed together so they communicate, but function according to different clocks.

Hence, even if it is possible to model temporally constrained problems in "classical" B, the complexity of proof obligations potentially generated, as well as the external modelling of time (the problem is not subject to time, but handles it through a machine acting as a clock), limit the class of problems that can be addressed.

3.2 *Event B* and temporal constraints

Event B (see e.g. [Abr00]) is an extension of B allowing abstract specification of reactive systems. Thus, it is easier to model protocols requiring concurrency, or systems described by events that can happen in it.

In [HJMO03], timed automatas are used in conjunction with ***event B*** to model timed event systems. The presented example is a railroad crossing, in which the train can take several states (modelled by a set), and to each event is associated a transition having the train go from one state to the next. We have here a correspondence between the events' system of the ***B*** machine and a timed automata representing the different states the system can be in, as well as associated transitions. This approach has two advantages :

– Many clocks can be defined for one or several problems, which allows the expression of numerous temporal properties of the model
– The refinement of timed automata is intuitive, and gives the ability to check that many properties of the model are kept at the refinement step.

There are also subtle points one has to take care of :

– The events can not be triggered explicitly. One can only act on variables of the model, ensuring during the proof that the guard of the awaited event is triggered. This keeps ***event B*** away from an immediate implementation.[12]
– The more clocks there are in the model, the more constraints on them may appear in events' guards, hence the harder the proof obligations can be. Besides, here the time is still not considered as implicit, but appears in the form of clocks.

4 Temporal extension for "classical" B

The extension presented here is based on [SH01], in which the presented method rather follows the design steps presented in section 2.2. So, instead of having logical specifications as a basis to validate a solution to a temporally constrained problem, we use the substitutions of ***B*** to describe the dynamical behaviour by giving them a temporal semantic, so as to extract temporal informations we are interested in validating.

We define this semantic under the usual ***B*** design hypotheses, that is to say that there is no concurrency, and the substitutions terminate. Moreover, we suppose the true synchrony hypothesis, stating that affectations take zero time. This hypothesis is made because, in practice, the delays used to synchronise the different components of a model are very big w.r.t. the execution time of little instructions (affectations, additions,etc), hence the latter do not play a big role in the core of the modelled problem. Let us notice though, that the possibility remains to add delay statements if one needs to take into account the duration (even little) of particular substitutions.

4.1 Temporal semantic of substitutions

We present in figure 2 a possible semantic for B substitutions. The formula $dur([\text{GSL}], P)$ means "the duration of the substitution GSL, knowing the postcondition P".

Let us first notice that the calculus of a duration formula bases itself on a predicate : this predicate represents the state whose evolution we want to watch during the "execution" of the substitution (more on this in section 4.2).

As an example, let us make the proof of the rule for the *While*, as made in [SH01] (for space reasons, we can not recall the definitions of WDC* here). The inference rule can be written as :

[12] This is half a problem, though, as ***event B*** has been designed for abstract modelisation, and there are systems allowing to explicit the operational semantic of events ([BF03]).

GSL	$dur([\text{GSL}],P)$
skip	$\lceil\rceil$
$x := E$	$\lceil\rceil$
delay d	$(\ell = d) \wedge \lceil P \rceil$
$g \vert S$	$dur([S],P)$
$g \Longrightarrow S$	$dur([S],P)$
$S;T$	$dur([S],([T]P))^\frown dur([T],P)$
$S [] T$	$dur([S],P) \vee dur([T],P)$
$@x.S$	$dur([S],P)$
$S \Vert T$	transformed through rewriting rules
WHILE C DO S VARIANT V INVARIANT I	$dur([S],I)^*$

Fig. 2. Calculus of a duration formula from a generalised substitution

$$\frac{\{[S]I\}[S,dur([S],I)]\{I\} \qquad I \wedge C \Rightarrow [S]I \qquad I \wedge \neg C \Rightarrow P}{\{I\}\begin{bmatrix}\text{WHILE } C & \\ \text{DO } S & ,dur([S],I)^* \\ \text{INVARIANT } I & \end{bmatrix}\{P\}}$$

