ntccMC: A bounded-time model checker for ntcc

Mauricio Toro
Universidad Eafit

October 4, 2018

Abstract

A disadvantage of concurrent constraint programming (ccp) calculi is that it is difficult to exploit its declarative point of view because of the lack of automated verification tools available. This limits the applicability of the verification techniques to small problems. As an example, a model checker for a non-deterministic timed concurrent constraint (ntcc) will be useful not only for music systems, but for many other fields on which ccp has been proven successful (e.g., system biology and security systems). We claim for the urgent need of a model checker for ntcc. First, because ntcc has been widely used to model reactive system and verify properties about them, but the verification had to be done by hand. Second, because there are not many frameworks to formally model and verify music interaction systems and ntcc has been proved to be successful in that field. In this paper we propose a model checker for ntcc.

1 Introduction

There has been a rapidly growing interest for *concurrent constraint programming* (ccp) [27] models in the last decade. Ccp processes can be analyzed from both a behavioral and declarative point of view, making them suitable for simulation and for verification of properties. Some programming languages have also been developed following the concepts of ccp. As an example *Mozart/Oz* [23] is a multiparadigm programming language inspired in the ccp paradigm.

Several interpreters for ccp calculi have been developed. As an example, *Ntcrt* [39] offers the possibility to execute ntcc models. There is also an interpreter for *Temporal Concurrent Constraint Programming* (tcc) [30]. Another example is the framework [10] that allows to execute and verify *Default Timed Concurrent Constraint* (tccp) [9] models.

Ccp calculi belongs to a family of formalisms named process calculi. Process calculi has been applied to the modeling of interactive music systems [38, 37, 43, 48, 36, 47, 2, 44, 39, 15, 34, 32, 33, 35, 4, 42, 49, 40, 41, 31] and ecological systems [45, 17, 46, 18]. In addition, research on algorithms [16, 13, 22] and software engineering [21] also contributes to this field.

The interest to verify musical models formally has also grown rapidly during the last decade. As an example, a formalism to define declaratively partial orders of musical processes and audio is *Tempo* [20]. There is a real-time capable interpreter and an automatic verification tool for *Tempo*. Unfortunately, *Tempo* does not allow to express simultaneity nor weak time-outs (e.g., perform an action if a condition cannot be deduced), critical in several music interaction systems.

To handle complex synchronization patterns and to predict the behavior of interactive scenarios, *Hierarchical Time Stream Petri Nets* (HTSPN) [29] has been used to formalize *interactive music scores* [1] and multimedia streaming systems. There are several model checkers for time Petri nets such as *Romeo* [11] and TAPAAL [8]. Unfortunately, none of them are suitable for hierarchical nor time stream extensions of Petri nets. In fact, it was proved in [6] that HTSPN cannot be translated into time Petri nets.

Another concurrency formalism widely used in the verification of music interactive systems is the *Non-deterministic Timed Concurrent Constraint* calculus (ntcc). Ntcc is an extension of ccp with time, asynchrony and non-determinism. Ntcc has been used since its beginnings to prove properties of music systems. As an example, it was used to verify properties of a well-known problem of western-african music [24]. The reader may also look to [25] and [26] for other examples of ntcc in the verification of music systems.

A disadvantage of ccp calculi is that it is difficult to exploit its declarative point of view because of the lack of automated verification tools available. This limits the applicability of the verification techniques to small problems. As an example, a model checker for a ntcc will be useful not only for music systems, but for many other fields on which ccp has been proven successful (e.g., system biology and security systems).

We claim for the urgent need of a model checker for ntcc. First, because ntcc has been widely used to model reactive system and verify properties about them, but the verification had to be done by hand. Second, because there are not many frameworks to formally model and verify music interaction systems and ntcc has been proved to be successful in that field.

1.1 Related work

In what follows, we describe some model checkers commonly used for reactive systems and why we cannot exploit them directly to verify ntcc. Afterwards, we explain existing verification (non-automatic) techniques for ntcc and why we cannot implement them straight forward to have an efficient model checker.

1.1.1 Existing model checkers

A well-known model checker for reactive systems is *Spin* [12]. Spin a model checker for the *Promela* language to verify Linear-Time Logic (LTL) [19] properties. *Kronos* [7] and UPPAAL [5] are model checkers for timed automata and

verify LTL properties and continuous-time logic properties. Although Kronos and UPPAAL are often used to verify real-time and reactive systems, they work upon *timed automata* [3]. Timed automata model checking is far more complex and limited than verification of finite state automata. Since `ntcc` is a concurrent formalism for discrete time, such translation would result into one complex time automaton (or several automata synchronized).

There is also a bounded model checker for `tccp`. An option could be to translate `ntcc` to `tccp`. Nonetheless, `ntcc` is finite state whereas `tccp` is infinite state. In addition, their semantics are very different. Therefore, if such translation exists, it might not be expressed compositionally and it might result in an exponential number of `tccp` process to represent a `ntcc` process.

1.1.2 Existing verification techniques for `ntcc`

The *strongest post-condition* in `ntcc` is set of sequences that a process can output interacting with any possible environment. It was proven by Valencia et al. in [51] that the strongest post-condition of a process can be translated into a Büchi automaton. Büchi automata are an extension of finite state automata for infinite input. Such automata accept exactly those runs in which at least one of the infinitely often occurring states is an accepting state.

