Towards a correct and efficient implementation of simulation and verification tools for Probabilistic ntcc

Mauricio Toro Bermúdez
mauriciotoro@cic.javerianacali.edu.co

August 11, 2009

Abstract

Using process calculi one can verify properties of a system and simulate the system as well. One should be able to do these operations at least semi-automatically. There is a tool to simulate the Non-deterministic Timed Concurrent Constraint (ntcc) calculus in the Ntccrt framework, but there are not tools for verification. We present a new simulation tool for Probabilistic ntcc (pntcc) and a prototype for verification of pntcc models. We include these tools in the Ntccrt framework. We also show the formal basis for correctness of the ntcc simulation tool and we show that the execution times of the simulation are still acceptable for real-time interaction using pntcc.

1 Introduction

Using process calculi one can simulate a model and also verify logical properties over it. For instance, the Non-deterministic Timed Concurrent Constraint (`ntcc`) [NPV02] calculus is as an extension of the Saraswat's concurrent constraint programming (ccp) [Sar92] to model new phenomena, in particular non-deterministic and timed behavior. The `ntcc` calculus is based on solid mathematical principles and it has succeed in a wide range of applications in emergent areas such as Security, Biology and Multimedia Semantic Interaction according to [Ola09]. `Ntcc` provides rich verification techniques allowing the inference of important temporal properties satisfied by the encoded applications. For instance, security breaches in cryptographic protocols [OV08], prediction of organic malfunctions [GPRV07] and rhythmic coherence in music improvisation [RV04].

It is also argued in [Ola09] that in spite of its modeling success, at present `ntcc` does not provide for the automatic, or even machine-assisted, verification of system properties. Since they deal with complex and large systems, machine-assisted verification is essential to our intended applications. However, this issue has not been deeply considered for ccp formalisms. To the best of their knowledge only Villanueva et al [MV06] have addressed automatic verification, but only in the context of finite-state ccp systems. Several interesting applications of `ntcc` are, however, inherently infinite-state according to [Ola09]. Automatic verification of large systems, not to mention infinite systems, represents a fundamental challenge because of the state explosion problem it poses because the number of states a system has is exponential in the number of concurrent processes. For that reason, it is proposed in [Ola09] to rise to this challenge by identifying `ntcc` fragments appropriate for automatic verification and developing efficient techniques and tools to machine assist the verification of system properties in `ntcc`.

Multiple tools have been developed for the simulation of `ntcc` models. For instance, the Ntccrt framework [TBAAR09]. However, it has not been proved formally the correctness of the framework's tools, thus, users cannot be sure that a simulation obtained using the `ntcc` simulation tool corresponds to the semantics of the calculus. In addition, Ntccrt does not provide tools for automated verification.

In this report we describe a formalization of the principles used by Ntccrt to simulate models. We also present the extension of the `ntcc` simulation tool for Probabilistic ntcc (`pntcc`) [PR08] models and we prove the correctness of the simulation tool for `ntcc`. In addition, we developed a verification tool for `pntcc` models. Using this verification tool, we can prove properties such as "the system will go to a successful state with probability p under t discrete time-units".

In what follows, we present in chapter 1 the formalization of the ntcc simulation tool. Then, in chapter 2 we explain the implementation of the frameworks' tools. In chapter 3, we present some applications ran using the framework's tools. Finally, we give some concluding remarks, current and future work in chapter 4.

2 Formal basis for simulation and verification for `ntcc`

In this chapter we explain the formal basis of our tool for simulation of `ntcc` models. Since `pntcc` is an extension of `ntcc`, some of these results could be later applied for `pntcc`.

2.1 Encoding a fragment of `ntcc` as a Constraint Satisfaction Problem

In this section we show how a star-free fragment of `ntcc` can be encoded as a Constraint Satisfaction Problem (CSP). We prove that the solutions of the CSP have a relation with the store computed in a single time unit. In such fragment, we do not not include the asynchronous operator (*) and we present a different operational semantic based on the concept of a scheduler or adversary (proposed in `pntcc` semantics).

Definition 1. *A **Constraint Satisfaction Problem (CSP)** is defined as a triple $\langle X, D, C\rangle$ where X is a set of variables, D is a set of domains and C is a set of constraints. A solution for a CSP is an evaluation that satisfies all constraints.*

The following star-free fragment of `ntcc` is parametrized by a Finite Domain FD[2^{32}] constraint system. Thirty-two is the size of an integer on a 32-bits computer architecture.

Definition 2. *A star-free fragment of `ntcc` parametrized by FD[2^{32}] where P, Q ::=*

$$P \| Q \mid \textbf{tell}\ (c) \mid \textbf{local}\ x\ \textbf{in}\ P \mid \sum_{i\in I} \textbf{when}\ c_i\ \textbf{do}\ P_i \mid \textbf{next}\ P \mid \textbf{unless}\ c\ \textbf{next}\ P \mid \textbf{!}P$$

Definition 3. *A **Constraint System (CS)** is a pair ($\sum$,Δ) where $\sum$ is a signature specifying constants, functions and predicate symbols, and Δ is consistent first-order theory over $\sum$ (i.e., a set of first-order sentences over $\sum$ having at least one model). We say that c entails d in Δ, written $c \models_\Delta d$ iff the formula $c \Rightarrow d$ is true in all models of Δ. We write $\models$ instead of $\models_\Delta$ when Δ is unimportant [Val02].*

Definition 4. *Let $n > 0$. A **Finite Domain FD(n)** is a CS such that:*

- $\sum$ is given by constant symbols $0..n-1$ and the equality.