Please note that we forgot the side-conditions for the *VARIANT*, as it is not used in the proof.
We keep the definition of real-time rules for the *while* : $\mathcal{M}_{fin}(WHILE) \equiv (\lceil C \rceil^0{}^\frown \mathcal{M}_{fin}(S)^\frown SCH_i)^{*\frown} \lceil \neg C \rceil^0$
Now let us prove that $\lceil I \rceil^0{}^\frown \mathcal{M}_{fin}(WHILE) \Rightarrow_{WDC^*} \mathcal{M}_{fin}(WHILE)^\frown \lceil P \rceil^0$

1. Assumption : $\{[S]I\}[S,dur([S],I)]\{I\}$
2. Assumption : $I \wedge C \Rightarrow [S]I$
3. Assumption : $I \wedge \neg C \Rightarrow P$
4. By 1, $\lceil [S]I \rceil^0{}^\frown \mathcal{M}_{fin}(S) \Rightarrow_{WDC^*} \mathcal{M}_{fin}(S)^\frown \lceil I \rceil^0$
5.

$$\begin{array}{lr}
\lceil I \rceil^0{}^\frown \mathcal{M}_{fin}(WHILE) & \\
\Rightarrow \lceil I \rceil^0{}^\frown(\lceil C \rceil^0{}^\frown \mathcal{M}_{fin}(S)^\frown SCH_i)^{*\frown} \lceil \neg C \rceil^0 & (\text{definition for } \textit{while}) \\
\Rightarrow \lceil I \wedge \neg C \rceil^0 \vee (\lceil I \wedge C \rceil^0{}^\frown \mathcal{M}_{fin}(S)^\frown SCH_i)^{+\frown} \lceil \neg C \rceil^0 & (WDC^*) \\
\Rightarrow \lceil I \wedge \neg C \rceil^0 \vee (\lceil [S]I \rceil^0{}^\frown \mathcal{M}_{fin}(S)^\frown SCH_i)^{+\frown} \lceil \neg C \rceil^0 & (2.) \\
\Rightarrow \lceil I \wedge \neg C \rceil^0 \vee (\mathcal{M}_{fin}(S)^\frown \lceil I \rceil^0{}^\frown SCH_i)^{+\frown} \lceil \neg C \rceil^0 & (4.) \\
\Rightarrow \lceil I \wedge \neg C \rceil^0 \vee (\mathcal{M}_{fin}(S)^\frown SCH_i{}^\frown \lceil I \rceil^0)^{+\frown} \lceil \neg C \rceil^0 & (\text{conservation of variables' state}, COND2) \\
\Rightarrow \lceil I \wedge \neg C \rceil^0 \vee (\mathcal{M}_{fin}(S)^\frown SCH_i)^{+\frown} \lceil I \wedge \neg C \rceil^0 & (WDC^*) \\
\Rightarrow (\mathcal{M}_{fin}(S)^\frown SCH_i)^{*\frown} \lceil I \wedge \neg C \rceil^0 & (WDC^*) \\
\Rightarrow (\lceil C \rceil^0{}^\frown \mathcal{M}_{fin}(S)^\frown SCH_i)^{*\frown} \lceil I \wedge \neg C \rceil^0 & (WDC^*) \\
\Rightarrow (\lceil C \rceil^0{}^\frown \mathcal{M}_{fin}(S)^\frown SCH_i)^{*\frown} \lceil \neg C \rceil^0{}^\frown \lceil I \wedge \neg C \rceil^0 & (WDC^*) \\
\Rightarrow \mathcal{M}_{fin}(WHILE)^\frown \lceil I \wedge \neg C \rceil^0 & (\text{definition of } \textit{while}) \\
\Rightarrow \mathcal{M}_{fin}(WHILE)^\frown \lceil P \rceil^0 & (3., WDC^*)
\end{array}$$

□

Then, we prove the duration formula (the proof is similar to the one in [SH01]) :

1. Assumption : $\{[S]I\}[S,dur([S],I)]\{I\}$
2. By 1, $\prod(\lceil [S]I \rceil^0{}^\frown \mathcal{M}_{fin}(S)) \Rightarrow dur([S],I)$
3.