There are few disadvantages of the Büchi translation to make a model checker based on the classic model checking algorithm [28]. First, complexity is very high because complement and intersection of Büchi automata complexity is non-elementary. The second problem is the specification logic used by `ntcc`. Constraint LTL (CLTL) [14] is an extension to use constraints as formula. We conjecture that although such translation exists for an interesting fragment described in [51], the translation might increase considerably the size of the formula. As an example, consider a formula “eventually there is a store where the constraint c can be deduced”. A translation to LTL could be “eventually we will reach state 1, 2, 3... or n ”. Note that the complexity of the classic algorithm depends exponentially on the size of the formulae.

There is another verification technique for `ntcc`. `Ntcc` is equipped with a inference system to proof if a `ntcc` process satisfies a CLTL property. This approach can provide a semi-automatic (or even automatic) approach for verification. We leave this alternative as future work.

1.2 Contribution

We propose a bounded model checking procedure for `ntcc`. The procedure is based upon an encoding of `ntcc` processes and CLTL formula into deterministic finite state automata. Process Bang represents infinite behavior in `ntcc`, it is analogous to replication the *Pi calculus*[?] or *CSS calculus*[?]. Unfortunately, there is no way, to our knowledge, to represent bang’s infinite behavior into finite state automata, thus only bounded model checking is possible with such approach.

1.3 Structure of the report

In what follows of this report, we present `ntcc` in section 2, the model checking procedure in section 3, the implementation in section 4, applications and examples in section 5, and conclusions and future work in section 6.

2 The Non-deterministic Timed Concurrent Constraint (ntcc) Calculus

In *Concurrent Constraint Programming* (ccp) [27] a system is modeled as a collection of concurrent processes whose interaction behavior is based on the information (represented by constraints) contained in a *global store*. Formally, ccp is based upon the idea of a *constraint system* (CS). A constraint system is composed of a set of (basic) constraints and an entailment relation specifying constraints that can be deduced from others. In this paper we use a *finite domain* CS providing arithmetic relations over natural numbers. As an example, using a finite domain CS, we can deduce $pitch \neq 60$ from the constraints $pitch > 40$ and $pitch < 59$.

Ccp has a disadvantage: In ccp it is not possible to delete nor change information accumulated in the store. For that reason, it is difficult to perceive a notion of discrete time, useful to model reactive systems communicating with an external environment (e.g., sensors and speakers).

Ntcc ([14]) circumvents this limitation by introducing the notion of discrete time as a sequence of time units. At each time unit, a ccp computation takes place, starting with an empty store (or one that has been given some information by the environment). Concurrent constraints agents operate on this store as in the usual ccp model to accumulate information into the store. As opposed to the ccp model, the agents can schedule processes to be run in future time units. In addition, since at the beginning of each time unit a new store is created, information on the value of a variable can change (e.g., it can be forgotten) from one unit to the next. As an example, process **tell** ($pitch_1 = 52$) **||** **when** $48 < pitch_1 < 59$ **do next tell** ($instrument = 1$) adds to the store the constraint $pitch_1 = 52$ and postpones one time unit the constraint $instrument = 1$. A description of ntcc processes can be found at table 1¹.

A simple form of recursion can be defined in terms of basic ntcc agents. Recursion can be defined with the form $q(x) =^{def} P_q$, where q is the process name and P_q is restricted to call q at most once and such call must be within the scope of a *next* (see [50]). The reader should not confuse a *simple definition* with a *recursive definition*; for instance, $Before_{i,j,\Delta} =^{def} \mathbf{tell}(P_i + \Delta < P_j)$ is a simple definition where the values of i, Δ and j are replaced “statically”, like a macro in a programming language. Instead, a recursive definition such as $Clock(v) =^{def} \mathbf{tell}(clock = v) \mathbf{||} \mathbf{next} Clock(v + 1)$ is like a function in a programming language.

¹The agent for asynchrony (*) provided by ntcc is omitted in this paper for simplicity.

Agent	Meaning
tell (c)	Adds c to the current store
when (c) do A	If c holds now, run A now
local (x) in P	Runs P with local variable x
$A \parallel B$	Parallel composition
next A	Runs A at the next time unit
unless (c) next A	Unless c holds now, next run A
$\sum_{i \in I} \text{when } (c_i) \text{ do } P_i$	Chooses P_i such that (c_i) holds
$!P$	Executes P each time unit
$*P$	Eventually executes P

Table 1: Intuitive semantics of the `ntcc` agents.

2.1 Operational Semantics of `ntcc`

`Ntcc` has two type of transitions: internal transitions that occur within a time unit ($\longrightarrow$) and observable transitions that occur from one time unit to the other ($\Longrightarrow$). A rule states that whenever the conditions have been obtained in the course of some derivation, the conclusion is also obtained (see Table 2).

Rule `PAR` states that whenever a process P can evolve to a process P' by performing an internal transition, it can also evolve from $P \parallel Q$ to $P' \parallel Q$. Rules `TELL`, `SUM` and `UNL` are interpreted in a similar way to rule `PAR`. Rule `REP` specifies that process $!P$ produces a copy of P at the current time unit, and then persists in the next time unit. Rule `STR` says that structurally congruent (defined in detail in [14]) configurations have the same reductions. Finally, rule `LOC` explains how x can have a local scope in the sense of local variables in programming languages. Rule `OBS` says that an observable transition from P labeled by (c, d) is obtained by performing a sequence of internal transitions from the initial configuration (P, c) to a final configuration (Q, d) in which no further internal evolution is possible². The residual process R to be executed in the next time unit is equivalent to the process $F(Q)$, where the future function $F : \text{Proc} \rightarrow \text{Proc}$ is a the partial function defined by

$$F(Q) = \begin{cases} \text{skip} & \text{if } Q = \sum_{i \in I} \text{when } c_i \text{ do } Q_i \\ F(Q_1) \parallel F(Q_2) & \text{if } Q = Q_1 \parallel Q_2 \\ (\text{local } x)F(R) & \text{if } Q = (\text{local } x, c)R \\ R & \text{if } Q = \text{next } R \text{ or } Q = \text{unless } c \text{ next } R \end{cases}$$

The process $F(Q)$, defined above, is obtained by removing from Q summations that did not trigger activity within the current time unit and any local information which has been stored in Q . Process **skip** is a short form for an empty summation: a process that does not perform any transition.

