HAL
archives-ouvertes.fr

From Discrete Duration Calculus to Symbolic Automata

Laure Gonnord, Nicolas Halbwachs, Pascal Raymond

To cite this version:

Laure Gonnord, Nicolas Halbwachs, Pascal Raymond. From Discrete Duration Calculus to Symbolic Automata. Third International Workshop on Synchronous Languages, Applications, and Programs (SLAP 2004), Mar 2003, Barcelona, Spain. 153 (4), pp.3-18, 2006, <10.1016/j.entcs.2006.02.022>. <hal-00198433>

HAL Id: hal-00198433

https://hal.archives-ouvertes.fr/hal-00198433

Submitted on 17 Dec 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

From Discrete Duration Calculus to Symbolic Automata

Laure Gonnord [2] Nicolas Halbwachs [3] Pascal Raymond [4]

Vérimag
Grenoble, France [1]

Abstract

The goal of this paper is to translate (fragments of) the quantified discrete duration calculus QDDC, proposed by P. Pandya, into symbolic acceptors with counters. Acceptors are written in the synchronous programming language Lustre, in order to allow available symbolic verification tools (model-checkers, abstract interpreters) to be applied to properties expressed in QDDC. We show that important constructs of QDDC need non-deterministic acceptors, in order to be translated with a bounded number of counters, and an expressive fragment of the logic is identified and translated. Then, we consider a more restricted fragment, which only needs deterministic acceptors.

Key words: Duration Calculus, QDDC, synchronous observers, non-deterministic observers, Lustre

1 Introduction

The classical way for showing the decidability of a temporal logic (e.g.,[VW86,BVW94]) is to associate with each formula of the logic a finite automaton which accepts exactly the models of the formula. This approach allows also the study of the theoretical complexity of the decision problem. Moreover, the resulting automata can be used for model-checking programs: in order to verify that a program satisfies a formula, one can show the emptiness of the language recognized by the synchronous product of the program (considered as a finite transition system) with the acceptor of the negation of the formula (used as a property checker).

[1] Verimag is a joint laboratory of Université Joseph Fourier, CNRS and INPG associated with IMAG.

[2] Email:Laure.Gonnord@imag.fr
[3] Email:Nicolas.Halbwachs@imag.fr
[4] Email:Pascal.Raymond@imag.fr

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Now, for practical program verification, this classical approach should be revisited.

First, decision procedures are not always interesting, in practice. On one hand, their complexity is generally prohibitive, and on the other hand, exact verification is generally not achievable: programs are not finite state or are too complex, so they must be abstracted [GL93,CGL94], and their verification is therefore only conservative (i.e., negative results are inconclusive). This means that there is no special reason to limit oneself to logics with finite state acceptors: if we have tools for exact or conservative verification of infinite state systems (e.g., [CH78], [JHR99], ...) they can be applied to the composition of infinite state programs with infinite state property checkers.

Moreover, usual methods translate formulas into explicit automata. While the size of these automata is generally not a problem — since simple formulas produce small automata — it can be useful to get them encoded as symbolic automata, i.e., transition relations or functions over state variables with initial values. On one hand, such a symbolic encoding is more convenient if symbolic model-checking techniques [CBM89,BCM+90] are applied. On the other hand, the extension to infinite state automata is straightforward, just by considering infinite domains for the state variables. A typical example is provided by many versions of the duration calculus, where the formulas involving durations give rise to bounded counters in the acceptors; the possible values of these counters are enumerated in the explicit automata, while they could stay symbolic and be naturally extended to unbounded counters.

The main purpose of this paper is to illustrate the idea of recognizing models of formulas by symbolic automata, extending the expressive power by considering infinite state acceptors, and using the results for conservative verification by means of abstract interpretation [CC77,CH78]. We consider the "Quantified Discrete Duration Calculus" QDDC introduced by P. Pandya [5] in [Pan01,Pan02,CP03].

We restrict ourselves to safety properties, which are those that can be verified by examining (an upper-approximation of) the reachable states. We first notice that any formulas of QDDC cannot be recognized by a deterministic symbolic automaton *with a finite number of counters*. We solve this problem by using non-deterministic acceptors, considered as $\forall$-automata [MP87], which can still be used by verification tools: the targeted verification tools proceed by showing that refusing states are not reachable. When the verification is only conservative, it is the only meaningful result we can obtain (since the verdict is inconclusive otherwise). As a consequence, a sequence is considered as accepted if it is accepted by *all* runs of the automaton on it (it is the acceptance criterion of $\forall$-automata). With this idea in mind, we identify a significant fragment of QDDC which can be easily translated into symbolic acceptors.

[5] see also `www.tcs.tifr.res.in/~pandya/dcvalid.html`.

Non-deterministic acceptors are convenient for verification, but cannot be used in other contexts, like property simulation and program testing. This is why we examine also a more restricted fragment of the logic which can be recognized by deterministic acceptors.

