Article

Not peer-reviewed version

Mapping Petri Nets onto a Calculus of Context-Aware Ambients

François Siewe *, Vasileios Germanos, Wen Zeng

Posted Date: 23 April 2024

doi: 10.20944/preprints202404.1400.v1

Keywords: Calculus of Context-aware Ambients; CCA; Petri nets; dining cryptographers problem; experiments; simulation; ccaPL; formal methods

 doi:10.20944/preprints202404.1400.v1

Article

Mapping Petri Nets onto a Calculus of Context-Aware Ambients

François Siewe [1,*], Vasileios Germanos [1] and Wen Zeng [2]

[1] De Montfort University; vasileios.germanos@dmu.ac.uk; wen.zeng.wz@hotmail.com
[2] Shanghai Polytechnic University
* Correspondence: fsiewe@dmu.ac.uk

Abstract: Petri nets are a graphical notation for describing a class of discrete event dynamic systems whose behaviours are charaterised by concurrency, synchronisation, mutual exclusion and conflict. They have been used over the years for the modelling of various distributed systems applications. With the advent of pervasive systems and the Internet of Things, the Calculus of Context-aware Ambients (CCA) emerged as a suitable formal notation for analysing the behaviours of these systems. In this paper, we are interested in comparing the expressive power of Petri nets to that of CCA. That is, can the class of systems represented by Petri nets be modelled in CCA? To answer this question, an algorithm is proposed that maps any Petri net onto a CCA process. We show that a Petri net and its corresponding CCA process behave the same way through experiments. It follows that CCA is at least as expressive as Petri nets, i.e. any system that can be specified in Petri nets can also be specified in CCA. Moreover, tools developed for CCA can also be used to analyse Petri nets.

Keywords: Calculus of Context-aware Ambients; CCA; Petri nets; dining cryptographers problem; experiments; simulation; ccaPL; formal methods

1. Introduction

Over the years, a large number of mathematical formalisms for concurrency theory have been developed, each with its distinct features. Petri nets [1,2] were among the first formalisms for modelling interacting sequential processes. Then, Tony Hoare developed the calculus of Communicating Sequential Processes (CSP) [3], and Robin Milner the Calculus of Communicating Systems (CCS) [4]. These formalisms provided notations for modelling different forms of concurrency. Later on, the π-calculus was introduced to extend CCS with the notion of mobility. In the π-calculus, mobility is modelled using the mechanism of scope extrusion, i.e. a name sent through a receptor (i.e. a name that is used to receive messages) can also be used as a receptor. However, this makes the implementation of the π-calculus difficult in distributed systems. The join-calculus [5] was thus developed by Fournet and Gonthier to provide a formal basis for the design of distributed programming languages, by replacing the communication and mobility mechanisms of the π-calculus with the notion of *join definition*, which is much easier to implement in distributed systems. Recently, the Calculus of Context-aware Ambients (CCA) [6,7], inspired from the Calculus of Mobile Ambient (MA) [8], was developed to provide constructs for concurrency, mobility and context-awareness based on a single notion of *ambient*, which is an abstraction of a place where computation can happen. CCA emeges as a suitable formalism to reason about the behaviours of pervasive systems and the Internet of Things [9–11].

It is customary in formal methods to compare the expressive powers of formalisms, i.e. whether one formalism can be mapped onto another. In [12], Mennicke presents an operational Petri net semantics for the join-calculus. An approach to mapping Petri nets to concurrent programs in CC++ is proposed in [13], which establishes a link between Petri nets and object-oriented concurrent programming and forms a foundation for a Petri net based transformational software development methodology. Similarly, [14] proposes a mapping from Petri nets to the language of the B-Method in an effort to incorporate the Petri nets graphical notation in a software development method based on the B-language. A mapping of Petri nets to DEVS (Discrete Event System Specification) is presented in [15] so DEVS based platforms can be used to analyse systems specified in Petri nets.

 doi:10.20944/preprints202404.1400.v1

This paper proposes an approach to mapping Petri nets onto CCA. This allows for the graphical language of Petri nets be used in combination with CCA for the modelling of complex pervasive and IoT systems. The contributions of the paper is fourfold:

- We propose an algorithm that transforms any Petri net into a CCA process (Sect. 2.3). This demonstrates that CCA is at least as expressive as Petri nets, i.e. any system that can be specified in Petri nets can also be specified in CCA.
- We demonstrate that the proposed algorithm is efficient and scalable (Sect. 2.3.3). Indeed, the time complexity of the algorithm is quadratic (i.e. the execution time is in the order of the square of the size of the Petri net in input) and the size of the CCA process generated in output grows linearly with the size of the Petri net in input.
- The proposed algorithm is implemented in Python so a Petri net can be translated into a CCA process automatically at a click of a button (Sect. 2.3.3 and Appendix A).
- We show through experiments that a Petri net and its corresponding CCA process behave the same way using the CCA simulator ccaPL. The proposed approach is illustrated using a case study of the dining cryptographers problem (Sect. 3).

The remaining of the paper is structured as follows. Section 2 presents the proposed algorithm for mapping Petri nets onto CCA and discusses the efficiency and the scalability of the algorithm. The experimental results and the case study are presented in Sect. 3. In Sect. 4, we discuss the results, highlight the limitations of the work, and propose future work. Section 5 concludes the paper.

2. Materials and Methods

2.1. Overview of Petri Nets

Petri nets are a graphical formalism to describe systems whose dynamics are characterised by concurrency, synchronisation, mutual exclusion and conflict [2,16]. A Petri net consists of *places*, *transitions*, and *directed arcs*. A place is represented by a circle and a transition by a rectangle. Arcs run from a place to a transition or vice versa, never between places or between transitions. A place may contain a discrete number of marks called *tokens*. An example of Petri net is depicted in Figure 1. Any distribution of tokens over the places will represent a configuration of the net called a *marking*.

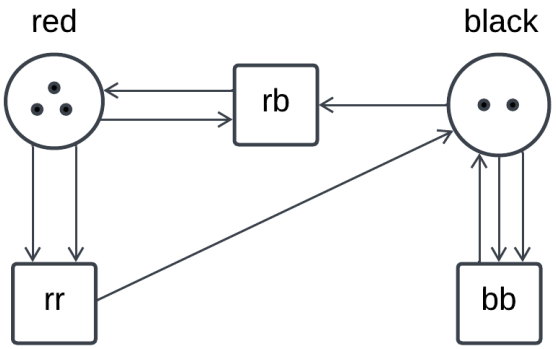


Figure 1. An example of Petri net [17]

The places from which an arc runs to a transition are called the *input places* of the transition; the places to which arcs run from a transition are called the *output places* of the transition. Similarly, the transitions from which an arc runs to a place are called the *input transitions* of the place; the transitions to which arcs run from a place are called the *output transitions* of the place. Therefore, a Petri net can be defined formally as a tuple (P, T, I, O, m_0), where

- P is a finite set of *places*.
- T is a finite set of *transitions*, such that $P \cup T \neq \emptyset$ and $P \cap T = \emptyset$.

 doi:10.20944/preprints202404.1400.v1

- $I : P \times T \to \mathbb{N}$ is an *input function*. $I(p,t)$ is the number of directed arcs from the place p to the transition t.
- $O : T \times P \to \mathbb{N}$ is an *output function*. $O(t,p)$ is the number of directed arcs from the transition t to the place p.
- $m_0 : P \to \mathbb{N}$ is the *initial marking*, which defines the initial number of tokens in each place.