²We use c, d in this paper to represent constraint stores.

TELL	$\frac{}{(\mathbf{tell}(c), d) \longrightarrow (\mathbf{skip}, d \wedge c)}$	PAR	$\frac{(P, c) \longrightarrow (P', d)}{(P \parallel Q, c) \longrightarrow (P' \parallel Q, d)}$
REP	$\frac{}{(!P, d) \longrightarrow (P \parallel \mathbf{next} !P, d)}$	STR	$\frac{\gamma_1 \longrightarrow \gamma_2}{\gamma'_1 \longrightarrow \gamma'_2} \text{ if } \gamma_1 \equiv \gamma'_1 \text{ and } \gamma_2 \equiv \gamma'_2$
SUM	$\frac{}{(\sum_{i \in I} \mathbf{when } c_i \text{ do } P_i, d) \longrightarrow (P_j, d)}$		if $d \models c_j, j \in I$
UNL	$\frac{}{(\mathbf{unless } c \text{ next } P, d) \longrightarrow (\mathbf{skip}, d)}$		if $d \models c$
LOC	$\frac{(P, c \wedge \exists_x d) \longrightarrow (P', c')}{((\mathbf{local } x, c)P, d) \longrightarrow ((\mathbf{local } x, c')P', d \wedge \exists_x c')}$		
OBS	$\frac{(P, c) \longrightarrow^* (Q, d) \not\vdash \gamma}{P \xrightarrow{(c, d)} R} \text{ if } R \equiv F(Q)$		
STAR	??		

Table 2: Operational semantics of `ntcc`.

2.2 Denotational Semantics of `ntcc`

The denotation of a `ntcc` process is a set of infinite sequences of stores (i.e., constraints). Each element of the sequence corresponds to each time unit. A function $\llbracket \cdot \rrbracket$ associates such sets to each process (see Table 3). Notation $\exists_x \alpha$ denotes the application of $\exists_x$ to each constraint on α . Set C represents the constraints in the constraint system. Henceforth C^ω, C^* are the set of infinite and finite sequences of constraints in C , respectively. We use α, α' to represent elements of C^ω , β to represent an element of C^* , and $\beta.\alpha$ to represent the concatenation of β and α .

Intuitively, $\llbracket P \rrbracket$ represents the sequences that a process P can output interacting

D1	$\llbracket \mathbf{tell}(c) \rrbracket = \{d.\alpha \mid d \vdash c, \alpha \in C^\omega\}$
D2	$\llbracket \sum_{i \in I} \mathbf{when } c_i \text{ do } P_i \rrbracket = \bigcup_{i \in I} \{d.\alpha \mid d \vdash c_i, d.\alpha \in \llbracket P_i \rrbracket\} \cup \bigcap_{i \in I} \{d.\alpha \mid d \not\vdash c_i, d.\alpha \in C^\omega\}$
D3	$\llbracket P \parallel Q \rrbracket = \llbracket P \rrbracket \cap \llbracket Q \rrbracket$
D4	$\llbracket \mathbf{local } x \text{ in } P \rrbracket = \{\alpha \mid \text{there exists } \alpha' \in \llbracket P \rrbracket \text{ s.t. } \exists_x \alpha = \exists_x \alpha'\}$
D5	$\llbracket \mathbf{next } P \rrbracket = \{d.\alpha \mid d \in C, \alpha \in \llbracket P \rrbracket\}$
D6	$\llbracket \mathbf{unless } c \text{ next } P \rrbracket = \{d.\alpha \mid d \vdash c, \alpha \in C^\omega\} \cup \{d.\alpha \mid d \not\vdash c, \alpha \in \llbracket P \rrbracket\}$
D7	$\llbracket !P \rrbracket = \{\alpha \mid \forall \beta \in C^*, \alpha' \in C^\omega \text{ s.t. } \alpha = \beta.\alpha', \text{ we have } \alpha' \in \llbracket P \rrbracket\}$
D8	$\llbracket *P \rrbracket = \{\beta.\alpha \mid \beta \in C^*, \alpha \in \llbracket P \rrbracket\}$

Table 3: Denotational semantics of `ntcc`.

with any possible environment. As an example, the sequences of $[[\text{tell}(c)]]$ are those whose first element is stronger than c (rule *D1*). Further information on the denotation of ntcc can be found in [14].

The infinite sequences of the denotation of a process can be represented as a Büchi automaton. Büchi automata are an extension of finite state automata for infinite input. Such automata accept exactly those runs in which at least one of the infinitely often occurring states is an accepting state. For simplicity we label the transitions as $c \vdash d$ (i.e., there is a transition for every constraint c stronger than d). The reader may find the encoding of the denotation of ntcc into Büchi in [51]. The encoding also defines a finite set of constraints (i.e., *the relevant constraints of P*) because the alphabet of a Büchi automaton must be finite.

relevant constraint table

2.3 cltl: ntcc's logic

table

2.3.1 cltl to ntcc

table Table from Val03.