The paper is organized as follows: in Section 2 we recall the syntax and semantics of QDDC. In Section 3 we define the symbolic acceptors we consider, which will be described in the language Lustre [HCRP91], and we give (Section 4) the intuition of the translation we have in mind. Sections 5 and 6 describe respectively the non-deterministic fragment and the deterministic one, together with the translation into symbolic acceptors and some examples.

2 The logic QDDC

The "Quantified Discrete Duration Calculus" was proposed by P. Pandya in [Pan01,Pan02,CP03]. Let use briefly recall its syntax and semantics:

Syntax:

Let $Prop(\ni p)$ be a finite set of propositional symbols. The set of propositional formulas is classically defined by

$$P ::= \ 0 \mid 1 \mid p \mid \neg P \mid P \wedge P \mid P \vee P$$

Formulas of QDDC are inductively defined by:

$$D ::= \ \lceil P \rceil^0 \mid \lceil\lceil P \rceil \mid \eta \ op \ c \mid \Sigma P \ op \ c \mid D_1 \wedge D_2 \mid \neg D \mid \exists p \, D \mid D_1 \frown D_2$$

where $op \in \{\leq, <, =, >, \geq\}$ and $c \in \mathbb{N}$.

Semantics:

Models of propositions are *states*, i.e., functions from propositional symbols to $\{true, false\}$. The meaning of propositions is standard. Models of formulas are finite sequences of states ($\sigma = \sigma_0\sigma_1 \ldots \sigma_n$). The meaning of a formula is first defined on intervals of such sequences: an interval $\sigma[b, e] = \sigma_b\sigma_{b+1} \ldots \sigma_e$ is defined by a pair (b, e) of indices, such that $0 \leq b \leq e \leq n$:

$\sigma[b, e] \models \lceil P \rceil^0$	iff	$b = e$ and $\sigma_b \models P$
$\sigma[b, e] \models \lceil\lceil P \rceil$	iff	$b < e$ and $\forall i, b \leq i < e, \sigma_i \models P$
$\sigma[b, e] \models \eta \ op \ c$	iff	$(e - b) \ op \ c$
$\sigma[b, e] \models \Sigma P \ op \ c$	iff	$Card\{i \mid b \leq i < e, \sigma_i \models P\} \ op \ c$
$\sigma[b, e] \models \exists p \, D$	iff	$\exists \sigma', \sigma'[b, e] \models D$, and $\forall i, \forall q \neq p, \sigma_i(q) = \sigma'_i(q)$
$\sigma[b, e] \models D_1 \frown D_2$	iff	$\exists i, b \leq i \leq e$, $\sigma[b, i] \models D_1$ and $\sigma[i, e] \models D_2$

The semantics of $D_1 \wedge D_2$ and of $\neg D$ is as usual. An interval satisfies $\lceil P \rceil^0$ if it is reduced to a state which satisfies P; an interval satisfies $\lceil\lceil P \rceil$ if all of its states (except the last one) satisfy P; an interval satisfies $\eta \ op \ c$ if its length η satisfies the relation op with c; an interval satisfies $\Sigma P \ op \ c$ if the number of its states which satisfy P satisfies the specified relation; an interval $\sigma[b, e]$ satisfies $\exists p \, D$ if each of its states σ_i can be changed into a state σ'_i which differs

only in the value of p, in such a way that $\sigma'[b,e]$ satisfies D; finally, an interval satisfies $D_1 \frown D_2$ if it can be split into two intervals (sharing a junction state) respectively satisfying D_1 and D_2.

Finally, a finite sequence $\sigma = \sigma_0\sigma_1 \ldots \sigma_n$ satisfies a formula D (noted $\sigma \models D$) if and only if $\sigma[0,n] \models D$.

Derived operators:

As usual, some useful derived operators are proposed:

- Usual Boolean operators: *true*, *false* $\vee$, $\Rightarrow$, $\Leftrightarrow$,...
- $[\![P]\!] \stackrel{\text{def}}{\equiv} [\![P]\!] \frown [P]^0$ (right closure of the interval)
- $\Diamond D \stackrel{\text{def}}{\equiv} \mathit{true} \frown D \frown \mathit{true}$ (eventually D)
- $\Box D \stackrel{\text{def}}{\equiv} \neg\Diamond\neg D$ (always D)
- $P_1 \xrightarrow{c} P_2 \stackrel{\text{def}}{\equiv} \neg\Diamond\left(([\![P_1]\!] \wedge \eta \geq c) \frown [\neg P_2]^0\right)$ (whenever P_1 has been continuously true during c steps, P_2 is true).

This logic was proven decidable in [Pan01]. Here, since we don't bother about decidability, we can also consider an extended version of the logic by allowing:

- the constant c to be symbolic (parameter) in formulas like $\eta \; op \; c$, $\Sigma P \; op \; c$, $P_1 \xrightarrow{c} P_2$
- the propositions be extended by conditions on parameters (e.g., $c_1 \geq c_2$).