- Δ is given by the axioms of equational theory $x = x, x = y \rightarrow y = x, x = y \wedge y = z \rightarrow x = z$, and $v = w \rightarrow$ `false` for each two different constants v,w in $\sum$ [Val02].

In order to define an encoding for a process given by Def. 2 into a CSP, we need to define a function to calculate the set of the non-local variables of a process and a function to solve non-deterministic choices for each $\sum$ process.

Definition 5. *Let vars(P): " `ntcc` process" → "set of variable names" be recursively defined*

$\text{vars}(\textbf{tell}(c)) = Cvars(c)$, the variables contained in a constraint.
$\text{vars}(P||Q) = \text{vars}(P) \cup \text{vars}(Q)$
$\text{vars}(\sum_{i\in I} \textbf{when } c_i \textbf{ do } P_i) = \bigcup_{i\in I} \text{vars}(P)$
$\text{vars}(\textbf{local } x \text{ in } P) = \text{vars}(P) - \{x\}$
$\text{vars}(\textbf{unless } c \textbf{ next } P) = \text{vars}(\textbf{next } P) = \emptyset$
$\text{vars}(\textbf{!}P) = \text{vars}(P)$

Before defining our function to solve non-determinism, let us introduce how the non-determinism is solved according to the pntcc semantics based on a Probabilistic Automata.

Definition 6. *A* ***Probabilistic Automata (PA)*** *is a tuple (Q, q, A, T), where Q is a countable set of states, $q \in Q$ is the start state, A is a countable set of actions, and $T \subset Q \times Probs(A \times Q)$ defines the transition groups of the automaton [Seg06]. In* `pntcc` *semantics, we do not need actions, thus, in this report we discard the set of actions. On the other hand, since we are describing non-probabilistic processes in this section, all the probabilities in the transitions are 1.*

We recall from the `pntcc` paper that a scheduler is a function $S_j : TM_i \to TM_i$ that takes a set of transition groups and returns a set of transition groups where we can only observe probabilistic choice.

One may think about defining a particular scheduler for each model one defines. This posses two problems: the person defining the model must be aware of the transition groups generated by each process involving non-determinism (parallel and non-deterministic choice processes); and the person has to consider all the possible inputs for a process, since the transition groups may be different according to the input.

In order to make things easier, we propose to define define a function FND : "Process, Set of $\mathbb{N}$" $\to \mathbb{N}$ to solve the non-determinism on each $\sum$ process P. FND takes the set of indexes of the guards that hold in a non-deterministic process and returns one index. Choosing an index with the function FND is equivalent to selecting a transition group generated by a $\sum$ process. The non-determinism generated by the parallel processes (after removing the non-determinism generated by the $\sum$ processes) is confluent (i.e., independent from the scheduling strategy) according to Falaschi et al. [FGMP97]. To clarify the purpose of FND, consider the execution of the first time unit of an improvisation process that chooses between playing a note or not.

$P \overset{def}{=} \textbf{tell}(\text{playnote} = \texttt{true}) + \textbf{tell}\ (\text{playnote} = \texttt{false})$
$Improv \overset{def}{=} \textbf{tell}\ (\text{dur=250}) \parallel \text{P}$

In the process P, both guards hold. Then, the purpose of the function FND is to solve the non-deterministic choice. For instance, a function $FND(P, I) = min(I)$ chooses the process that plays the note. On the other hand, the function $FND(P, I) = max(I)$ chooses not to play the note. According to Falaschi et al. [FGMP97], we get the same results executing **tell** (dur=250) and P in any order once the non-deterministic choices are solved.

Let us explore the concept of scheduler defined in the operational semantics. Consider a store d as the input for the *Improv* process. Then, the probabilistic automata M_i represents the transitions for the first time unit of the improvisation process.

$M_i = (T_i, \gamma_0, TM_i)$
$T_i = \{\gamma_1 = \langle skip, dur = 250 \wedge playnote \wedge d\rangle, \gamma_2 = \langle skip, dur = 250 \wedge \neg playnote \wedge d\}$
$\gamma_0 = \langle Improv, d\rangle$
$TM_i = \{\{\gamma_0\{\rightarrow_1 \gamma_1\}, \gamma_0\{\rightarrow_1 \gamma_2\}\}\}$

In the example above, the scheduler returns $\{\gamma_o\{\longrightarrow_1 \gamma_1\}\}$ if the function $FND(P, I) = min(I)$ or $\{\gamma_o\{\longrightarrow_1 \gamma_2\}\}$ if the function $FND(P, I) = max(I)$.

In what follows, we use the function FND to make an encoding of the star-free fragment of `ntcc` given by Def. 2 into a CSP. In order to define the encoding, we assume a constraint $C \leftrightarrow b$ (where b does not occur free in C) in the constraint system for each constraint used as guard in the $\sum$ processes. These constraints are called *reified constraints.* Using constraints as guards for $\sum$ processes are limited by the reified constraints supplied by the constraint solving tool.

Definition 7. ***Reification*** *represents the validity of a constraint into a 0/1-variable. Therefore, constraints can be combined using 0/1-variables. The reification of a constraint C with respect to a variable x is the constraint $(C \leftrightarrow b) \wedge b \in \{0, 1\}$ where it is assumed that b does not occur free in C. The operational semantics of a propagator for the reification of a constraint C with respect to b is given by the following rules [Kor01]:*

- *If the constraint store entails b = 1, the propagator for the reification reduces to a propagator for C.*
- *If the constraint store entails b = 0, the propagator for the reification reduces to a propagator for C.*
- *If a propagator for C would realize that the constraint store entails C, the propagator for the reification tells b = 1 and ceases to exist.*
- *If a propagator for C would realize that the constraint store is inconsistent with C, the propagator for the reification tells b = 0 and ceases to exist.*

Next, we define the encoding of a `ntcc` process given by Def. 2 into a constraint. The encoding is parametrized by the function FND.