3 model checking for ntcc

3.1 ntcc to buchi

Description from Val03

Figure 1: Encoding Ntcc into a büchi automaton.

3.2 ntcc to fsa

The encoding of a ntcc process into a finite state automaton (FSA) is based on the encoding into büchi automaton. The bang cannot be encoded in the same way. Following the same principle of the previous encoding, the bang will be codify as the Kleene star (well known in regular expressions). Such encoding has a problem, it recognizes lengths of any size, which is not the intended meaning of the bang. For that reason, this encoding of ntcc consider a fragment of ntcc without bang; instead, we provide a bounder version $![0, n]P = \dots$. In what follows we will explain in detail the encoding of each process into a FSA.

tell Process **tell**(c) is encoded as a automaton with two states. The first state is an initial state with a transition labeled by d such that $c \models d, c \in S$. The second state is the final state and it has a transition to itself labeled by c such that $c \in S$.

parallel Parallel composition is the intersection of two automata. Unlike Büchi automata, the intersection of two FSA is simple and well-known: it can be coded as the union of the intersection of both automaton. The union can be coded as ??

sum The encoding is given by an automaton that recognizes the following language: ?? To build such automaton, we proceed as follows. ??

bang Process $![0, n]P = \dots$ is first encoded into a next and parallel composition, then the encoding proceeds as it does for parallel and next. The problem to translate $!P$ into $![0, n]P$ is to compute the right value for n . Encoding should be made such that all sequences accepted by the automaton are on length n . How do we do that?

star The encoding of the star is different to the one in the encoding into Büchi. We take the encoding of P and we add arcs from the initial state to itself labeled by c such that $c \in S$.

local For local processes, we also proceed different than for the encoding Büchi. Under a static analysis, we can replace processes local x in P by $P[x/\text{fresh}]$ where fresh is a “fresh variable”. This works because there are no loops, except self-loops, and fresh does constraints with fresh does not appear in such self-loops. This is based on ?? CITE!

next Process $\text{next } P$ can be encoded by adding a new state to the encoding of P . Such state is the new initial state. Such state has a transitions labeled by c such that $c \in S$ to the previous initial state.

unless Encoding of unless proceeds as the encoding into Büchi. We add a new state. Such state is the new initial state. Such state has transitions d such that $d \vdash (\text{not})c \wedge d \in S$ to the previous initial state. It is also intercepted with an automaton that accepts a constraint d such that $d \vdash c \wedge d \in S$ and then accepts anything.

recursion and cells Recursion and cells as encoded as it was described in ??, but we use the bounded bang instead of the bang.

3.3 model checking algorithm

introduction of the LTL model checking algo [28]

Intersection of the fsa of the process and the fsa of the formulae. The model checking algorithm is the following.

1. Translate the process into a FSA.
2. Translate the formulae into a process and then the process into a FSA.
3. Intercept both automata.
4. Check for emptyness.

3.3.1 correctness of ntcc to fsa

Proposition 3.1. *The first n elements in the set of sequences of $[[P]] P \vdash \text{formula}(P)$ iff $P_{FSA} \models_{FSA} \text{formula}(P_{FSA})_{FSA}$. Is there a problem because the complement of star is no longer the complement of bang? Nope*

Proof. We proceed by induction on the structure of P . □

4 implementation

The implementation is based upon x tools: the constraint solving library Gecode to test constraint deduction and equivalence, the Automata Standard Library to manage Finite State Automata efficiently, and Flex and Bison to construct the parser.

4.1 Architecture

We use Object oriented programming to build a reusable and extensible architecture to represent ntcc processes and cltl formula.

4.2 Handling automata

Automata Standard library. Cite the article

hash-table automaton Using a hash-table implementation speeds up the model checking process. This is the most efficient implementation in this case; however, it requires in compilation process to fix the size of the alphabet. In this case, it means to know in advance the number of relevant constraints.

–¿ hash table (explain complexities)

4.3 constraint deduction

We are using Gecode for constraint deduction based upon search.

Gecode Gecode is a constraint solving library very efficient equipped with state-of-the-art propagation and search algorithm. Get inspiration from Rueda's report.

algo To verify if a constraint c can be deduced from a constraint d (i.e, $d \vdash c$), we proceed as follows. First, we add a reified constraint of the form $d \Leftrightarrow b_1$ and $c \Leftrightarrow b_2$, where b_1 and b_2 are two new fresh boolean variables. Then, we post a constraint of the form $b_1 \Rightarrow b_2 \Leftrightarrow b_3$, where b_3 is a fresh boolean variable. Finally, we post a constraint $b_3 = 0$ and we perform search on the set of boolean reified variables b_1 , b_2 and b_3 . If the constraint c can be deduced from d , this problem should be insatisfiable and therefore, no solutions would be found.

removing equivalent constraints Valencia's definition of relevant constraints introduces many logical equivalent constraints, for instance $a \wedge b$ and $b \wedge a$. We reduce those.

4.4 parser

The parser follows the syntax of ntcc in a very intuitive manner. As an example, figure ? shows the process ?? and the formulae ??.

Tools We use Flex and Bison to make our parser. Flex is a lexical analyzer generator and Bison is a parser generator that generates a LALR parser for a context-free language.