3 Symbolic automata

Our goal is to define symbolic acceptors of formulas. Symbolic acceptors are special cases of symbolic automata, that we define precisely now.

If V is a finite set of typed variables, let Val_V be the set of valuations of variables in V, i.e., the set of functions from variables to their respective sets of values. Let $\mathcal{S}$ (state variables), $\mathcal{I}$ (input variables), and $\mathcal{O}$ (output variables) be three disjoint sets of variables.

A symbolic automaton on $\mathcal{S}+\mathcal{I}+\mathcal{O}$ is given by an initial state $s_{init} \in \mathit{Val}_{\mathcal{S}}$, and two functions $\mathit{next} : \mathit{Val}_{\mathcal{S}} \times \mathit{Val}_{\mathcal{I}} \mapsto \mathit{Val}_{\mathcal{S}}$ (transition function) and $\mathit{out} : \mathit{Val}_{\mathcal{S}} \times \mathit{Val}_{\mathcal{I}} \mapsto \mathit{Val}_{\mathcal{O}}$ (output function). A *run* of the automaton is a sequence $(s_0, i_0, o_0), (s_1, i_1, o_1), \ldots, (s_n, i_n, o_n)$ of triples from $\mathit{Val}_{\mathcal{S}} \times \mathit{Val}_{\mathcal{I}} \times \mathit{Val}_{\mathcal{O}}$, such that $s_0 = s_{init}$ and for each $\ell, s_{\ell+1} = \mathit{next}(s_\ell, i_\ell)$, and $o_\ell = \mathit{out}(s_\ell, i_\ell)$. Notice that we consider deterministic automata, in the sense that a run is completely determined by the sequence of inputs $i_0, i_1, \ldots, i_n$.

To emphasize the fact that states as well as transition and output functions are given symbolically, we will describe these automata using the syntax of Lustre [HCRP91]:

Each output variable, say x, will be described by an equation "x = exp" where the expression "exp" is made of constants, input variables, usual arithmetic, Boolean, conditional operators, and references to *previous values* of variables (noted "pre(y)"). Since the previous value of a variable is undefined at the very first step (initial state), expressions can be given an initial value: "e0 -> e1" has initially the value of "e0", and then the value of "e1" forever. With respect to the model of symbolic automata defined above, previous variables are state variables and the first argument of each "->" operator contributes to the definition of the initial state. Here are some examples of definitions that we will use in the rest of the paper (other examples can be found in the appendix):

- Let us define an output after_p which is true if p is true or has been true in the past:

  after_p = p or (false -> pre(after_p))

- Now, we want the output nb_q_since_p to be the number of times q has been true since the last time p was true:

```
nb_q_since_p = if p then (if q then 1 else 0)
               else if after_p then (pre(nb_q_since_p)) + (if q then 1 else 0)
               else 0
```

We will often use functional versions of these programs, thus writing after(p) and nb_since(q,p), in order to apply them to various arguments (this is allowed by the notion of *node* in Lustre).

Now, a symbolic acceptor is a symbolic automaton with only one, Boolean, output variable, say a. In general, input variables are Boolean, and represent the values of atomic propositions. An acceptor of a formula D will be such that $(i_0, i_1, \ldots, i_n) \models D$ if and only if the output sequence $(a_0, a_1, \ldots, a_n)$ returned by the automaton in response to inputs $(i_0, i_1, \ldots, i_n)$, is such that $a_n = true$.

4 Recognizing models of formulas with symbolic acceptors

In this section, we illustrate the kind of translation we want to perform, and the way we intend to use it. With each formula we will associate an acceptor, taking as inputs the values of the atomic propositions, possibly the numerical parameters when the extended version of the logic is considered, and an additional Boolean input, say b, which is true at the beginning of the interval of interest. The output of the acceptor is true whenever the formula is satisfied by the interval elapsed since the last step when b was true (by convention, it is also true before the first occurrence of b). We first consider some examples of such translations.

4.1 Some examples

(i) Let us consider the formula $\lceil\lceil p \rceil\rceil$. The acceptor output must be true before the first occurrence of b, take the value of p whenever b is true, and remain true as long as p is true:

```
a = not after(b) or (if b then p else (p and pre(a)))
```

(ii) The formula $D = p \xrightarrow{c} q$ states that q is true whenever p has been true during c steps. Intuitively, an acceptor of this formula should manage a counter (an integer state variable), say x, which is set to 0 when p is false, incremented whenever p is true, and reset when b is true (beginning of the interval). Then the output becomes false when the counter is greater or equal to c and q does not hold; it can only come back to true when a new interval starts:

```
a = not after(b) or ((b or (true -> pre(a)) and (age_p < c or q)));
age_p = if p then (if b then 1 else (0 -> pre(age_p)+1)) else 0;
```

An important remark is that, in this automaton, the counter is an integer variable, and that the delay c can be symbolic. To expand it into an explicit automaton would require the value of c to be fixed and known, and the counter to be expanded into c explicit states. But if we use a verification tool dealing with numbers, the automaton can remain symbolic, and we can even try to prove properties parameterized by c.