Definition 8. *Let $[[P\ ]]_{FND}$: "`ntcc` process" $\rightarrow$ "constraint" be defined recursively.*

$[\![\mathbf{tell}(c)]\!]_{FND} = c$

$$[\![\sum_{i\in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i]\!]_{FND} = \begin{cases} [\![P_j]\!]_{FND} & I \neq \emptyset \\ \texttt{true} & I = \emptyset \end{cases}$$

where $j = FND(P, I)$ and $I = \{i | \exists b.i \in I \wedge c_i \leftrightarrow b \wedge b = 1\}$
$[\![Q || R]\!]_{FND} = [\![\mathrm{Q}]\!]_{FND} \wedge [\![\mathrm{R}]\!]_{FND}$
$[\![\mathbf{local}\ \mathrm{x}\ \mathbf{in}\ Q]\!]_{FND} = \exists x.[\![\mathrm{Q}]\!]_{FND}$
$[\![\mathbf{next}\ \mathrm{Q}]\!]_{FND} = [\![\mathbf{unless}\ \mathrm{c}\ \mathbf{next}\ \mathrm{Q}]\!]_{FND} = \texttt{true}$
$[\![\mathbf{!}\mathrm{Q}]\!]_{FND} = [\![Q]\!]_{FND}$

Definition 9. *A* ***scheduler*** S_j ***defined by a function*** FND *is a function* $S_j : TM \rightarrow TM$*, such that a transition group* $\gamma_i\{\rightarrow_1 \gamma_{i+1}\} \in S_j(TM_i)$ *if* $\langle P, c\rangle = \gamma_i, \langle Q, d\rangle = \gamma', [\![P]\!]_{FND} \wedge c = d$*, it exists a path from* γ_{i+1} *to* γ' *in* TM_i *and* γ' *has no more successors. Non-determinism in parallel processes can be discarded because it is confluent in this context.*

It exists a path from A to B in a set of transition groups TM *iff the following groups belong to the set* $A\{\longrightarrow_1 A_1\})...A_i\{\longrightarrow_1 A_{i+1}\}...\{A_{i+1}\{\longrightarrow_1 B\}$*. B has not successors in a set of transition groups TM iff there is not a group* $B\{\longrightarrow_1 X\}$ *in TM.*

To prove the correctness of the encoding presented above, we need to prove three properties. (1) All the solutions for a CSP given by the constraint computed by the previous encoding implies the store calculated by a process at the end of a time unit. (2) The future function, calculated using the previous encoding, corresponds to the future function defined in the semantics. (3) The store calculated by a process at the end of a time unit has a connection to the solutions of the CSP (which connection do they have?). In this report we do not present propositions (2) and (3). Before presenting the proposition (1), we present some theorems taken from the `ntcc` operational semantics.

Lemma 1. *Every sequence of internal sequences is terminating (i.e., there are not infinite sequences) [Val02].*

Property 1. ***Internal Extensiveness*** *if* $\langle P, c\rangle \longrightarrow \langle Q, d\rangle$ *then* $d \models c$*. [Val02].*

Proposition 1. *Let* P*,* FND *and* $[[P]]_{FND}$ *be a process in the* `ntcc` *fragment given by Def. 2, a function* FND : *set of* $\mathbb{N} \rightarrow \mathbb{N}$ *and the encoding of* P *given by Def. 10. Then, for every constraint* c*, it exists a constraint* d *such that*

$\forall x.x \in Solutions\ of\ the\ CSP$

$\quad Variables : Vars(P)$

$\quad Domain\ for\ each\ variable : [0..2^{32} - 1]$

$\quad Constraints : \{[[P]]_{FND} \wedge c\}$

then, in any time unit, it holds that $\langle P, c\rangle\{\longrightarrow_1^* \langle Q, d\rangle\}_{S_j} \not\longrightarrow$ *and* $x \Rightarrow d$*, where* S_j *is a scheduler defined by* FND *according to Def. 9.*

Proof. $\{\langle P, c\rangle \longrightarrow_1^* \langle Q, d\rangle\}_{S_j} \not\longrightarrow$holds by lemma 1. We have left to prove that for all x in the solutions of the CSP, where the constraints are given by $\{[[P]]_{FND} \wedge c\}$, it holds that $x \Rightarrow d$. The proof proceeds by induction on the structure of P.

- P = tell(e)

  By Def. 10, we have $[\![P]\!]_{FND} \wedge c = e \wedge c$ and it holds that $\langle tell(e), c\rangle\{\longrightarrow_1 \langle skip, c \wedge e\rangle\}_{S_j} \not\longrightarrow$according to the TELL rule. Then, $d = e \wedge c$ and it holds that $x \Rightarrow c \wedge e$.

- P = $\sum\limits_{i \in I}$ when c_i do P_i

  By the inductive hypothesis, we assume that the proposition holds for each $i \in I$: for all x_i in the solutions of the CSP given by the constraint $[[P_i]]_{FND} \wedge c$, then it holds that $x_i \Rightarrow d_i$.

  We have to prove that for all x in the solutions of the CSP given by the constraint $[\![P]\!]_{FND} \wedge c$ it holds that $x \Rightarrow d$. There are two cases:

  - Case where all the guards are false or they cannot be deduced from the store: We know that $[\![P]\!]_{FND} = true$ by Def. 10 and $\langle P, c\rangle \not\longrightarrow$by the semantics of $\sum$. Then, for all x in the solutions for the CSP given by the constraint $true \wedge c$, it holds that $x \Rightarrow c$.