Tokens /* PROCESSES */

token TTELL token TWHEN token TDO token TPARALLEL token TLOCAL
token TNEXT token TUNLESS token TSTAR token TBANG token TBANGN
token TWHENEVER token TABORT token TEQUALDEF token TSKIP token
TLEFTARROW token TIN

/*CONSTRAINTS*/ token TAND token TTRUE token TFALSE token TOR
token TIMPLY token TNEGATION token TEQUAL token TEXISTS token TGE
token TGQ token TLE token TLQ token TDIFFERENT token TPLUS // Multipli-
cation = star token TDIVISION token TMINUS token TID token TINTEGER

/*OTHERS*/ token TPARENTHESIS0 token TPARENTHESIS1 token TBRACK-
ET0 token TBRACKET1 token TPOINT token TCOMMA token TTIMEUNITS
token TREC token TCARET

/*LOGIC*/ token TSQUARE token TCIRCLE token TROMBO token TCLTLEX-
ISTS token TCLTLNEGATION token TCLTLAND token TCLTLTRUE token TCLTL-
FALSE token TCLTLOR token TCLTLIMPLY

Grammar

Grammar

0 \$accept: PROGRAM \$end

1 PROGRAM: TTIMEUNITS TEQUAL TINTEGER PROCESSDECLARATION TID TEQUALDEF FORMULA

```

2          | PROCESSDECLARATION TID TEQUALDEF FORMULA

3 PROCESSDECLARATION: PROCESSDECLARATIONLIST

4 PROCESSDECLARATIONLIST: TID TEQUALDEF PROCESS
5                      | TID TEQUAL PROCESS
6                      | PROCESSDECLARATIONLIST TCOMMA TID TEQUALDEF PROCESS

7 TIDLIST: TID
8          | TIDLIST TIDLIST

9 CONSTRAINT: TTRUE
10         | TFALSE
11         | TNEGATION CONSTRAINT
12         | CONSTRAINT TOR CONSTRAINT
13         | CONSTRAINT TAND CONSTRAINT
14         | CONSTRAINT TIMPLY CONSTRAINT
15         | TEXISTS CONSTRAINT
16         | LINEAR_LIST REL TINTEGER
17         | LINEAR_LIST
18         | LINEAR_LIST REL TID

19 REL: TLE
20     | TLQ
21     | TEQUAL
22     | TGQ
23     | TGE
24     | TDIFFERENT

25 LINEAR_LIST: TINTEGER TPOINT TID
26             | TMINUS TINTEGER TPOINT TID
27             | TID
28             | LINEAR_LIST TPLUS LINEAR_LIST
29             | LINEAR_LIST TMINUS LINEAR_LIST

30 PROCESS: TPARENTHESIS0 NONPARENTHESISPROCESS TPARENTHESIS1
31         | NONPARENTHESISPROCESS
32         | /* empty */

33 NONPARENTHESISPROCESS: TELL
34                     | PARALLEL
35                     | SUM
36                     | LOCAL
37                     | NEXT

```

```

38          | UNLESS
39          | STAR
40          | ABORT
41          | BANGN
42          | BANG
43          | BANGNM
44          | STARNM
45          | NEXTN
46          | PROCEDURECALL
47          | WHENEVER

48 TELL: TTELL TPARENTHESIS0 CONSTRAINT TPARENTHESIS0
49      | TTELL CONSTRAINT

50 WHEN: TWHEN CONSTRAINT TDO PROCESS

51 @1: /* empty */

52 WHEN: TWHEN CONSTRAINT @1 PROCESS

53 PARALLEL: PROCESS TPARALLEL PROCESS

54 SUM: SUM_LIST

55 SUM_LIST: WHEN
56      | SUM_LIST TPLUS SUM_LIST
57      | TSKIP

58 LOCAL: TLOCAL TID TIN PROCESS

59 NEXT: TNEXT PROCESS

60 UNLESS: TUNLESS CONSTRAINT TNEXT PROCESS
61      | TUNLESS CONSTRAINT TDO PROCESS

62 STAR: TSTAR PROCESS

63 BANGN: TBANGN PROCESS

64 BANG: TBANG PROCESS

65 ABORT: TABORT

66 NEXTN: NEXT TCARET TINTEGER PROCESS

```

```

67 BANGNM: TBANG TBRACKETO TINTEGER TCOMMA TINTEGER TBRACKETC PROCESS
68 STARNM: TSTAR TBRACKETO TINTEGER TCOMMA TINTEGER TBRACKETC PROCESS
69 PROCEDURECALL: TID
70                | TID TPARENTHESISO TIDLIST TPARENTHESISC
71 WHENEVER: TWHENEVER CONSTRAINT TDO PROCESS
72 FORMULA: TPARENTHESISO NONPARENTHESISFORMULA TPARENTHESISC
73          | NONPARENTHESISFORMULA
74          | /* empty */
75 NONPARENTHESISFORMULA: ALWAYS
76                        | EVENTUALLY
77                        | FORMULAAND
78                        | FORMULAOR
79                        | FORMULANEXT
80                        | FORMULAEXISTS
81                        | FORMULANEGATION
82                        | FORMULANEXTN
83                        | TTRUE
84                        | TFALSE
85                        | CONSTRAINT
86 ALWAYS: TSQUARE FORMULA
87 EVENTUALLY: TROMBO FORMULA
88 FORMULAAND: FORMULA TCLTLAND FORMULA
89 FORMULAOR: FORMULA TCLTLOR FORMULA
90 FORMULANEXT: TCIRCLE FORMULA
91 FORMULANEXTN: TCIRCLE TCARET TINTEGER FORMULA
92 FORMULAEXISTS: TCLTLEXISTS TID TIN FORMULA
93 FORMULANEGATION: TCLTLNEGATION CONSTRAINT

```

Associativity and priorities Explain associativity of operators and priorities /*