(iii) Now, consider the formula $\Box((\eta > c) \Rightarrow (\Sigma p \geq d))$ stating that p occurs at least d times in any interval longer that c. We can use counters, like above, to measure the length of an interval and count the number of occurrences of p. The problem is that fresh counters should be created, at least after each occurrence of p (since one can easily see that each step after an occurrence of p can start a "worst case" interval for the formula), and these counters should be managed until the counter of p's occurrences reaches the value d. As a consequence, the standard approach would need d pairs of counters, and would not work if d is symbolic.

To solve this problem, for this kind of formulas, we will use *non deterministic* acceptors. Instead of using an unbounded number of counters, such an acceptor freely chooses to start an observation (in the specific case above, this choice could be restricted to the initial step and the steps after each occurrence of p, but this restriction is useless for verification). Now the semantics is that the formula is satisfied if the acceptor accepts the input *whatever be* its non deterministic choices (it can be seen as a $\forall$-automaton [MP87]).

4.2 Non deterministic acceptors

Our symbolic acceptors are deterministic, but we can use them as non-deterministic acceptors by adding auxiliary inputs: the set $\mathcal{I}$ of inputs is

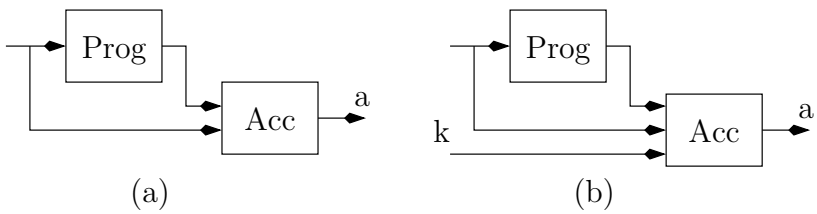

Fig. 1. Using acceptors in program verification

split into $\mathcal{J} + \mathcal{K}$, where variables in $\mathcal{J}$ are classical inputs, while those in $\mathcal{K}$ are additional Boolean inputs called "oracles". Now, an input sequence $(j_0, j_1, \ldots, j_n)$ $(j_\ell \in Val_{\mathcal{J}})$ will be accepted if and only if, for any sequence $(k_0, k_1, \ldots, k_n)$ $(k_\ell \in Val_{\mathcal{K}})$, the output sequence $(a_0, a_1, \ldots, a_n)$ returned by the automaton in response to inputs $([j_0, k_0], [j_1, k_1], \ldots, [j_n, k_n])$, is such that $a_n = true$.

Coming back to the previous example, an acceptor of the formula $\Box((\eta > c) \Rightarrow (\Sigma p \geq d))$ is given by the following code, where k is an oracle:

```
length = nb_since(true, k);
nb_p = nb_since(p, k);
a = not after(b) or b or ((true -> pre(a)) and (length < c or nb_p ≥ d));
```

Of course, in the finite state case, deterministic acceptors have the same expressive power as non deterministic ones, but if we want to keep them symbolic, non-deterministic automata cannot be determinized. As a consequence, while deterministic acceptors can easily be complemented (simply by negating their output), it is not the case of non-deterministic ones.

4.3 Use in verification

The verification tools we have in mind (e.g., those dedicated to Lustre programs: Lesar [RHR91] or NBac [JHR99]) are specialized for safety properties: they consider the synchronous composition of the program and the acceptor of the property (as in Fig. 1.a), and explore the reachable states set (or an upper approximation of this reachable set) of this composition to show that bad states cannot be reached. In other words, they try to show that, *whatever be* the inputs, the output is always true.

In this framework, non deterministic acceptors can easily be used: oracles are just additional free inputs (Fig. 1.b), and the verification tools will work as before *as long as oracles are universally quantified.*

Example: Let us consider the following QDDC formula, which is obviously a tautology: $\left(p \xrightarrow{c} q \wedge d \geq c\right) \Rightarrow (p \xrightarrow{d} q)$
A way of proving this property is to submit the following acceptor — where we use the acceptor of $p \xrightarrow{c} q$ given in Section 4.1, and where the oracle b,

represents the beginning of an interval of interest — to a verification tool (here, there is no program to verify), with the goal of showing that the output is always true:

```
a1 = not after(b) or ((b or (true -> pre(a1)) and (age_p < c or q)));
a2 = not after(b) or ((b or (true -> pre(a2)) and (age_p < d or q)));
age_p = if p then (if b then 1 else (0 -> pre(age_p)+1)) else 0;
a = not(a1 and d≥c) or a2;
```

As a matter of fact, both Lesar and NBac prove it quite instantly, in spite of the fact that the property is parameterized by c and d. *(end of example)*

5 A fragment recognizable by non-deterministic acceptors

In this section, we present a fragment of QDDC for which we can build non deterministic acceptors, and use these acceptors in program verification as shown in the previous section. A said before, oracles can only be universally quantified. Now, in QDDC, there are two constructs which would need existentially quantified oracles, namely $\exists p\, D$ and $D_1 \frown D_2$. As a consequence, the fragment will be restricted to negations of formulas containing such constructs.