$$\begin{array}{lr}
\prod(\lceil I \rceil^0{}^\frown \mathcal{M}_{fin}(WHILE)) & \\
\Rightarrow \prod(\lceil I \wedge \neg C \rceil^0) \vee \prod((\lceil [S]I \rceil^0{}^\frown \mathcal{M}_{fin}(S)^\frown SCH_i)^+)^\frown \prod(\lceil \neg C \rceil^0) & (\text{definition of } \textit{while} \text{ and monotony of } \prod) \\
\Rightarrow \ell = 0 \vee (dur([S],I)^\frown \ell = 0)^{+\frown} \ell = 0 & (2., \text{definition of } \prod) \\
\Rightarrow \ell = 0 \vee dur([S],I)^+ & (DC^*) \\
\Rightarrow dur([S],I)^* & (DC^*)
\end{array}$$

□

Thus we have proved that the rules for the *while* are correct w.r.t. its definition in WDC*. The proofs for other substitutions are similar : for instance, the proofs for the bounded and unbounded choice happen to be compatible with the definition of the rules for the *if then else* structure of [SH01]. That is the least one could have expected from two formalisms originally based on Hoare's triple.

Examples The following examples are not meant to reflect actual temporally-constrained problems, but rather to give the reader an intuition of the mechanism of the generation of duration formulas from the operations.

```
S =
  BEGIN
    delay 3;
    IF
      x ≥0
    THEN
      delay 1; skip;
    ELSE
      delay 2; x:=-x;
    END;
  END
```

Fig. 3. Example 1

Example 1 The duration formula associated with the substitution in figure 3, knowing that, after execution, we have $x \geq 0$, is :

$$\begin{aligned}
\mathrm{dur}([S], x \geq 0) &= \mathrm{dur}([delay 3], [IF...](x \geq 0)^\frown \mathrm{dur}([IF...], x \geq 0) \\
&= \begin{array}{l} \mathrm{dur}([delay\ 3], (x \geq 0 \Rightarrow x \geq 0) \wedge (\neg x \geq 0 \Rightarrow -x \geq 0) \\ {}^\frown \mathrm{dur}([x \geq 0 \Longrightarrow delay\ 1; skip], x \geq 0) \vee \mathrm{dur}([\neg x \geq 0 \Longrightarrow delay\ 2; x := -x], x \geq 0) \end{array} \\
&= \begin{array}{l} \ell = 3 \wedge \lceil\!\lceil x \geq 0 \Rightarrow x \geq 0) \wedge (\neg x \geq 0 \Rightarrow -x \geq 0 \rceil\!\rceil \\ {}^\frown(\ell = 1 \wedge \lceil\!\lceil x \geq 0 \rceil\!\rceil) \vee (\ell = 2 \wedge \lceil\!\lceil -x \geq 0 \rceil\!\rceil) \end{array} \\
&= \ell = 3 \wedge \lceil\!\lceil \mathrm{true} \rceil\!\rceil^\frown(\ell = 1 \wedge \lceil\!\lceil x \geq 0 \rceil\!\rceil) \vee (\ell = 2 \wedge \lceil\!\lceil -x \geq 0 \rceil\!\rceil)
\end{aligned}$$

```
S =
  BEGIN
    delay1;
    ANY
      y
    WHERE
      y ∈ { a |∃b b >0 ∧a =2 ∗b }
    THEN
      x:=y
    END;
  END
```

Fig. 4. Example 2

Example 2 The duration formula associated with the substitution in figure 4, knowing that after execution we have $x \geq 0$, is :

$$\begin{aligned} \mathrm{dur}([S], x \geq 0) &= \ell = 1 \wedge \lceil\!\lceil \forall y, y \in \{a | \exists b, b > 0 \wedge a = 2 * b\} \Rightarrow y > 0 \rceil\!\rceil \\ &= \ell = 1 \wedge \lceil\!\lceil \mathrm{true} \rceil\!\rceil \end{aligned}$$

4.2 Use in abstract machines

Temporal correctness of operations Let us start this section with two remarks :

- Verifying the consistency of the machine invariant with the generated temporal formulas is irrelevant : the invariant helps ensure the variable of the machines are valid between each operation execution, namely at a particular point in time. In DC, statements made at a point in time can be reduced to the axiom $\int 0 = 0$, thus they are not useful for the proof of temporal formulas
- Different operations inside a machine, generally speaking, have different behaviours : the invariant can be seen as the common subset of what all these operations establish. That is why we need to give the developer a way of expressing more precise requirements for each operation. These requirements will be predicates we can base the duration formulas' generation on. This remark relates to those made in [TS99, section 6.4], where to each operation corresponds a timed operation, in which temporal constraints are expressed in the precondition.