Example 1. *The Petri net of Figure 1 is formally defined as follows:*

- $P = \{red, black\}$
- $T = \{bb, rb, rr\}$
- $I = \{(red, bb, 0), (red, rb, 1), (red, rr, 2), (black, bb, 2), (black, rb, 1), (black, rr, 0)\}$
- $O = \{(bb, red, 0), (rb, red, 1), (rr, red, 0), (bb, black, 1), (rb, black, 0), (rr, black, 1)\}$
- $m_0 = \{(red, 3), (black, 2)\}$

Transitions are the active components of a Petri net. A transition t is enabled if each of its input places p contains at least $I(p,t)$ tockens. A transition may execute if it is enabled. The execution of a transition t is atomic and consumes $I(p,t)$ tokens from each input place p, and creates $O(t,p)$ tokens in each output place p. Therefore, the execution of a transition t updates the marking of each place p connected to it as in (1), where $m(p)$ is the marking of a place p.

$$m(p) = m(p) - I(p,t) + O(t,p). \tag{1}$$

For example, Figure 2 shows the marking of the Petri net of Figure 1 after the transition rr is executed. The execution of Petri nets is non-deterministic: when multiple transitions are enabled at the same time, they will execute in any order.

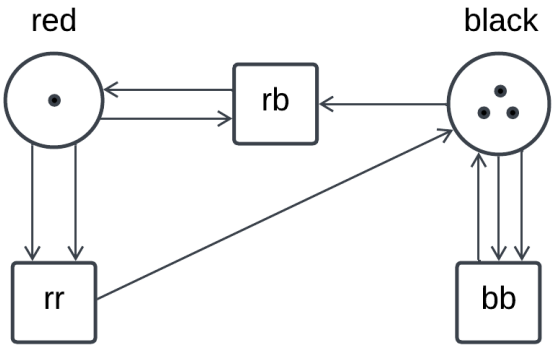


Figure 2. Petri net of Figure 1 after the execution of the transition rr

2.2. Overview of CCA

CCA [6,7] is a process calculus for specifying and reasoning about the behaviour of context-aware, mobile, and concurrent systems. Table 1 depicts the syntax of CCA, based on three syntactic categories: processes P (or Q), capabilities M, and context-expressions κ. The symbols n, x, y and z are names. Note that comments can be added anywhere in a specification using the prefix // for a single line comment or the pair /* and */ for a multiline comment.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 23 April 2024 doi:10.20944/preprints202404.1400.v1

Table 1. Syntax of CCA

P, Q ::=		**Processes**	κ ::=		**Context-expressions**
	$\mathbf{0}$	inactivity		$\mathbf{0}$	empty context
	$P \mid Q$	parallel composition		`true`	true
	$\{P\}$	block		`false`	false
	(`new` n) P	name restriction		$n = m$	name match
	$!P$	replication		`this`	hole
	$n[P]$	ambient		$n[\kappa]$	location context
	$< \kappa > M.P$	context-guarded prefix		$\kappa_1 \mid \kappa_2$	parallel composition
	`if` $< \kappa_1 > M_1.P_1$	if-then		κ_1 `and` κ_2	conjunction
	...			κ_1 `or` κ_2	disjunction
	$< \kappa_\ell > M_\ell.P_\ell$ `fi`			`not` κ	negation
	`if` $< \kappa_1 > M_1.P_1$	if-then-else		`next` κ	spatial next modality
	...			`somewhere` κ	somewhere modality
	$< \kappa_\ell > M_\ell.P_\ell$ `else` P `fi`				
	`let` $x_1 = e_1, \ldots, x_\ell = e_\ell$ `in` P	arithmetic			
	`find` $x_1, \ldots, x_\ell$: κ `for` P	search			
	`proc` $x(y_1, \ldots, y_\ell)$ P	process abstraction			
M ::=		**Capabilities**	α ::=		**Locations**
	`skip`	one transition		@	any parent
	`in` n	move into ambient n		n@	specific parent n
	`out`	move out of parent		#	any child
	`del` n	delete ambient n		n#	specific child n
	α `recv`$(y_1, \ldots, y_\ell)$	receive data from α		::	any sibling
	α `send`$(z_1, \ldots, z_\ell)$	send data to α		n ::	specific sibling n
	α $x(z_1, \ldots, z_\ell)$	process abstraction call		ϵ	locally