5.1 Syntax

The propositions are as before: $P ::= \ 0 \mid 1 \mid p \mid \neg P \mid P \wedge P \mid P \vee P$
Then we need a first intermediate class of formulas, which can be complemented without raising translation problems:

$$N ::= \ \lceil P \rceil^0 \mid \lceil\lceil P \rceil \mid \eta \; op \; c \mid \Sigma P \; op \; c \mid \neg N$$

with $op \in \{\leq, <, =, >, \geq\}$ and c being an integer constant or a parameter. As second class of formulas will give rise to existentially quantified oracles:

$$E ::= \ N \mid E_1 \vee E_2 \mid E_1 \wedge E_2 \mid \exists p\, E \mid E_1 \frown E_2$$

Finally, the properties are: $C ::= \neg E$

5.2 Translation into non-deterministic acceptors

Our goal is to associate with each formula C an acceptor $\mathcal{A}_C$, which will take as inputs $\mathcal{I}$ the sequences $\mathcal{J}$ of values of the propositional formulas, a Boolean b indicating the beginning of the interval of interest, and a set of oracles $\mathcal{K}$. The output is considered as a function $a = \mathcal{A}_C(b, \mathcal{J}, \mathcal{K})$. It is important to assume that the acceptor is triggered only once, since it is not reentrant. So the oracles used to start formulas will be transformed into *starters*: a starter is a Boolean which is true only once.

Propositions:
We first define (in the opposing table) the acceptors $\mathcal{A}_P$ for propositions, which return *true* at step ℓ if the proposition is true at step ℓ.

P	$\mathcal{A}_P(\mathcal{I})$
p	p
$\neg P'$	not $\mathcal{A}_{P'}(\mathcal{I})$
$P_1 \wedge P_2$	$\mathcal{A}_{P_1}(\mathcal{I})$ and $\mathcal{A}_{P_2}(\mathcal{I})$
$P_1 \vee P_2$	$\mathcal{A}_{P_1}(\mathcal{I})$ or $\mathcal{A}_{P_2}(\mathcal{I})$

Formulas N :
We first introduce some useful operators (they are all given in the appendix):

- always_since(P,Q) returns *true* if P has been always true since the last occurrence of Q;
- nb_since(P,Q) returns the number of occurrences of P since the last occurrence of Q;
- strict_after(P) returns *true* if P has been true in the strict past.
- starter(b) transforms an oracle b into a starter:
  starter(b) = b and not strict_after(b).

Then the acceptor $\mathcal{A}_N$ is defined inductively by the following table:

N	$\mathcal{A}_N(b,\mathcal{I})$
$\lceil P \rceil^0$	$\mathcal{A}_P(\mathcal{I})$ and b
$\lceil\lceil P \rceil$	strict_after(b) and pre(always_since($\mathcal{A}_P(\mathcal{I})$,b))
$\eta\ op\ c$	nb_since(true,b) op c
$\Sigma P\ op\ c$	nb_since($\mathcal{A}_P(\mathcal{I})$,b) op c
$\neg N'$	not $\mathcal{A}_{N'}(b,\mathcal{I})$

Formulas E :
Acceptors of formulas E distinguish between real inputs ($\mathcal{J}$) and oracles ($\mathcal{K}$).

$$\mathcal{A}_N(b,\mathcal{J},\mathcal{K}) = \mathcal{A}_N(b,\mathcal{J}\cup\mathcal{K})$$
$$\mathcal{A}_{E_1\vee E_2}(b,\mathcal{J},\mathcal{K}) = \mathcal{A}_{E_1}(b,\mathcal{J},\mathcal{K}) \vee \mathcal{A}_{E_2}(b,\mathcal{J},\mathcal{K})$$
$$\mathcal{A}_{E_1\wedge E_2}(b,\mathcal{J},\mathcal{K}) = \mathcal{A}_{E_1}(b,\mathcal{J},\mathcal{K}) \wedge \mathcal{A}_{E_2}(b,\mathcal{J},\mathcal{K})$$
$$\mathcal{A}_{\exists p\, E}(b,\mathcal{J},\mathcal{K}\uplus\{p\}) = \mathcal{A}_E(b,\mathcal{J}\uplus\{p\},\mathcal{K})$$
$$\mathcal{A}_{E_1\frown E_2}(b,\mathcal{J},\mathcal{K}\uplus\{b'\}) = \mathcal{A}_{E_2}(\mathsf{starter}(b') \wedge \mathcal{A}_{E_1}(b,\mathcal{J},\mathcal{K}),\mathcal{J}\uplus\{b'\},\mathcal{K})$$

Formulas C :
Finally, the acceptor of a formula $C = \neg E$ is just: not $\mathcal{A}_E(b,\mathcal{J},\mathcal{K})$.