Then, all we have to do is associate to each substitution of the system a temporal constraint it must establish. To this end, we propose to add a new substitution we name TIMING, whose role is to state the temporal constraint of the substitution it guards. Finally, we have to provide the predicate whose evolution we will watch in the substitution (the P of figure 2). Then, we have two possibilities :

- Use the *INVARIANT* of the machine : assuming that the usual B proof obligations have showed that the invariant is established, it is an informative representation of the state of the variables of the system. The drawback is that not all the predicates composing the invariant might be useful, and may make the resulting duration formula unwieldy. Likewise, predicates corresponding to typing of variables will not be much useful, as a type-checking step should have been realized at the time.
- Another solution is to use a predicate representing what the operation will have realized, while staying consistent with the invariant of the machine and not containing irrelevant predicates : this solution is presented in [Pet03, section 4.3.3], in the form of postconditions. Postconditions possess all the characteristics mentioned above : they contain only predicates relevant for the associated operation, and they are informative enough to represent the result of the operation.

TIMING
    Substitution
POST
    Postcondition (in the form of a predicate)
REQUIRES
    Temporal constraint
END

Fig. 5. Form of a substitution with a temporal constraint

Thus a substitution with a temporal constraint will have the shape indicated in figure 5. Then, in order to prove the temporal correctness of the substitution, it suffices to generate the corresponding trace with the provided postcondition, and check the constraint is verified, thus :

$$dur([\text{Substitution}], \text{Postcondition}) \Rightarrow \text{Temporal constraint}$$

Example 1 The generated formula for the example of figure 6 on the left side is : $\ell = 1 \wedge \lceil\!\lceil x - 1 \geq 0 \rceil\!\rceil \Rightarrow \Box(\lceil\!\lceil x \geq 0 \rceil\!\rceil)$, which is easy to prove. Note also that the postcondition is intuitively verified.

```
x ⟵ Example1 =
  TIMING
    PRE
      x ≥1
    THEN
      delay 1; x:=x-1;
    END
  POST
    x ≥0
  REQUIRES
    □(⟦x ≥ 0⟧)
  END
```

```
Example2 =
  TIMING
    x:=100;
    WHILE
      x ≥1
    DO
      x:=Example1
    INVARIANT
      x≥0
    VARIANT
      x
    END
  POST
    true
  REQUIRES
    □(⟦x ≥ 0⟧)
  END
```

Fig. 6. An example of specification

Example 2 In this example (figure 6, on the right side), two substitutions with temporal constraints are nested. We then have two possibilities :

- Prove the internal substitution, then forget the internal postcondition and time constraint to prove the external substitution
- Prove the internal substitution, and, instead of re-calculating the part corresponding to the internal substitution, use the time constraint it establishes.

The generated duration formulas is : $(\Box(\llbracket x \geq 0 \rrbracket))^* \Rightarrow \Box(\llbracket x \geq 0 \rrbracket)$. Though we have not enough space to develop the proof here, one can see the formula hold : in the case the interval is $\ell = 0$, then there is no non-zero interval on which the state $\llbracket x \geq 0 \rrbracket$ is false, and in the case the interval is non-zero, then the properties of the $\Box$ operator allow us to prove the consequence hold.

We have also replaced the calculus of $\mathrm{dur}([Example1], x \geq 0)$ with the temporal specification established by *Example1*. Indeed, it allows us better precision than using the only invariant, as the postcondition of *Example1* guarantees the invariant is respected (the proof is done at the usual step of verification of the correctness of the proof obligations).