  - Case where at least one guard holds.

    Let I be the set of the indexes whose guards holds, $j = FND(P, I)$ and S_j the scheduler defined by FND, then it holds that $\langle P, c\rangle\{\longrightarrow_1 \langle P_j, e\rangle\}_{S_j}$

    We know by the inductive hypothesis that $\langle P_j, e\rangle \longrightarrow_1^* \langle Q_j, d_j\rangle\}_{S_j} \not\longrightarrow$ and for all x in the solutions of $[[P_j]]_{FND} \wedge e$, it holds that $x \Rightarrow d_j$.

    By the property 1, $e \models c$, then $e \Rightarrow c$ by Prop. 1. In the same way, $e \models d_j$, then $e \Rightarrow d_j$, therefore $c \Rightarrow d_j$.

    Finally, since $[[P]]_{FND} = [[P_j]]_{FND}$ by Def. 10, then it holds for all x in the solutions of the CSP given by the constraint $[\![P]\!]_{FND} \wedge c$ that $x \Rightarrow d_j$.

- P = $Q\|R$

  By the inductive hypothesis, we assume that the proposition holds for each Q and R. We have to prove that for all x in the solutions of the CSP given by the constraint $[\![Q]\!]_{FND} \wedge [\![R]\!]_{FND} \wedge c$ it holds that $x \Rightarrow d$.

  We know that $d \models [\![Q]\!]_{FND}$, $d \models [\![R]\!]_{FND}$, $d \models [\![Q]\!]_{FND} \wedge [\![R]\!]_{FND}$, $d \models [\![Q]\!]_{FND} \wedge [\![R]\!]_{FND} \wedge c$, but how can we prove that $[\![Q]\!]_{FND} \wedge [\![R]\!]_{FND} \wedge c \models d$?

- P = **local** x **in** Q

  By the inductive hypothesis, the assume that the proposition holds for Q. This means that $\langle Q, e \wedge \exists x.c\rangle \longrightarrow \langle Q', e\rangle \longrightarrow^* \langle Q'', d'\rangle \not\longrightarrow$ and for all x in the solutions in the CSP given by $[\![Q]\!]_{FND}$, it holds that $x \Rightarrow d'$.

  We have to prove that for all x in the solutions of the CSP given by the constraint $[[P]]_{FND} \wedge c$, it holds that $x \Rightarrow d$. We have $[[P]]_{FND} = \exists x.[[Q]]_{FND}$ by Def. 10. By lemma 1 and the LOC rule, we know that $\langle local\ (x, c)\ in\ P, c\rangle \longrightarrow \langle local\ (x, c')\ in\ P', e \wedge \exists x.c\rangle \longrightarrow^* \langle Q, f \wedge \exists x.c' \wedge \exists x.c'' \wedge ...\exists x.c^n\rangle \not\longrightarrow$. Then $d = f \wedge \exists x.c' \wedge \exists x.c'' \wedge ...\exists x.c^n$. What is the relation between d and f?

- P = **next** Q and P = **unless** c **next** Q

  By Def. 10, $[\![P]\!]_{FND} = \texttt{true}$. We also know that $\langle P, c\rangle \not\longrightarrow$. Then, $d = c$ and for all x in the solutions of $c \wedge true$, it holds that $x \Rightarrow d$.

- P = !Q

  We assume by the inductive hypothesis that the proposition holds for Q. Since we know that $[\![P]\!]_{FND} = [\![Q]\!]_{FND}$ by Def. 10, then the proposition holds for P.

□

In what follows, we define a function $\mathbb{F}$ that returns the process to be executed in the next time unit.

Definition 10. *Let $\mathbb{F}(P)_{FND,d}$: "`ntcc` process" $\rightarrow$ "`ntcc` process" be defined recursively.*

$\mathbb{F}(\textbf{tell}(c))_{FND,d} = skip$

$\mathbb{F}(\sum_{i\in I} \textbf{when } c_i \textbf{ do } P_i)_{FND,d} =$

$$\begin{cases} \mathbb{F}(P_j)_{FND,d} & \|I\| \neq 0 \\ skip & \|I\| = 0 \end{cases}$$

where $j = FND(P, I)$ and $I = \{i|\exists b.i \in I \wedge c_i \leftrightarrow b \wedge b = 1\}$

$\mathbb{F}(Q||R)_{FND,d} = \mathbb{F}(\text{Q})_{FND,d}||\mathbb{F}(\text{R})_{FND,d}$
$\mathbb{F}(\textbf{local } x \textbf{ in } Q)_{FND,d} = \textbf{local } x \textbf{ in } \mathbb{F}(\text{ Q})_{FND}$
$\mathbb{F}(\textbf{next } \text{Q})_{FND,d} = \text{Q}$
$\mathbb{F}(\textbf{unless } c \textbf{ next } \text{Q})_{FND} =$

$$\begin{cases} Q, & d \models c \\ skip, & d \not\models c \end{cases}$$

$\mathbb{F}(\textbf{!}\text{Q})_{FND,d} = \mathbb{F}(Q)_{FND,d}||!Q$

To prove proposition (3), we have to prove that the process calculated by the function $\mathbb{F}$ is equivalent to the process calculated by the future function defined in the operational semantics of ntcc.

3 Implementation of the tools for ntcc and pntcc

In the implementation, we use Gecode as the constraint-solving library. Gecode is an efficient library using state-of-the-art algorithms for propagation. We do not use search is our tools, thus, our implementation heavily relies on propagation. Gecode has a great advantage, it is easily extensible: we can define new propagators without recompiling it.