Top-level acceptance:
Recall that the satisfaction of a QDDC formula by a sequence $\sigma = \sigma_0\sigma_1\ldots\sigma_n$ is defined by $\sigma \models D$ iff $\sigma[0,n] \models D$
So, the top-level acceptor of a formula C is $\mathcal{A}_C$(true -> false, $\mathcal{J},\mathcal{K}$), since the "starter" true -> false is true only at the first step.

In [DG03], this translation is proven correct, i.e.:

$\sigma \models D, |\sigma| = n \Leftrightarrow \forall \widetilde{\mathcal{K}}, \mathcal{A}_D(\widetilde{b}, \widetilde{\mathcal{J}}, \widetilde{\mathcal{K}}) = true^n$, where

- $\widetilde{b} = true.(false)^{n-1}$, the flow being true only at the initial instant;
- $\widetilde{\mathcal{J}}$ are the values imposed by σ to the variables of the set $\mathcal{J}$.

5.3 Examples

Let us first consider the formula $\neg(\lceil\!\lceil p \rceil\!\rceil \frown \lceil\!\lceil q \rceil\!\rceil)$. From our rules, we get

- $\mathcal{A}_{\lceil\!\lceil p \rceil\!\rceil \frown \lceil\!\lceil q \rceil\!\rceil}(b, p, q, b') = \mathcal{A}_{\lceil\!\lceil q \rceil\!\rceil}(\mathsf{starter}(b') \wedge \mathcal{A}_{\lceil\!\lceil p \rceil\!\rceil}(b, p, q), p, q, b')$
- $\mathcal{A}_{\lceil\!\lceil p \rceil\!\rceil}(b, p, q) =$ strict_after(b) and pre(always_since(p,b))
- $\mathcal{A}_{\lceil\!\lceil q \rceil\!\rceil}(b'', p, q, b') =$ strict_after(b") and pre(always_since(q,b"))
  (with $b'' = \mathsf{starter}(b') \wedge \mathcal{A}_{\lceil\!\lceil p \rceil\!\rceil}(b, p, q)$)

So (introducing "local variables" e and b" for clarity) we can translate the formula into

```
a = not e;
e = strict_after(b") and pre(always_since(q,b"));
b"= starter(b') and strict_after(b) and pre(always_since(p,b));
```

As an other example, we have already considered the formula $\Box((\eta > c) \Rightarrow (\Sigma p \geq d))$ motivating the introduction of non-deterministic acceptors. Translated into basic QDDC, this formula becomes

$$\neg(true \frown (\eta > c \wedge \Sigma p < d) \frown true)$$

and falls into our fragment. Its translation (after simplification) gives

```
a = not e;
e = starter(b') and nb_since(true,starter(b)) > c and
        nb_since(p,starter(b)) < d;
```

where b' is an oracle used to start the counters nb_since.

Finally, the whole example of "mine pump" given in [Pan01] falls into our fragment, showing that this fragment is of practical interest.

6 A deterministic fragment

An recognized advantage of expressing safety properties by means of acceptors written in a programming language (like Lustre), is that these acceptors can be executed: they can be tested with various input sequences to check that they express well the initial intuition; they can also be run with the program, for testing or "runtime verification" [HR02] purposes. Now, this is not the case of our non-deterministic acceptors, which can only be used in connection with a verification tool. This is why it is also interesting to look for a fragment of the logic that be translated into *deterministic* acceptors. This is the purpose of this section.

First, we have to come back to the sources of non-determinism in QDDC. If we forget about the construct $\exists pD$ (which is completely non deterministic), we are left with the "chop" construct, whose non-determinism comes from the arbitrary split of the considered interval. Now, in real specification, it appears that the "chop" is very often used deterministically: for instance in $\lceil\!\lceil P\rceil\!\rceil^\frown\lceil Q\rceil^0$, the interval is split into itself and its last state. The basic idea of our deterministic fragment is to replace the "chop" by a deterministic restriction, called "then", where D_1 *then* D_2 means that the interval consists of a *maximal* interval satisfying D_1 followed by an interval satisfying D_2. Now, the notion of maximal interval can only be handled for some kind of formulas for D_1: we will restrict ourselves to formulas which can only change from true to false when the interval increases. We first define these formulas, noted G.

6.1 Syntax and semantics

Propositions P are as before.