Modularity Section 4.2 illustrates the way we can validate nested substitutions with temporal constraints. Now, this is the way proof obligations for operations in "classical" ***B*** are generated. We do not want, as stated in [Pet03, 4.3.3], to depend on the code of the called operations to keep a *software component* view of ***B*** machines. To this end, we need :

- Code independence towards the operations called in the included machines, i.e. the called operation must satisfy a contract with the help of which we will be able to validate the current operation, without knowing the code of the called operation.
- Limited scope of the variables, i.e. an operation need not export a contract containing variables from an included machine.

The first constraint is achieved, when generating the duration formula, by using the duration formula of the called operation in the same manner as the nested substitutions of section 4.2.

It is up to the developer to achieve the second constraint, as he is the one who determines what each operation must guarantee. Then, the verification of duration formula is done as in section 4.2.

Refinement Refinement in ***B*** allows us the checking of that the result of an operation is not inconsistent with the one of the operation it refines. Thanks to this, we can know that what is calculated by the operation keeps certain properties. It allows also the use of new variables, more concrete ones (in the programming sense), to realise these calculi. The verification is then made by expressing the relation between the new variables and those of the refined machine through a so-called *gluing* invariant.

However, temporal verification is about the *way* the operation unfolds, which may cause problems : thus, a refined operation can have a more precise temporal trace, but is not allowed to redefine its working steps. So, we have several cases :

```
MACHINE
   LittleExample
VARIABLES
   y
INVARIANT
   y ∈ 𝔽(NAT1)
INITIALISATION
   y :=∅
OPERATIONS
   lire(n) =
      PRE
         n ∈ NAT1
      THEN
         y :=y ∪{n}
      END;
m ⟵maximum =
      PRE
         y ≠∅
      THEN
         m :=max(y)
      END
```

```
REFINEMENT
   LittleExample1
REFINES
   LittleExample
VARIABLES
   z
INVARIANT
   z = max(y ∪{0})
INITIALISATION
   z :=0
OPERATIONS
   lire(n) =
      PRE
         n ∈ NAT1
      THEN
         z :=max(z,n)
      END;
m ⟵maximum =
      PRE
         z ≠0
      THEN
         m :=z
      END
```

Fig. 7. Machine LittleExample and its refinement

Direct demonstration If ϕ is the duration formula of the operation, and ϕ' the duration formula of its refinement, check $\phi' \Rightarrow \phi$. In that case :

- If new, more concrete, variables are introduced, replacing those from the refined machine, the formula is generally not provable. For instance, in figure 7, the variable of the machine, a set, is refined by an integer variable. This means that, although the new variable corresponds functionally to the refined one (proved by the ***B*** proof obligations), the way they are are calculated is different, and then the different states the variable can have during the calculus can differ in the abstraction and in the refinement
- The temporal trace must also be strongly similar to the one of the abstraction. For instance, in figure 8, the operation *oper*2 can not refine *oper*1, because the formula to prove would be $\ell = 5 \Rightarrow \ell = 0$, which is false. This way of designing would force us to have a great temporal precision at the beginning of the modelisation, and that is not what we wish for the designer.

The contract approach The contract approach is another approach to ensure that a refinement will not contradict the operations that might use it : the new operation also fulfils the temporal constraints of the operation it refines. This corresponds to an *operation refinement* approach.

In the example figure 8, we achieve this by removing the temporal constraints from *oper*2, calculating its temporal trace with the postcondition of *oper*1, and checking this trace validates the temporal constraints of *oper*1. We have chosen to adopt this more flexible approach, for the temporal validation of a refinement. We have just proposed a formal description of postconditions and their refinement in [CMP04]. Then, the only remaining problem would be the addition of new variables refining the ones from the abstract machine (which corresponds to *data refinement*).

oper1 = **TIMING** skip **POST** $x = x$ **REQUIRES** $\ell \leq 10$ **END**	oper2 = **TIMING** delay 5; **POST** $x = x$ **REQUIRES** $\ell \leq 10$ **END**

Fig. 8. Example of temporally different refinements

5 Conclusion

After having presented DC and one of its extension suitable for the verification of real-time programs, as well as some of its properties, we have reminded the reader of the foundations of the ***B*** method, and some of the way used to express temporal problems with it. We have seen that it is possible to use :

- Either the bare formalism, but then we have to face difficulties in the designing and the proof steps
- Or ***event B*** completed with known methods, coming from the real-time community, but in that case we have to face a lack of tools.