2.2.1. Processes

The process $\mathbf{0}$, aka *inactivity process*, does nothing and terminates immediately. The process $P \mid Q$ denotes the parallel composition of the processes P and Q. A process of the form $\{P\}$ behaves just like P. The process (`new` n) P creates a new name n and the scope of that name is limited to the process P. The replication $!P$ denotes a process which can always create a new parallel copy of P, i.e. $!P$ is equivalent to $P \mid !P$. The process $n[P]$ denotes an ambient named n whose behaviour is described by the process P. A context expression κ is a logical formula that specifies a property upon the state of the environment. A context-guarded prefix $< \kappa > M.P$ is a process that waits until the environment satisfies the context expression κ, then performs the capability M and continues like the process P. We let $M.P$ denote the process $<$ `true` $> M.P$. An if-then process `if` $< \kappa_1 > M_1.P_1 \ \ldots \ < \kappa_\ell > M_\ell.P_\ell$ `fi` waits until at least one of the context-expressions $(\kappa_i)_{1 \leq i \leq \ell}$ holds; then proceeds non-deterministically like one of the processes $< \kappa_j > M_j.P_j$ for which κ_j holds. An if-then-else process `if` $< \kappa_1 > M_1.P_1 \ \ldots \ < \kappa_\ell > M_\ell.P_\ell$ `else` P `fi` behaves like an if-then process, but does not wait and continues like the process P if none of the branches can be executed. A process `let` $x_1 = e_1, \ldots, y_\ell = e_\ell$ `in` P behaves like the process P in which each occurrence of x_i is substituted to the value of the arithmetic expression e_i, for $1 \leq i \leq \ell$ and $\ell \geq 1$. A search process `find` $x_1, \ldots, x_\ell$: κ `for` P looks for a list of names $n_1, \ldots, n_\ell$ in the context such that the context-expression κ in which each occurrence of x_i is replaced by n_i holds, and continues like the process P in which each occurrence of x_i is replaced by n_i, $1 \leq i \leq \ell$ and $\ell \geq 1$. A process of the form `proc` $x(y_1, \ldots, y_\ell)$ P defines a process abstraction named x, whose behaviour is described by the process P. The names $y_1, \ldots, y_\ell$ are the formal parameters and the process abstraction, $\ell \geq 0$. A process abstraction is a mechanism to give a name say x to a process P and later use that name anywhere to refer to the process P, just the same way functions and procedures are used in programming languages.

 doi:10.20944/preprints202404.1400.v1

2.2.2. Capabilities

A capability is an elementary action that an ambient can perform. The capability `skip` represents one transition, i.e. one execution step. An ambient can move into a sibling ambient n by performing the capability `in` n; and move out of its parent ambient by executing the capability `out`. An ambient can exchange messages with another ambient using the output capability α `send`$(z_1,\ldots,z_\ell)$ to send a list of names $z_1,\ldots,z_\ell$ to a location α, and the input capability α `recv`$(y_1,\ldots,y_\ell)$ to receive a list of names from a location α into the variables $y_1,\ldots,y_\ell$, for some $\ell \geq 0$. The location α can be '@' to mean any parent, 'n@' to mean a specific parent n, '#' to mean any child ambient, 'n#' to mean a specific child n, '::' to mean any sibling, 'n ::' to mean a specific sibling n, or ϵ (empty string) to mean the executing ambient itself. A capability `del` n deletes an empty child ambient n (i.e. $n[\mathbf{0}]$). A capabilty of the form $\alpha\ x(z_1,\ldots,z_\ell)$ calls the process abstraction x defined at the location α and the names $z_1,\ldots,z_\ell$ are the actual parameters, $\ell \geq 0$.

2.2.3. Context Model

In CCA, a context is modelled as a process with possibly a single hole in it. The hole (denoted by $\odot$) in a context C represents the position of the process, which C is a context. For example, suppose a system is modelled by the process $P \mid n[Q \mid m[R \mid S]]$. So, the context of the process R in that system is $P \mid n[Q \mid m[\odot \mid S]]$, and that of the ambient named m is $P \mid n[Q \mid \odot]$. Thus the context of a `CCA` process is described by the grammar in Table 2. A context-expression (CE, for short) is a formula representing some property over context.