We use Gecode for concurrency control. Although threads are a common way to represent concurrency, we do not use them in our implementation. Instead, we rely on the scheduler of Gecode propagators to execute all the parallel processes concurrently. In the previous section, we show how we can encode all the processes as constraints, the intuition of the simulation and verification tools is to use a propagator for each of those constraints and let Gecode schedule those propagators.

Tools for Ntccrt are written in C++ because they runs faster than using other programming languages and using a wrapper for Gecode. In addition, all the processes have to be represented as propagators and at the end Gecode propagators are written in C++. We also provide interfaces for Common Lisp, OpenMusic [BAA05], Max/MSP and Pure Data [PAZ98]. The reader may find more information about the implementation of Ntccrt at [TBAAR09] and [TB08]. In what follows, we explain how to represent each ntcc process as a propagator and how to extend the ntcc simulation tool for the simulation and verification of pntcc models.

3.1 Representing each ntcc process as a propagator

Processes are represented by classes. These classes inherit from a class Process declaring a virtual Execute method.

Tell classes have an Execute method to post a Gecode propagator. For instance, a process $\textbf{tell}(a = b)$ is represented by a class, which execute method posts an equality-relation propagator. Objects of the Parallel class contain a list of processes. Its Execute method calls the Execute method for each object in the list.

In Local processes, local variables are fresh variables created on execution time. Variables are classes wrapping Gecode variables. Gecode variables cannot be used directly because they only exist during a single time unit, instead we create a an instance of our Variable class that creates a new instance of a Gecode variable for each time unit. Non-local variables are created at the beginning of the simulation.

Non-deterministic-choice classes have an Execute method that creates a non-deterministic-choice propagator. This propagator randomly chooses a process from those whose guards holds. We use reification to represent the conditions and we use an array of reified variables as a parameter for the non-deterministic-choice propagator. This propagator is the implementation of one possible scheduler. However, this propagator can be changed to stablish other policies of scheduling.

Temporal processes interact with a sequence of time units. Time units are represented by a list of queues of processes. For instance, The execution of the Process object inside a Next object is postponed for the next time unit. The execution of a Bang object calls the Execute method of its nested process. It also duplicates the Bang object for the next time unit.

Finally, **unless** c **do** P is represented by a class, which execution is postponed until all the propagators for the processes (except unless processes) achieve a common fixpoint. We recall that a propagator is a function f: *store* $\rightarrow$ *store*, thus, a common fixpoint for all the propagators is a state such that none of the propagators can perform an action that changes the store. After the fixpoint is calculated, the unless Execute method postpones the execution of P for the next time-unit if c cannot be deduced from the store.

3.2 Extending the simulation tool for `pntcc`

In order to extend the `ntcc` simulation tool for `pntcc`, we only need to add a new class for probabilistic choice inheriting from the Process class. This class takes an array of Process objects; an array of boolean variables, used to represent the reified constraints of the guards; and an array of integers, used to represent the probabilities.

To avoid errors produced by float-number calculations, users must represent the probabilities as integers. For instance, $[0.3, 0.4, 0.3]$ can be represented as $[30, 40, 30]$ or $[3, 4, 3]$. Precision of the probabilities is limited by the size of non-signed integers (32 bits in a 32-bits computer architecture). Users also need to provide a reified constraint for each guard in the process.

Definition 11. *A **Discrete-Time Markov Chain (DTMC)** is a tuple $\langle Q_{OBS}, q, T_{OBS}, LM\rangle$, where Q_{OBS} is a finite set of states (with initial state $q \in Q_{OBS}$); $T_{OBS} : Q_{OBS} \times Q_{OBS} \rightarrow [0, 1]$ is a transition, such that $\forall q \in Q_{OBS}, \sum_{q' \in Q_{OBS}} T_{OBS}(q, q') = 1$; and the labeling function $LM : Q_{OBS} \rightarrow 2^A$ assigns atomic propositions to states.*

There is a connection between a `pntcc` process and a DTMC. We recall the following proposition from the `pntcc` paper.

Proposition 2. *Given a `pntcc` process P_0, for every P_n reachable from P_0 through an observable sequence, in the DTMC given by DTMC($\langle P_o, true\rangle$) there exists a path from $\langle P_0, true\rangle$ to $\langle P_n, d\rangle$, for some constraint d.*

In order to make a simulation for a `pntcc` model, we only need to calculate a single path of the DTMC produced by a `pntcc` process. In order to achieve that, we solve non-determinism using the user-defined FND function. On the other hand, in order to solve probabilistic choice, we perform a simple algorithm. (1) Calculate the sum of the list of integers (representing probabilities) for the indexes whose guards that holds. (2) Using a random-number-generator library, calculate a number between 0 and $sum - 1$. (3) According to the number generated by the library, choose the appropriate process and call its Execute method.

3.3 A verification tool for `pntcc`

The purpose of our verification tool is calculating a DTMC based on three things: a list of inputs (i.e., constraints), a `pntcc` model and a function FND to solve non-determinism.

Then, the DTMC is saved as a PRISM input file. Finally, using PRISM, the tool can verify properties for the model.

According to `pntcc` authors, there is a straightforward connection with model-checking procedures for `pntcc`: "The rationale given by the language and the support provided by operational semantics make it possible a natural relation between `pntcc` and the probabilistic logic PCTL [CG04]. Formulas in PCTL express soft deadlines, statements explicitly involving a time bound and a probability. Hence, they allow for including quantitative parameters directly in the properties of interest. The relation between pntcc and PCTL is based on the fact that the observable behavior of a process can be interpreted as the discrete time Markov chain (DTMC) defining satisfaction in PCTL. This way, model checking for tccp process specifications becomes possible". [PR08]

PRISM can verify Probabilistic Computational Tree Logic (PCTL) properties for a DTMC. Therefore, if we can construct a DTMC for a `pntcc` process using our verification tool, the verification is straightforward. In order to create a DTMC, we need to extend the simulation tool. First, we need to extend the definition of the variables, since a variable with the same name can be in different time units in the simulation tool, but now, there are several instances of a time unit. We also need to extend the way how temporal process add processes to the next time unit, since there are several instances of a time unit.