Hence we have showed that another way is possible : extending the most used formalism with a temporal logic. This allowed us to define a temporal semantic for B substitutions without modifying the foundations of the semantic, based on set theory. In fact, the opposite effect was achieved : DC, being defined partly on top of the predicate calculus, allowed us to use the substitutions to find their temporal trace with regard to a postcondition that we know to be correct (the correctness proof was made during the classical ***B*** design stage).

Moreover, the temporal validation step of the operations justifies the need for the modular validation of B machines, by requiring the use of postconditions, and by using the same mechanism as in the call of operations of included machines.

6 Perspectives

Now that we are able to specify and verify a problem with temporal constraints in ***B*** under the usual hypotheses (no concurrency, termination), what is left is to remove these hypotheses in order to benefit from the expressiveness of DC, in order to check, for instance, the railroad crossing problem (see [CBR93] for a general description). With this aim in view, we can express problems with more subtle temporal requirements (for example replacing the temporal logic used in [LFD96] by DC), with an easier treatment of non-terminating (for instance, the validation of a mutual exclusion protocol, as in [SH01]).

It can also be interesting to use DC as a way of expressing and validating fairness and liveness constraints in ***event B***, by expressing time quantifications more naturally (i.e. without using clocks or machines manipulating time).

Another interest of DC is the ability to express timed automatas (see [JH00]), and vice versa, provided certain constraints are respected : it then allows the use of several methods at the same time to model a temporal problem, i.e. a timed automata to specify the temporal behaviour of the model, and use the ***B*** method to build the implementation step by step, with the help of refinement.

In the end, ***B*** is also used to validate UML models (see [ML02]). A temporal extension to B will allow the checking of OCL models with temporal constraints, those being inherited from the corresponding UML model, since UML 2.0 will include notions of time.

References

[Abr96] Jean-Raymond Abrial. *The B Book - Assigning Programs to Meanings.* Cambridge University Press, August 1996.

[Abr00] Jean-Raymond Abrial. Event driven sequential program construction. École Jeunes chercheurs en programmation, March 2000.

[BBFM99] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. METEOR : A successful application of B in a large project. In Wing et al. [WWD99], pages 369–387.

[BF03] Jean-Paul Bodeveix and Mamoun Filali. Machines virtuelles pour le B événementiel. [IRI03], pages 227–242.

[CBR93] Heitmeyer C.L, Labaw B.G, and Jeffords R.D. A benchmark for comparing different approaches for specifying and verifying real-time systems. In *Tenth IEEE Workshop on Real-Time Operating Systems and Software*. IEEE Computer Society Press, 1993. http://chacs.nrl.navy.mil/personnel/heitmeyer.html.

[CMP04] Samuel Colin, Georges Mariano, and Vincent Poirriez. A natural extension of B substitutions : postconditions. Technical report, LAMIH/ROI, 2004. http://www.univ-valenciennes.fr/ROAD/WP/.

[CPM03] Samuel Colin, Vincent Poirriez, and Georges Mariano. Thoughts about the implementation of the duration calculus with coq. In *4th International Workshop on the Implementation of Logics*, volume Technical report ULCS-03-018. University of Liverpool, september 2003. http://www.csc.liv.ac.uk/research/techreports/.

[DCa] http://www.iist.unu.edu/dc/.

[DCb] http://www.iist.unu.edu/home/Unuiist/newrh/III/1/page.html.

[Dut95] Bruno Dutertre. Complete proof systems for first order interval temporal logic. In *Logic in Computer Science*, pages 36–43, 1995.

[GH99] Dimitar P. Guelev and Dan Van Hung. Completeness and decidability of a fragment of duration calculus with iteration. In *Asian Computing Science Conference (ASIAN'99)*, volume 1742 of *LNCS*, pages 139–150, Phuket, Thailand, December 1999. Springer-Verlag. Also presented at International Conference on Mathematical Foundation of Informatics, Hanoi, October 25-28, 1999.