Table 2. Syntax of contexts

C	::=	$\mathbf{0} \mid \odot \mid n[C] \mid C \mid P \mid (\texttt{new}\ n)\ C$

2.2.4. Context-Expressions

The formal semantics of context-expressions (CEs) with respect to the context model of Table 2 is given in Table 3, where the notation $C \models \kappa$ means that the context C satisfies the context-expression κ. We also write $\models \kappa$ to mean that a context-expression κ is *valid*, i.e. κ is satisfied by all context.

Table 3. Satisfaction relation for context expressions

	$\models$	`true`		
C	$\models$	$n = m$	iff	$n = m$
C	$\models$	$\mathbf{0}$	iff	$C = \mathbf{0}$
C	$\models$	`this`	iff	$C = \odot$
C	$\models$	`not` κ	iff	$C \not\models \kappa$
C	$\models$	$\kappa_1 \mid \kappa_2$	iff	exist C_1, C_2 such that $C = C_1 \mid C_2$ and $C_1 \models \kappa_1$ and $C_2 \models \kappa_2$
C	$\models$	κ_1 `and` κ_2	iff	$C \models \kappa_1$ and $C \models \kappa_2$
C	$\models$	$n[\kappa]$	iff	exists C' such that $C = n[C']$ and $C' \models \kappa$
C	$\models$	`next` κ	iff	exist C', n such that $C = n[C']$ and $C' \models \kappa$
C	$\models$	`somewhere` κ	iff	$C \models \kappa$ or exist C', n such that $C = n[C']$ and $C' \models$ `somewhere` κ

The CE `true` holds for all context while the CE `false` holds for no context. A CE $n = m$ holds if the names n and m are identical. The CE $\mathbf{0}$ holds for the empty context $\mathbf{0}$. The CE `this` holds solely for the hole context, i.e. the position of the process evaluating that context expression. Propositional operators such as `not`, `and` and `or` expand their usual semantics to context expressions. A CE $\kappa_1 \mid \kappa_2$ holds for a context if that context is a parallel composition of two contexts such that κ_1 holds for one and κ_2 holds for the other. A CE $n[\kappa]$ holds for a context if that context is an ambient named n such that κ holds inside that ambient. A CE `next` κ holds for a context if that context has a child context for which κ holds. A CE `somewhere` κ holds for a context if there exists somewhere in that context a

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 23 April 2024 doi:10.20944/preprints202404.1400.v1

sub-context for which κ holds. Some examples of context-expressions that are used later in this paper are given in Table 4.

Table 4. Examples of context expressions

$has(n)$	=	$\texttt{somewhere}(\texttt{this} \mid n[\texttt{true}] \mid \texttt{true})$	n is located at self.
$at(n)$	=	$\texttt{somewhere}(n[\texttt{next}(\texttt{this} \mid \texttt{true})] \mid \texttt{true})$	self is located at n.
$at(n,m)$	=	$\texttt{somewhere}(n[m[\texttt{true}] \mid \texttt{true}] \mid \texttt{true})$	m is located at n.
$with(n)$	=	$\texttt{somewhere}(n[\texttt{true}] \mid \texttt{next}(\texttt{this} \mid \texttt{true}) \mid \texttt{true})$	self is with n.
$with(n,m)$	=	$\texttt{somewhere}(n[\texttt{true}] \mid m[\texttt{true}] \mid \texttt{true})$	n is with m.
$state(p,x)$	=	$\texttt{somewhere}(p[x[0] \mid \texttt{true}] \mid \texttt{true})$	the current state of p is x.
$lockOn()$	=	$\texttt{somewhere}(lock[on[0] \mid \texttt{true}] \mid \texttt{true})$	the lock is on.

Context-expressions are used in CCA to specified context-aware processes. We recall in Table 5 the formal semantics of context-aware processes, where σ is a substitution of names and $\longrightarrow$ is the reduction relation of processes. The complete formal semantics of CCA can be find in [6,7]. The next section presents an algorithm for generating from a Petri net a CCA process that behaves in a similar manner.