The execution of a time unit in the verification tool is slightly different from the simulation tool. It proceeds in the following way. First, we create a state $\langle S, c\rangle$. For every time unit, $P \| Q$, $\sum$ and local are executed as described for the simulation tool. Temporal processes create a new state and a transition from the current state to the new state. Probabilistic processes create a state and a transition for each guard that holds, except the first guard, which process is executed in the current state.

The last step is to reduce the DTMC-tree into a DTMC by reducing repeated states. There is a configuration $\langle P, C\rangle$ associated to each state by the means of the labeling function. We consider that two configurations are equal if their processes are structural congruent and they have logical equivalent stores.

To check structural congruence, Valencia proposes in [Val02] to construct a syntax tree for each process and check whether they are isomorphic. This feature is still experimental in our tool.

4 Applications

In this section we present some applications ran with the simulation and verification tools for `pntcc` models.

4.1 Probabilistic Ccfomi

Concurrent Constraint Factor Oracle Model for Music Improvisation (Ccfomi) [RAD06] is a `ntcc` model for music improvisation, where improvisation is made non-deterministically. Probabilistic Ccfomi [PR08] extends the choice in the improvisation process probabilistically. We ran Probabilistic Ccfomi in our tool and we defined constants for the probabilities of the probabilistic choice process.

4.2 The zigzag robot

Zigzagging is a task on which a robot can go either forward, left, or right, but (1) it cannot go forward if its preceding action was to go forward, (2) it cannot turn right if its

second-last action was to go right, and (3) it cannot turn left if its second-last action was to go left. Valencia models this problem by using cells a_1 and a_2 to "look back" and three different distinct constants $f, r, l \in D - \{0\}$ and the predicate symbols forward, right, left [Val02].
Using our ntcc simulation tool, we obtain the following data

direction	last direction	second-last direction
1	0	0
3	1	0
1	3	1
2	1	3
2	2	1
1	2	2
3	1	2
3	3	1
1	3	3
2	1	3
2	2	1
3	2	2
1	3	2

where $f = 1, r = 2, l = 3$.

Valencia verified that the model eventually goes right or left.For this model, we proved that the robot does not go forward twice, the robot does not go right more than twice, the robot does not go left more than twice, the robot always makes a good move.

We also provide a probabilistic extension of this model where the direction is chosen probabilistically. We also keep track of the cartesian coordinate (x,y) of the robot. Using our pntcc simulation tool, we obtain the following data data

direction	last direction	second-last direction	x	y
2	0	0	0	0
1	2	0	1	0
3	1	2	1	1
1	3	1	0	1
2	1	3	0	2
2	2	1	1	2
1	2	2	2	2
3	1	2	2	3

For this model, we use the verification tool to find a probability distribution for the position (x,y) of the robot. We also provide a graphical simulation using "ardilla python" (a visual programming environment based on robot moving on a two-dimensional board').

4.3 Herman stabilization protocol

A self-stabilizing protocol (according to PRISM website) for a network of processes is a protocol which, when started from some possibly illegal start configuration, returns to a legal/stable configuration without any outside intervention within some finite number of steps.

The network is a ring of identical processes. The stable configurations are those where there is exactly one process designated as "privileged" (has a token). This privilege (token) should be passed around the ring forever in a fair manner. For further details on self-stabilization see http://www.prism.org.

An interesting property for these protocols is to compute the minimum probability of reaching a stable configuration and the maximum expected time (number of steps) to reach a stable configuration (given that the above probability is 1) over every possible initial configuration of the protocol.

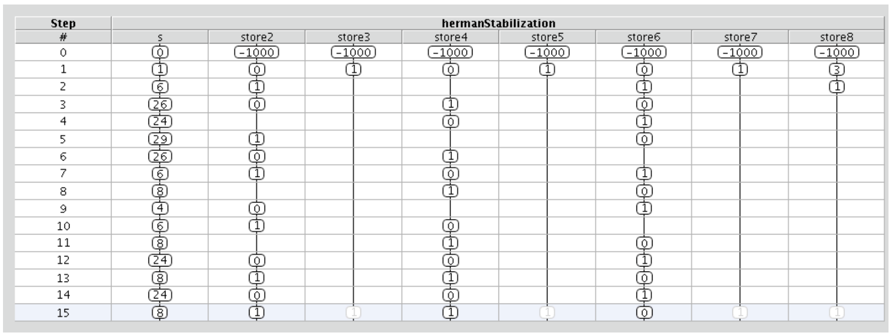
We modeled Herman's protocol [Her90]. This protocol is described in PRISM website as follows: "The protocol operates synchronously, the ring is oriented, and communication is unidirectional in the ring. In this protocol the number of processes in the ring must be odd.

Each process in the ring has a local boolean variable x_i, and there is a token in place i if $x_i = x_{i-1}$. In a basic step of the protocol, if the current values of x_i and x_{i-1} are equal, then it makes a (uniform) random choice as to the next value of x_i, and otherwise it sets it equal to the current value of x_{i-1}".