Asociatividad de los operadores */

left TCLTLIMPLY left TCLTLOR left TCLTLAND nonassoc TCLTLEXISTS
 TCLTLSQUARE TCLTLCIRCLE TCLTLROMBO TCLTLNEGATION left TEQUALDEF
 left TLEFTARROW left TPARALLEL nonassoc TNEXT nonassoc TBANG TWHEN-
 EVER TBANGN TSTAR TLOCAL TWHEN TDO nonassoc TEXISTS left TIM-
 PLY left TOR left TAND nonassoc TNOT left TLE TLQ TEQUAL TGQ TGE TD-
 IFFERENT left TPLUS TMINUS left TDIVISION nonassoc TUNARYMINUS

5 applications

5.1 African music example

#Test for a Nzakara described n = 30, p = 6, described in [RV04];

#Processes;

NOTES64 =def

```
#SUM V IN [0,29];
when errors2 = v do(
  unless PNU = CNL and (not PNL = CNL and not PNU = CNU) and (not BNL = CNL and r
    next tell (errors2 = v + 1)
  ||when PNU = CNL and (not PNL = CNL and not PNU = CNU) and (not BNL = CNL and r
    do
      ( tell (CNU = 64)
        || next tell (PNU = 64)
      || next^6 (tell (CNL = 60) #f(64) = 60;
        || tell (BNU = 64)
      || next tell (PNL = 60) ) #f(64) = 60;
        || next^12 tell (BNL = 60) #f(64) = 60;
        || next tell (errors2 = v)
      ))
```

NOTES67 =def

```
#SUM V IN [0,29];
when errors2 = v do(
  unless PNU = CNL and (not PNL = CNL and not PNU = CNU) and (not BNL = CNL and r
    next tell (errors2 = v + 1)
  ||when PNU = CNL and (not PNL = CNL and not PNU = CNU) and (not BNL = CNL and r
    do
      ( tell (CNU = 67)
        || next tell (PNU = 67)
      || next^6 (tell (CNL = 62) #f(67) = 62;
        || tell (BNU = 67)
```

```

|| next tell (PNL = 62) ) #f(67) = 62;
    || next^12 tell (BNL = 62) #f(67) = 62;
    || next tell (errors2 = v)
    ))

NOTES70 =def
#SUM V IN [0,29];
when errors2 = v do(
    unless PNU = CNL and (not PNL = CNL and not PNU = CNU) and (not BNL = CNL and not PNU = CNU)
        next tell (errors2 = v + 1)
    ||when PNU = CNL and (not PNL = CNL and not PNU = CNU) and (not BNL = CNL and not PNU = CNU)
        do
            (
                tell (CNU = 70)
                || next tell (PNU = 70)
            )
|| next^6 (tell (CNL = 64) #f(70) = 64;
    || tell (BNU = 70)
|| next tell (PNL = 64) ) #f(70) = 64;
    || next^12 tell (BNL = 70) #f(70) = 64;
    || next tell (errors2 = v)
    ))

CHOOSE =def
when true do NOTES64 + when true do NOTES67 + when true do NOTES70 # To do: define

COUNTER =def
#SUM V IN [0,29];
    when errors1 = v do (
        (when (CNL = 60 and CNU = 64) or (CNL = 60 and CNU = 67) or (CNL = 60 and CNU = 68)
            next tell (errors1 = v + 1)) #errors replaced by errors1;
        ||(unless (CNL = 60 and CNU = 64) or (CNL = 60 and CNU = 67) or (CNL = 60 and CNU = 68)
            next tell (errors1 = v))
    )
)

P =def
    ![0,5] tell (CNL = 0 and BNU = 0 and PNL = 0)
|| tell (errors1 = 0 and errors2 = 0)
|| next^6 tell (PNL = 0)
|| ![0,11] tell (BNL = 0)
|| ![0,29] (CHOOSE || COUNTER)

#Formulae;

```

`F =def < > ( )^30 (errors2 = 0 .and. errors1 < 7)`

6 conclusions and future work

6.1 summary

6.2 what we have learned

Although ntcc is finite state the current encoding to Büchi has a space and time complexity non-elementary. It has not been proven that a more efficient encoding cannot be made; however, we find that difficult by considering the complexity issues with such automata: complementation, non-equivalence between non-deterministic and deterministic, intersection. We believe that for practical purposes finite state is the better. Another option is proof system stuff.

6.3 limitations

- Formula and processes are bounded in time.
- CLTL formula are negation restricted.
- $!P$ cannot be represented as a Kleene star (*).
- The variables are finite domain, which range should not be too big because we are using search.
- The automaton class is parametrized by the size of the alphabet (the number of relevant constraints).
- The number of processes must be small because of technical problems. Meta processes should be used instead.
- Local processes generate too many relevant constraints.
- Non-recursive parametrized processes are not supported (e.g., ??).

6.4 future work

6.4.1 To do: intelligent parser: build a process graph, not a tree

In the future, the model checker must be optimized to threat larger problems and higher number of time units. There is a lot of work to do in the parser. As an example, to build a process graph instead of a process tree, so that two equivalent processes do not have to be encoded twice.

6.5 Acknowledgments

Frank Valencia, Andrés Aristizabal, Camilo Rueda, Carlos Olarte, Victor Rivera, Julian Gutierrez, Myriam Desainte-Catherine, Alexander Heussner, Alain Griffau.