Table 5. Reduction relation for context-aware processes

(R1) $C(M.P) \longrightarrow C'(P\sigma) \quad \Rightarrow \quad C(<\kappa> M.P) \longrightarrow C'(P\sigma) \quad \text{if } C \models \kappa$

(R2) $C(<\kappa_i> M_i.P_i) \longrightarrow C'(P_i\sigma)$
$\Rightarrow \quad C(\texttt{if } <\kappa_1> M_1.P_1 \ \ldots \ <\kappa_\ell> M_\ell.P_\ell \ \texttt{fi}) \longrightarrow C'(P_i\sigma)$, for some $i, 1 \leq i \leq \ell$.

(R3) $C(\texttt{if } <\kappa_1> M_1.P_1 \ \ldots \ <\kappa_\ell> M_\ell.P_\ell \ \texttt{fi}) \longrightarrow C'(P_i\sigma)$
$\Rightarrow \quad C(\texttt{if } <\kappa_1> M_1.P_1 \ \ldots \ <\kappa_\ell> M_\ell.P_\ell \ \texttt{else } P \ \texttt{fi}) \longrightarrow C'(P_i\sigma)$

(R4) $C(\texttt{if } <\kappa_1> M_1.P_1 \ \ldots \ <\kappa_\ell> M_\ell.P_\ell \ \texttt{else } P \ \texttt{fi}) \longrightarrow C(P)$
if $\forall i \in [1,\ell] \ \nexists (C',\sigma)$ such that $C(<\kappa_i> M_i.P_i) \longrightarrow C'(P_i\sigma)$.

(R5) $C(\texttt{find } x_1,\ldots,x_\ell \ : \ \kappa \ \texttt{for } P) \longrightarrow C(P\{x_1 \leftarrow n_1 \ldots x_\ell \leftarrow n_\ell\})$
if $C \models \kappa\{x_1 \leftarrow n_1 \ldots x_\ell \leftarrow n_\ell\}$

2.3. An Algorithm for Mapping a Petri Net onto a CCA Process

Algorithm 1 translates a Petri net (see Sect. 2.1) into a CCA process. The algorithm takes in input a Petri net (P,T,I,O,m_0) and returns in output a CCA process in the variable *cca_str*. The initial marking m_0 assigns to each place a number of tokens. The execution of a transition updates the number of tokens in the input places and the output places of the transition according to (1). A semaphore is used to guarantee that the execution of a transition is atomic, i.e. at most one transition can be executed at a time. In order to execute, a transition must obtain the semaphore, otherwise the transition must wait until the semaphore is released by another transition. This semaphore is described in CCA by the ambient `lock` defined in (2). The semaphore is in the *obtained* state if the ambient contains a child ambient `on[0]` and is in the *released* state otherwise.

```
lock[
 ! recv(). :: recv(t).{on[0] | t :: recv().del on.send().0}
 | send().0
]
```
(2)

The behaviour of the ambient `lock` can be explained as follows. Initially, the semaphore is in the released state, i.e. no transition is being executed. The ambient waits (using the capability :: `recv(t)`) until a transition t is willing to obtain the semaphore, then creates a child ambient `on[0]` to indicate that the semaphore is obtained by the transition t. The semaphore remains in the obtained state until it

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 23 April 2024 doi:10.20944/preprints202404.1400.v1

is released by the transition t (using the capability `t :: recv()`), in which case the child ambient `on[0]` is deleted. The context-expression `lockOn()` defined in Table 4 holds if the semaphore `lock` is in the obtained state, i.e. it contains a child ambient `on[0]`.

In Algorithm 1, the declaration of the context expression `lockOn()` is created in line 2, together with other simulation directives such as the execution mode and the simulation length. In ccaPL, these declarations appear between the keywords "BEGIN_DECLS" and "END_DECLS". The semaphore `lock` is created in line 3 of Algorithm 1. The following subsections explain how places and transitions are modelled in CCA.