For instance, consider a simulation for the protocol for three nodes and an initial state where $x_0 = 0, x_1 = 0, x_2 = 0$. Using the simulation tool, we obtain the following data

number of tokens	x0	x1	x2
1	0	0	0
1	1	0	0
1	0	1	1
1	1	0	0
1	0	1	1
1	1	0	0
1	0	1	1
1	1	0	1
1	1	1	0

We can also create a DTMC model for PRISM and then perform a simulation (fig. 1).



Step	hermanStabilization							
#	s	store2	store3	store4	store5	store6	store7	store8
0	0	-1000	-1000	-1000	-1000	-1000	-1000	-1000
1	1	0	1	0	1	0	1	3
2	6	1				1		1
3	26	0		1		0		
4	24			0		1		
5	29	1				0		
6	26	0		1				
7	6	1		0		1		
8	8			1		0		
9	4	0				1		
10	6	1		0				
11	8			1		0		
12	24	0		0		1		
13	8	1		1		0		
14	24	0		0		1		
15	8	1	1	1	1	0	1	1

Figure 1: simulation

In the same way as it is done for PRISM study cases, we check the correctness of the protocol, namely that: from any configuration, a stable configuration is reached with probability 1 and the minimum probability of reaching a stable configuration within K steps (from any configuration). In figure 2, we present the chart obtained on PRISM study cases and then, the graph obtained using our tool in joint work with PRISM.

5 Results, conclusions and future work

In a previous work [TBAAR09], we ran *Ccfomi* as an stand-alone application over an Intel 2.8 GHz iMac using Mac OS 10.5.2 and Gecode 2.2.0. Each time-unit took an average of

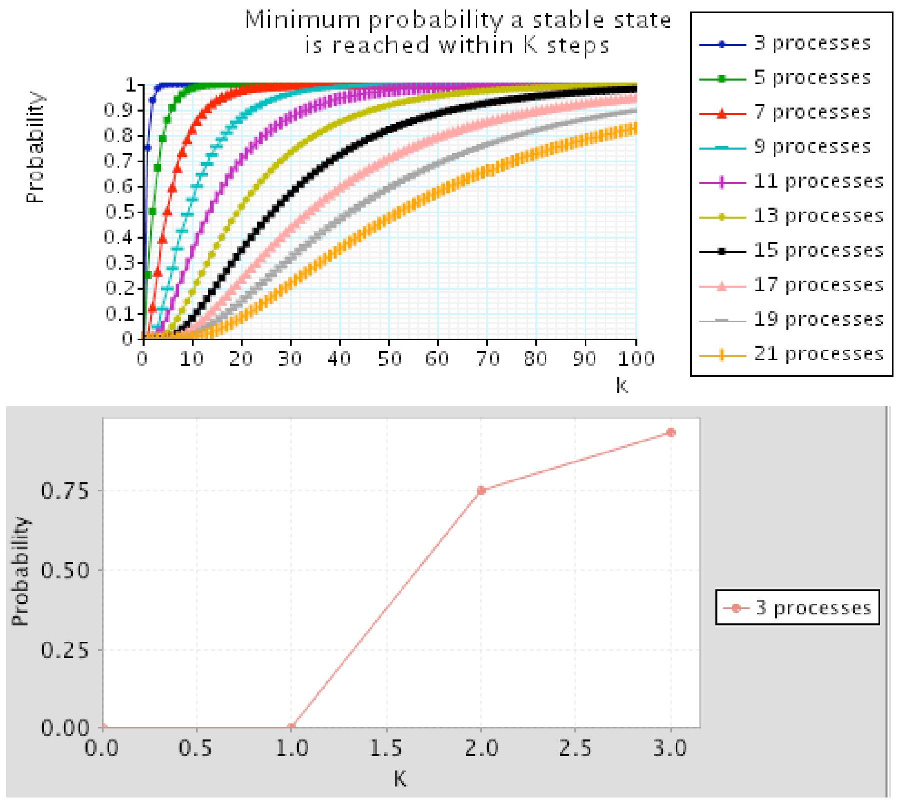


Figure 2: The minimum probability of reaching a stable configuration within K steps

20 ms, scheduling around 880 processes per time-unit. We simulated 300 time-units and we ran each simulation 100 times in our tests.

In this work, we compared the performance of CCFOMI and Probabilistic CCFOMI. The probabilistic model was around 50% slower than CCFOMI. Although, we still can do multiple optimizations to the `pntcc` simulation tool, we conjecture that the performance is still acceptable for real-time interaction.

Pachet argues in [Pac02] that an improvisation system able to learn and produce sequences in less than 30ms is appropriate for real-time interaction. Since our implementation of *Ccfomi* has a response time of 20ms in average, we conclude that it is capable of real-time interaction for a 300 (or less) time-units simulation. There is not enough data to conclude if the simulation of Probabilistic Ccfomi is also appropriate for real-time interaction.

For this work, we made all the test under Mac OS X using Pd. However, we also ran successfully the tools on Linux. Since we are using Gecode and Flext to generate the externals, they were easily compiled to Linux. In the future, we will also compile them to use Max/MSP. This is due to Gecode and Flext portability.

In this report, we presented the formal basis of the `ntcc` simulation tool of the Ntccrt framework. We presented two new tools for the framework: a simulation tool for `pntcc` and a verification tool for `pntcc`. We also discussed some application ran using the simulation and verification tools.

We want to encourage the use of process calculi to develop reactive systems. For that reason, this research focuses on developing real-life applications with `ntcc` and `pntcc` and showing how we can verify properties of those systems with a tool, from which we also prove its correctness. In a previous work, we showed that our interpreter Ntccrt is a user-friendly tool, providing a graphical interface to specify `ntcc` models and compiling them to efficient C++ programs capable of real-time interaction in Pd. In this report, we show that the new framework's tools are also user friendly and efficient. Furthermore, they have a formal basis.