The fragment G:

As announced, formulas G can only become false when the time passes:

$$G ::= \mathit{begin}(P) \mid \lceil\!\lceil P\rceil\!\rceil \mid \eta \leq c \mid \Sigma P \leq c \mid \mathit{age}(P) \leq c \mid G_1 \wedge G_2 \mid G_1 \vee G_2$$

The only new constructs are $\mathit{begin}(P)$ — which tells that the first state of the interval satisfies P — and $\mathit{age}(P) \leq c$ — telling that P has been continuously true for less than c steps:

$$\begin{array}{lll} \sigma[b,e] \models \mathit{begin}(P) & \text{iff} & \sigma_b \models P \\ \sigma[b,e] \models \mathit{age}(P) \leq c & \text{iff} & e - m \leq c \\ \text{where } m = \begin{cases} \mathit{max}\{i \mid b \leq i \leq e,\ \sigma_i \models \neg P\} & \text{if } \neg P \text{ occurred in } [b,e] \\ b-1 & \text{otherwise} \end{cases} \end{array}$$

The full fragment:

The formulas of our deterministic fragment are called F:

$$F ::= G \mid \mathit{end}(P) \mid G \mathit{\ then\ } F \mid F_1 \wedge F_2 \mid \neg F$$

The new operators are defined as follows:

$$\begin{array}{lll} \sigma[b,e] \models \mathit{end}(P) & \text{iff} & \sigma_e \models P \\ \sigma[b,e] \models G \mathit{\ then\ } F & \text{iff} & \exists m, b \leq m < e,\ \sigma[b,m] \models G,\ \sigma[b,m+1] \not\models G, \\ & & \sigma[m+1,e] \models F \end{array}$$

6.2 Translation into deterministic acceptors

The acceptors $\mathcal{A}_P$ of propositions are as before. For formulas G, we only define the acceptors for $\mathit{begin}(P)$ and $\mathit{age}(P) \leq c$, the other cases are as before:

$\mathcal{A}_{begin(P)}(b,\mathcal{I}) =$ after(b and $\mathcal{A}_P(\mathcal{I})$)
$\mathcal{A}_{age(P)\leq c}(b,\mathcal{I}) =$ age($\mathcal{A}_P(\mathcal{I})$,b) $\leq$ c

where age(p,b) stands for:

```
age_p_b =  if after(b) and p then (0 ->pre(age_p_b)) +1
           else 0
```

The acceptor of a formula F is inductively defined by the opposing table, where "first(b,p)" stands for "after(b) and p and pre(always_since(not p, b))" and is true at the first occurrence of p following b.

F	$\mathcal{A}_F(b,\mathcal{I})$
$end(P)$	after(b) and $\mathcal{A}_P(\mathcal{I})$
G *then* F	$\mathcal{A}_F$(first(not $\mathcal{A}_G(b,\mathcal{I})$, $\mathcal{I}$))
$F_1 \wedge F_2$	$\mathcal{A}_{F_1}(b,\mathcal{I})$ and $\mathcal{A}_{F_2}(b,\mathcal{I})$
$F_1 \vee F_2$	$\mathcal{A}_{F_1}(b,\mathcal{I})$ or $\mathcal{A}_{F_2}(b,\mathcal{I})$
$\neg F$	not $\mathcal{A}_F(b,\mathcal{I})$

6.3 Examples

Let us show some examples of formulas using the "chop" operator, and which fall into the deterministic fragment:

- $\lceil\lceil P\rceil\frown\lceil Q\rceil^0$ is equivalent to $\lceil\lceil P\rceil \wedge\ end(Q)$
- $p \xrightarrow{c} q$ can be rewritten into $\Box((\ age(p) \leq c)\vee q)$, meaning that the formula is satisfied if and only the acceptor "age(p, true->false)≤c or q" always returns *true*.

7 Conclusion

This work does not intend to be a theoretical one. Of course, as soon as we consider automata extended with unbounded counters, the expressive power is maximal, and we can encode any kind of properties involving infinite memory. However, such an encoding is of little practical interest.

We wanted to connect the way synchronous programs are often specified by means of synchronous observers, with formalisms better known in the TCS community. This is why we chose to translate a logic belonging to a wide community (the duration calculus family [CHR91]), or at least useful fragments of it, into our observers.

Our initial inspiration was [Ray96], which is a very efficient translation of rational expressions into Lustre acceptors. It appeared soon that the same technique cannot be applied to automata with counters: the linear cost of the translation performed in [Ray96] was obtained thanks to reentrant acceptors (the same component of the acceptor can be activated several times while running). This is not possible for extended automata. This is why the present work diverges significantly from [Ray96], particularly because we had to consider non-deterministic automata with oracles.

We wanted also to defend the idea that, in practice, there is no need to

restrict oneself to decidable logics, nor to finite state acceptors. Abstraction and approximation are routinely used to deal with programs, and there is no reason not to apply them also to properties. In particular, automata extended with *counters* are extremely powerful and useful, and more and more techniques are proposed to handle them (e.g., [BW94], [CJ98], [JHR99], [FS00], ...).

Acknowledgements: We are indebted to Paritosh Pandya, for having initiated this work, and for many helpful discussions.