References

- [1] A. Allombert, M. Desainte-Catherine, J. Larralde, and G. Assayag. A system of interactive scores based on qualitative and quantitative temporal constraints. In *Proc. of Artech 2008*, Porto, Portugal, 2008. The Artech International Association.
- [2] A. Allombert, M. Desainte-Catherine, and M. Toro. Modeling temporal constraints for a system of interactive score. In G. Assayag and C. Truchet, editors, *Constraint Programming in Music*, chapter 1, pages 1–23. Wiley, 2011.
- [3] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
- [4] J. Aranda, G. Assayag, C. Olarte, J. A. Pérez, C. Rueda, M. Toro, and F. D. Valencia. An overview of FORCES: an INRIA project on declarative formalisms for emergent systems. In P. M. Hill and D. S. Warren, editors, *Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009. Proceedings*, volume 5649 of *Lecture Notes in Computer Science*, pages 509–513. Springer, 2009.
- [5] G. Behrmann, A. David, K. G. Larsen, O. Miller, P. Pettersson, and W. Yi. UPPAAL - present and future. In *Proc. of 40th IEEE Conference on Decision and Control*. IEEE Computer Society Press, 2001.
- [6] M. Boyer and M. Diaz. Non equivalence between time petri nets and time stream petri nets. In *Proceedings of the The 8th International Workshop on Petri Nets and Performance Models*, number 198–, Washington, DC, USA, 1999. IEEE Computer Society.
- [7] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A model-checking tool for real-time systems. In A. Ravn and H. Rischel, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1486 of *Lecture Notes in Computer Science*, pages 298–302. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0055357.
- [8] J. Byg, K. Y. Jørgensen, and J. Srba. Tapaal: Editor, simulator and verifier of timed-arc petri nets. In *Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis, ATVA '09*, pages 84–89, Berlin, Heidelberg, 2009. Springer-Verlag.

- [9] F. S. de Boer, M. Gabbrielli, and M. C. Meo. A timed concurrent constraint language. *Inf. Comput.*, 161(1):45–83, 2000.
- [10] M. Falaschi and A. Villanueva. Automatic verification of timed concurrent constraint programs. *Theory and Practice of Logic Program*, 6(4):265–300, 2006.
- [11] G. Gardey, D. Lime, M. Magnin, and O. H. Roux. Roméo: A tool for analyzing time Petri nets. In *17th International Conference on Computer Aided Verification (CAV'05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 418–423, Edinburgh, Scotland, UK, 2005. Springer.
- [12] G. J. Holzmann. The model checker spin. *Software Engineering*, 23(5):279–295, 1997.
- [13] J. D. A. Moreno, S. Passos, and M. Toro. On-line assembling mitochondrial DNA from de novo transcriptome. *CoRR*, abs/1706.02828, 2017.
- [14] M. Nielsen, C. Palamidessi, and F. Valencia. Temporal concurrent constraint programming: Denotation, logic and applications. *Nordic Journal of Comp.*, 1(9):145–188, 2002.
- [15] C. Olarte, C. Rueda, G. Sarria, M. Toro, and F. Valencia. Concurrent Constraints Models of Music Interaction. In G. Assayag and C. Truchet, editors, *Constraint Programming in Music*, chapter 6, pages 133–153. Wiley, Hoboken, NJ, USA., 2011.
- [16] C. Patiño-Forero, M. Agudelo-Toro, and M. Toro. Planning system for deliveries in Medellín. *ArXiv e-prints*, Nov. 2016.
- [17] A. Philippou and M. Toro. Process Ordering in a Process Calculus for Spatially-Explicit Ecological Models. In *Proceedings of MOKMASD'13*, LNCS 8368, pages 345–361. Springer, 2013.
- [18] A. Philippou, M. Toro, and M. Antonaki. Simulation and Verification for a Process Calculus for Spatially-Explicit Ecological Models. *Scientific Annals of Computer Science*, 23(1):119–167, 2013.
- [19] A. Pnueli. The temporal logic of programs. In *In Proc. of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57. IEEE, IEEE Computer Society Press, 1977., 1977.
- [20] R. Ramirez. A logic-based language for modeling and verifying musical processes. In *Proc. of ICMC '06*, San Francisco, USA., 2006. Computer Music Association.
- [21] M. Raul, T. Mauricio, and C. Luz. Definicion de la arquitectura de referencia de un dominio: de la elucidacion al modelado. In R. Mazo, editor, *Guia para*

la adopcion industrial de lineas de productos de software, pages 193–210. Editorial Eafit, 2018.

- [22] J. M. C. Restrepo, A. F. Z. Palacio, and M. Toro. Assembling sequences of DNA using an on-line algorithm based on debruijn graphs. *CoRR*, abs/1705.05105, 2017.
- [23] P. V. Roy, editor. *Multiparadigm Programming in Mozart/Oz, Second International Conference, MOZ 2004, Charleroi, Belgium, October 7-8, 2004, Revised Selected and Invited Papers*, volume 3389 of *Lecture Notes in Computer Science*. Springer, 2005.
- [24] C. Rueda and F. Valencia. Proving musical properties using a temporal concurrent constraint calculus. In *In Proc. of the 28th International Computer Music Conference (ICMC 2002)*, 2002.
- [25] C. Rueda and F. Valencia. On validity in modelization of musical problems by ccp. *Soft Comput.*, 8(9):641–648, 2004.
- [26] C. Rueda and F. D. Valencia. Formalizing timed musical processes with a temporal concurrent constraint programming calculus. In *Proc. of Musical Constraints Workshop CP2001*, 2002.
- [27] V. A. Saraswat. *Concurrent Constraint Programming*. MIT Press, Cambridge, MA, 1992.
- [28] A. Schimpf, S. Merz, and J.-G. Smaus. Construction of büchi automata for ltl model checking verified in isabelle/hol. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs '09*, pages 424–439, Berlin, Heidelberg, 2009. Springer-Verlag.
- [29] P. Sénac, P. d. Saqui-Sannes, and R. Willrich. Hierarchical time stream petri net: A model for hypermedia systems. In *Proc. of the 16th International Conference on Application and Theory of Petri Nets*, pages 451–470, London, UK, 1995. Springer-Verlag.
- [30] S. H. T. Sjoland, Erik Klinskog. An interpreter for timed concurrent constraints in mozart. *Swedish Institute of Computer Science*, 2001.
- [31] M. Toro. Exploring the possibilities and limitations of concurrent programming for multimedia interaction and graphical representations to solve musical csp’s. Technical Report 2008-3, Ircam, Paris.(FRANCE), 2008.
- [32] M. Toro. Probabilistic Extension to the Factor Oracle Model for Music Improvisation. Master’s thesis, Pontificia Universidad Javeriana Cali, Colombia, 2009.