In the International Colloquium on Emergent Trends in Concurrency Theory in Paris in honor of Turing-award winner Robin Milner [VP08], leading researchers reflected about strategic directions in concurrency theory, but we adhere (as well as presented on [Ola09] in particular to Garavel's position statement [Gar08]: "The times have gone, where formal methods were primarily a pen-and-pencil activity for mathematicians. Today, only languages properly equipped with software tools will have a chance to be adopted by industry. It is therefore essential for the next generation of languages based on process calculi to be supported by compilers, simulators, verification tools, etc. The research agenda for theoretical concurrency should therefore address the design of efficient algorithms for translating and verifying formal specifications of concurrent systems".

In the future, we want to extend our implementation to fully support `pntcc` simulation and verification. We also want to support `rtcc` [Sar08], and to generate an input for Spin [Hol97] based on a `ntcc` model for verification of temporal properties in `ntcc`.

6 Acknowledgments

In first place, we want to thank Camilo Rueda and Jorge Pérez for supervising this research. We also want to thank Gerardo Sarria, Yamil S. Perchy, Víctor Rivera and Jorge Victoria for their valuable comments during this research.

References

[BAA05] Jean Bresson, Carlos Agon, and Gérard Assayag. Openmusic 5: A cross-platform release of the computer-assisted composition environment. In *10th Brazilian Symposium on Computer Music*, Belo Horizonte, MG, Brésil, Octobre 2005.

[CG04] F. Ciesinski and F. Größer. On probabilistic computation tree logic. In *Validation of Stochastic Systems*, pages 147–188, 2004.

[FGMP97] M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Confluence in concurrent constraint programming. *Theor. Comput. Sci.*, 183:281–315, 1997.

[Gar08] Hubert Garavel. Reflections on the future of concurrency theory in general and process calculi in particular. *Electron. Notes Theor. Comput. Sci.*, 209:149–164, 2008.

[GPRV07] Julian Gutiérrez, Jorge A. Pérez, Camilo Rueda, and Frank D. Valencia. Timed concurrent constraint programming for analyzing biological systems. *Electron. Notes Theor. Comput. Sci.*, 171(2):117–137, 2007.

[Her90] T. Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35(2):63–67, 1990.

[Hol97] Gerard J. Holzmann. The model checker spin. *Software Engineering*, 23(5):279–295, 1997.

[Kor01] Leif Kornstaedt. Alice in the land of Oz – an interoperability-based implementation of a functional language on top of a relational language. In *Proceedings of the First Workshop on Multi-language Infrastructure and Interoperability (BABEL'01), Electronic Notes in Computer Science*, volume 59, Firenze, Italy, September 2001. Elsevier Science Publishers.

[MV06] Falaschi Moreno and Alicia Villanueva. Automatic verification of timed concurrent constraint programs. *Theory Pract. Log. Program.*, 6(3):1471–0684, 2006.

[NPV02] M. Nielsen, C. Palamidessi, and F. Valencia. Temporal concurrent constraint programming: Denotation, logic and applications. *Nordic Journal of Computing*, 1, 2002.

[Ola09] Carlos Olarte. React plus proposal: http://www.lix.polytechnique.fr/comete/pp.html, february 2009.

[OV08] Carlos Olarte and Frank D. Valencia. The expressivity of universal timed ccp: undecidability of monoadic fltl and closure operators for security. In *PPDP '08: Proceedings of the 10th international ACM SIGPLAN conference on Principles and practice of declarative programming*, pages 8–19. ACM, 2008.

[Pac02] François Pachet. Playing with virtual musicians: the continuator in practice. *IEEE Multimedia*, 9:77–82, 2002.

[PAZ98] M. Puckette, T. Apel, and D. Zicarelli. Real-time audio analysis tools for Pd and MSP. In *Proceedings of the International Computer Music Conference.*, 1998.

[PR08] Jorge Pérez and Camilo Rueda. Non-determinism and probabilities in timed concurrent constraint programming. In Maria Garcia de la Banda and Enrico Pontelli, editors, *Logic Programming, 24th Internatinoal Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings*, volume 5366 of *Lecture Notes in Computer Science*, pages 677–681, pntcc probabilities nondeterminism 2008.

[RAD06] Camilo Rueda, Gérard Assayag, and Shlomo Dubnov. A concurrent constraints factor oracle model for music improvisation. In *XXXII Conferencia Latinoamericana de Informtica CLEI 2006*, Santiago, Chile, Aot 2006.

[RV04] C. Rueda and F. Valencia. On validity in modelization of musical problems by ccp. *Soft Comput.*, 8(9):641–648, 2004.

[Sar92] Vijay A. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1992.

[Sar08] Gerardo Sarria. *Formal Models of Timed Musical Processes.* PhD thesis, Universidad del Valle, Colombia, 2008.

[Seg06] Roberto Segala. Probability and nondeterminism in operational models of concurrency. In *CONCUR, LNCS*, pages 64–78. Springer, 2006.

[TB08] Mauricio Toro-Bermúdez. 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.

[TBAAR09] Mauricio Toro-Bermúdez, Carlos Agón, Gérard Assayag, and Camilo Rueda. Ntccrt: A concurrent constraint framework for signal processing languages. To be published in Proceedings of International Computer Music Conference (ICMC), August 2009.

[Val02] Frank D. Valencia. *Temporal Concurrent Constraint Programming.* PhD thesis, University of Aarhus, November 2002.

[VP08] Frank D. Valencia and Catuscia Palamidessi. Proc. of lix colloquium on emergent trends in concurrency theory. *Electron. Notes Theor. Comput. Sci.*, 209:1–4, 2008.