[Hei99] Søren T. Heilmann. *Proof Support for Duration Calculus*. Phd-thesis, Department of Information Technology, Technical University of Denmark, Januar 1999.

[Hei8a] Søren T. Heilmann. *PC/DC Users Guide*, 1998(a).

[HJMO03] A. Hammad, Jacques Julliand, H. Mountassir, and D. Okalas Ossami. Expression en B et raffinement des sytèmes réactifs temps réel. [IRI03], pages 211–226.

[HZ97] M.R. Hansen and C.C. Zhou. Duration calculus, logical foundations. In *Formal Aspects of Computing*, volume 9, pages 283–330. 1997.

[IRI03] IRISA. *Approches Formelles dans l'Assistance au Développement de Logiciels*, IRISA Rennes – France, January 2003. IRISA.

[JH00] Zhao Jianhua and Dang Van Hung. Checking timed automata for some discretisable duration properties. In *Journal of Computer Science and Technology*, volume 15, pages 423–429. September 2000.

[Lan98] Jean-Louis Lanet. Using the B method to model protocols. In *AFADL'98* [LIS98], pages 79–90.

[LFD96] Kevin Lano, J. Fiadeiro, and Jeremy Dick. Extending B AMN with concurrency. Technical report, Dept. of Computing, Imperial College, 1996.

[LIS98] LISI/ENSMA. *Approches Formelles dans l'Assistance au Développement de Logiciels*, Téléport2 - Avenue 1 - BP109 - 86960 FUTUROSCOPE Cedex, October 1998. LISI/ENSMA.

[ML02] R. Marcano and N. Levy. Using B formal specifications for analysis and verification of UML/OCL models. In *Workshop on consistency problems in UML-based software development. 5th International Conference on the Unified Modeling Language*, Dresden, Germany, September 2002.

[Nai99] Zhan Naijun. Another formal proof for deadline driven scheduler. Technical Report 169, UNU/IIST, P.O. Box 3058, Macau, august 1999.

[Pan01] P.K. Pandya. Specifying and deciding quantified discrete-time duration calculus formulae using dcvalid. In *RT-TOOLS'2001*, Aalborg, August 2001. (affiliated with CONCUR 2001). Technical report TCS-00-PKP-1, Tata Institute of Fundamental Research, Mumbai, 2000.

[Pet03] Dorian Petit. *Génération automatique de composants logiciels sûrs à partir de spécifications formelles B*. Thèse de doctorat, Université de Valenciennes et du Hainaut-Cambrésis, December 2003.

[SH01] François Siewe and Dan Van Hung. Deriving real-time programs from duration calculus specifications. In *11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2001)*, volume LNCS 2144, pages 92–97, Livingston-Edinburgh, Scotland, september 2001. Springer-Verlag. (Technical Report 222, UNU-IIST, P.O. Box 3058, Macau, December 2000).

[TS99] Helen Treharne and Steve Schneider. Capturing timing requirements formally in AMN. Technical Report CSD-TR-99-06, Royal Holloway, Department of Computer Science, University of London, Egham, Surrey TW20 0EX, England, June 1999.

[WWD99] Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors. *Proceedings of FM'99: World Congress on Formal Methods*, number 1709 in Lecture Notes in Computer Science (Springer-Verlag). Springer Verlag, September 1999.

[ZGN99] C.C. Zhou, D.P. Guelev, and Z. Naijun. A higher-order duration calculus. In *Symposium in Celebration of the Work of C.A.R. Hoare*, Oxford, september 1999. (Technical report 167, UNU-IIST, P.O.Box 3058, Macau, July 1999).

[ZHR91] C.C. Zhou, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. In *Information Processing Letters*, volume 10(5), pages 269–276. Dezember 1991.

[ZWR95] C.C. Zhou, J. Wang, and A.P. Ravn. A duration calculus with infinite intervals. In H. Reichel, editor, *Fundamentals of Computing Theory*, volume 965 of LNCS, pages 16–41. Springer-Verlag, Lübeck, Germany, 1995.