- [33] M. Toro. Structured interactive musical scores. In M. V. Hermenegildo and T. Schaub, editors, *Technical Communications of the 26th International Conference on Logic Programming, ICLP 2010, July 16-19, 2010, Edinburgh, Scotland, UK*, volume 7 of *LIPICs*, pages 300–302. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.
- [34] M. Toro. *Structured Interactive Scores: From a simple structural description of a multimedia scenario to a real-time capable implementation with formal semantics*. PhD thesis, Université de Bordeaux 1, France, 2012.
- [35] M. Toro. Structured interactive music scores. *CoRR*, abs/1508.05559, 2015.
- [36] M. Toro. Probabilistic Extension to the Concurrent Constraint Factor Oracle Model for Music Improvisation. *ArXiv e-prints*, Feb. 2016.
- [37] M. Toro. Probabilistic Extension to the Concurrent Constraint Factor Oracle Model for Music Improvisation. *Inteligencia Artificial*, 57(19):37–73, 2016.
- [38] M. Toro. CURRENT TRENDS AND FUTURE RESEARCH DIRECTIONS FOR INTERACTIVE MUSIC. *Journal of Theoretical and Applied Information Technology*, 69(16):5569–5606, 2018.
- [39] M. Toro, C. Agón, G. Assayag, and C. Rueda. Ntcrt: A concurrent constraint framework for real-time interaction. In *Proc. of ICMC '09*, San Francisco, USA., 2009. Computer Music Association.
- [40] M. Toro and M. Desainte-Catherine. Concurrent constraint conditional branching interactive scores. In *Proc. of SMC '10*, Barcelona, Spain, 2010.
- [41] M. Toro, M. Desainte-Catherine, and P. Baltazar. A model for interactive scores with temporal constraints and conditional branching. In *Proc. of Journées d'Informatique Musicale (JIM) '10*, May 2010.
- [42] M. Toro, M. Desainte-Catherine, and J. Castet. An extension of interactive scores for multimedia scenarios with temporal relations for micro and macro controls. In *Proc. of Sound and Music Computing (SMC) '12*, Copenhagen, Denmark, July 2012.
- [43] M. TORO, M. DESAINTE-CATHERINE, and J. CASTET. An extension of interactive scores for multimedia scenarios with temporal relations for micro and macro controls. *European Journal of Scientific Research*, 137(4):396–409, 2016.
- [44] M. Toro, M. Desainte-Catherine, and C. Rueda. Formal semantics for interactive music scores: a framework to design, specify properties and execute interactive scenarios. *Journal of Mathematics and Music*, 8(1):93–112, 2014.

- [45] M. Toro, A. Philippou, S. Arboleda, M. Puerta, and C. M. Vélez S. Mean-field semantics for a process calculus for spatially-explicit ecological models. In C. A. Muñoz and J. A. Pérez, editors, *Proceedings of the Eleventh International Workshop on Developments in Computational Models*, Cali, Colombia, October 28, 2015, volume 204 of *Electronic Proceedings in Theoretical Computer Science*, pages 79–94. Open Publishing Association, 2016.
- [46] M. Toro, A. Philippou, C. Kassara, and S. Sfenthourakis. Synchronous parallel composition in a process calculus for ecological models. In G. Ciobanu and D. Méry, editors, *Proceedings of the 11th International Colloquium on Theoretical Aspects of Computing - ICTAC 2014, Bucharest, Romania, September 17-19*, volume 8687 of *Lecture Notes in Computer Science*, pages 424–441. Springer, 2014.
- [47] M. TORO, C. RUEDA, C. AGÓN, and G. ASSAYAG. Ntcrt: A concurrent constraint framework for soft real-time music interaction. *Journal of Theoretical & Applied Information Technology*, 82(1), 2015.
- [48] M. TORO, C. RUEDA, C. AGÓN, and G. ASSAYAG. Gelisp: A framework to represent musical constraint satisfaction problems and search strategies. *Journal of Theoretical & Applied Information Technology*, 86(2), 2016.
- [49] M. Toro-Bermúdez. Towards a correct and efficient implementation of simulation and verification tools for probabilistic ntcc. Technical report, Pontificia Universidad Javeriana, May 2009.
- [50] F. D. Valencia. *Temporal Concurrent Constraint Programming*. PhD thesis, University of Aarhus, 2002.
- [51] F. D. Valencia. Decidability of infinite-state timed ccp processes and first-order ltl. *Theor. Comput. Sci.*, 330(3):557–607, 2005.