Algorithm 1: Mapping a Petri net onto a CCA process

input : A Petri net: $N = (P, T, I, O, m_0)$
output: A CCA process: *cca_str*

1 $Space$ = " ";
2 *cca_str* = "BEGIN_DECLS\n" + $Space$ + "def lockOn() = { somewhere (lock[on[0] | true] | true) }\n" + $Space$ + "def state(p,x) = { somewhere (p[x[0] | true] | true) }\n" + $Space$ + "//display code\n" + $Space$ + "//display congruence\n" + $Space$ + "mode random\n" + $Space$ + "length=100\n END_DECLS\n";
3 *cca_str* = *cca_str* + "lock[\n" + $Space$ + "! recv().::recv(t).{ on[0] | t::recv(x).del on.send().0 }\n" + $Space$ + "| send().0\n]\n";
4 // Generate the place ambients
5 **for** $p \in P$ **do**
6 $xx = 1000 + m_0(p)$;
7 *cca_str* = *cca_str* + " | \n" + p + "[\n send(" + $m_0(p)$+").0\n" + " | !recv(n).let zz=_+(1000+n) in ::recv(v).del zz.let w=n+v, y=_+(1000+n+v) in send(w).::send().y[0]\n" + $Space$ + "| _" + xx + "[0]\n] \n";
8 // Generate the transition ambients
9 **for** $t \in T$ **do**
10 *var* = "";
11 *cond* = "";
12 *trouve* = False;
13 **foreach** $p \in P$ *such that* $I(p,t) \neq 0$ **do**
14 $val = 1000 + I(p,t)$;
15 **if** *not trouve* **then**
16 *cond* = *cond* + "_M_" + p + ">=_" + *val*;
17 *trouve* = True;
18 **else**
19 *cond* = *cond* + " and _M_" + p + ">=_" + *val*;
20 *var* = *var* + "find _M_" + p + ": state(" + p + ",_M_" + p + ") for ";
21 *cca_str* = *cca_str* + " | \n" + t +"[\n" + $Space$ + "! < not lockOn() > lock::send(" + t + ")." + *var* + "if \n" + $Space$ + $Space$ + "< " + *cond* + " >"
22 **foreach** $p \in P$ *such that* $I(p,t) \neq 0$ *or* $O(t,p) \neq 0$ **do**
23 $val = -I(p,t) + O(t,p)$;
24 *cca_str* = *cca_str* + p + "::send(" + *val* + ")." + p + "::recv().';
25 *cca_str* = *cca_str* + "lock::send(end).0\n" + $Space$ + $Space$ + "else lock::send(not_enabled).0\n" + $Space$ + $Space$ + "fi.0\n]\n";

2.3.1. Modelling Places

In this section, we show how a place can be modelled as an ambient in CCA. For each place $p \in P$ an ambient of the same name is created in the line 7 of Algorithm 1. The general form of this ambient is given in Table 6. Initially, a place p contains $m_0(p) = x$ tokens. This is represented in Table 6 by the process "`send(x).0`". A child ambient to the ambient p is also created as "`_(1000 + x)[0]`", e.g. if `x = 1` then the child ambient is "`_1001[0]`". This encoding reflects in the child ambient's name the number of

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 23 April 2024 doi:10.20944/preprints202404.1400.v1

token in the ambient p and satisfies the property stated in Theorem 1. Therefore, the names of the child ambients of places can be compared instead of the number of tokens of these places. The behaviour of a place ambient p can be summarised as follows. A place ambient (see Table 6) waits until a transition it is connected to starts executing (see "`:: recv(v)`"). It then deletes its current child ambient, updates its number of tokens according to (1) and creates a child ambient that reflects its new number of tokens. It also signals to the running transition that it has completed updating (see "`:: send()`"). For example, the place ambients for the places *red* and *black* of the Petri net in Figure 1 are given in Table 7. The loop in line 5 of Algorithm 1 creates such a place ambient for each place $p \in P$ and composes them in parallel.