References

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Fifth IEEE Symposium on Logic in Computer Science, Philadelphia*, pages 428–439, June 1990.

[BVW94] O. Bernholtz, M. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. In D. Dill, editor, *6th International Conference on Computer Aided Verification, CAV'94*, Stanford, June 1994.

[BW94] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *CAV'94*, Stanford (Ca.), 1994. LNCS 818, Springer Verlag.

[CBM89] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using boolean functional vectors. In L. J. M. Claesen, editor, *Formal VLSI Correctness Verification*. North-Holland, November 1989.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages, POPL'77*, Los Angeles, January 1977.

[CGL94] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM TOPLAS*, 16(5):1512–1542, 1994.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th ACM Symposium on Principles of Programming Languages, POPL'78*, Tucson (Arizona), January 1978.

[CHR91] Z. Chaochen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5), 1991.

[CJ98] H. Comon and Y. Jurski. Multiple counters automata, safety analysis and presburger arithmetic. In *CAV'98*, Vancouver (B.C.), 1998. LNCS 1427, Springer Verlag.

[CP03] G. Chakravarty and P.K. Pandya. Digitizing interval duration logic. In *CAV 2003*, Boulder, Colorado, July 2003.

[DG03] L. Danthony-Gonnord. Du calcul des durées aux automates symboliques. Master thesis, Universités Paris VI/VII, June 2003. `www-verimag.imag.fr/~danthony/papers/rapport_dea.ps`.

[FS00] A. Finkel and G. Sutre. An algorithm constructing the semilinear post* for 2-dim reset/transfer vass. In *25th Int. Symp. Math. Found. Comp. Sci. (MFCS'2000)*, Bratislava, Slovakia, August 2000. LNCS 1893, Springer Verlag.

[GL93] S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. In *Fifth Conference on Computer-Aided Verification, CAV'93*, Elounda (Greece), July 1993. LNCS 697, Springer Verlag.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HR02] K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, Grenoble, France, April 2002.

[JHR99] B. Jeannet, N. Halbwachs, and P. Raymond. Dynamic partitioning in analyses of numerical properties. In A. Cortesi and G. Filé, editors, *Static Analysis Symposium, SAS'99*, Venice (Italy), September 1999. LNCS 1694, Springer Verlag.

[MP87] Z. Manna and A. Pnueli. Specification and verification of concurrent programs by $\forall$-automata. In *14th ACM Symposium on Principles of Programming Languages, POPL'87*, Munchen, January 1987.

[Pan01] P.K. Pandya. Specifying and deciding quantified discrete-time duration calculus formulae using dcvalid. In *Real-Time Tools, RTTOOLS'2001*, Aalborg, August 2001.

[Pan02] P.K. Pandya. Interval duration logic: Expressiveness and decidability. In *Workshop on Theory and Practice of Timed Systems TPTS'2002*, Grenoble, April 2002.

[Ray96] P. Raymond. Recognizing regular expressions by means of dataflows networks. In *23rd International Colloquium on Automata, Languages, and Programming, (ICALP'96)* Paderborn, Germany. LNCS 1099, Springer Verlag, July 1996.

[RHR91] C. Ratel, N. Halbwachs, and P. Raymond. Programming and verifying critical systems by means of the synchronous data-flow programming language Lustre. In *ACM-SIGSOFT'91 Conference on Software for Critical Systems*, New Orleans, December 1991.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Symposium on Logic in Computer Science*, June 1986.

Appendix: Usual temporal operators in Lustre

after(p)
returns true after or at the first occurrence of p

```
after(p) =
    if p then true
    else (false -> pre(after(p)));
```

starter(b)
transforms b *into a single occurence starter*

```
starter(b) =
    b and not strict_after(b)
```

always_since(p,b)
returns true if p *has been continuoulsy true since the last occurrence of* b*; returns also true before the first occurrence of* b

```
always_since(p,b) =
    if b then p
    else if after(b) then
        (p and pre(always_since(p,b)))
    else true
```

age(p,b)
counts the time elapsed since the latest of the occurrence of b *(assumed to be unique) and the last occurrence of* not p

```
age(p,b) = if after(b) and p then
                (0 ->pre(age(p,b))) + 1
           else 0
```

strict_after(p)
returns true strictly after the first occurrence of p

```
strict_after(p) =
    if (false ->pre(p)) then true
    else (false -> pre(strict_after(p)));
```

first(p,b)
returns true at the first occurrence of p *following an occurrence of* b

```
first(p,b)= p and never_p;
never_p =
    if b then true
    else if (false -> pre(p))then false
    else (false -> pre(never_p))
```

nb_since(p,b)
counts the number of occurrences of p *since the last occurrence of* b*; returns 0 before the first occurrence of* b

```
nb_since(p,b) =
    if b then (if p then 1 else 0)
    else if after(b) then
        (pre(nb_since(p,b))) +
            if p then 1 else 0
    else 0
```