Table 6. General form of a place ambient

```
p[
   send(x).0
   | !recv(n).let zz=_+(1000+n) in ::recv(v).del zz.
     let w=n+v, y=_+(1000+n+v) in send(w).::send().y[0]
   | _(1000 + x)[0]
]
```

Table 7. The place ambients for the places *red* and *black* of Figure 1

```
red[
   send(3).0
   | !recv(n).let zz=_+(1000+n) in ::recv(v).del zz.
     let w= n+v, y=_+(1000+n+v) in send(w).::send().y[0]
   | _1003[0]
]

black[
   send(2).0
   | !recv(n).let zz=_+(1000+n) in ::recv(v).del zz.
     let w= n+v, y=_+(1000+n+v) in send(w).::send().y[0]
   | _1002[0]
]
```

Definition 1. *Let $str(z)$ be the string representation of an integer value z. For example $str(0)$ ="0" and $str(23)$ ="23".*

Theorem 1. *Let n, x and y be 3 integers such that $n \geq 1$, $0 \leq x < 10^n$, and $0 \leq y < 10^n$. Then*

$$x \geq y \Leftrightarrow str(10^n + x) \geq str(10^n + y).$$

The proof of Theorem 1 can be done by induction on n as follows.

Proof. Let $\|z\|$ denote the number of digit in a non-negative integer z. For example $\|0\| = 1$, $\|23\| = 2$, and $\|100\| = 3$.

Base case:

Prove that the theorem holds for $n = 1$.

We have: $0 \leq x < 10 \Rightarrow \|x\| = 1$. Similarly $\|y\| = 1$. Therefore $x \geq y \Rightarrow str(10 + x)$ ="1x" $\geq str(10 + y)$ ="1y".

Reversely, "1x" $\geq$"1y" $\Rightarrow$ "x" $\geq$ "y". And since $\|x\| = 1$ and $\|y\| = 1$, it follows that $x \geq y$.

 doi:10.20944/preprints202404.1400.v1

Induction case:

Suppose that the theorem holds for n and prove that the theorem also holds for $n+1$.

- If $0 \leq x < 10^n$, and $0 \leq y < 10^n$, then $0 \leq x < 10^{n+1}$, and $0 \leq y < 10^{n+1}$. Let $str(10^n + x)$ ="$1\overline{x}$" and $str(10^n + y)$ ="$1\overline{y}$", for some strings $\overline{x}$ and $\overline{y}$. It follows that $str(10^{n+1} + x)$ ="$10\overline{x}$" and $str(10^{n+1} + y)$ ="$10\overline{y}$". In string comparison, "$1\overline{x}$" $\geq$ "$1\overline{y}$" $\Leftrightarrow$ "$10\overline{x}$"$\geq$ "$10\overline{y}$". We conclude that $x \geq y \Leftrightarrow str(10^{n+1} + x) \geq str(10^{n+1} + y)$.
- If $10^n \leq x < 10^{n+1}$, and $0 \leq y < 10^n$, then $\|x\| \geq \|y\| + 1$. Let $str(10^{n+1} + x)$ ="$1\sigma\overline{x}$" and $str(10^{n+1} + y)$ ="$10\overline{y}$", for some digit $\sigma \geq 1$ and strings $\overline{x}$ and $\overline{y}$ of equal length n. Since $\sigma \geq 1$, "$1\sigma\overline{x}$" $\geq$ "$10\overline{y}$" is always true. Therefore we conclude that $x \geq y \Leftrightarrow str(10^{n+1} + x) \geq str(10^{n+1} + y)$.
- If $10^n \leq x < 10^{n+1}$, and $10^n \leq y < 10^{n+1}$, then let $x = \sigma \times 10^n + x'$ and $y = \theta \times 10^n + y'$, for some integers $x' < 10^n$ and $y' < 10^n$, and digits $\sigma \neq 0$ and $\theta \neq 0$. There are 2 cases: $\sigma = \theta$ or $\sigma \neq \theta$.
  - If $\sigma = \theta$, then $x \geq y \Leftrightarrow (\sigma \times 10^n + x' \geq \sigma \times 10^n + y') \Leftrightarrow x' \geq y' \Leftrightarrow str(10^n + x') \geq str(10^n + y') \Leftrightarrow str(\sigma \times 10^n + x') \geq str(\sigma \times 10^n + y') \Leftrightarrow str(10^{n+1} + \sigma \times 10^n + x') \geq str(10^{n+1} + \sigma \times 10^n + y') \Leftrightarrow str(10^{n+1} + x) \geq str(10^{n+1} + y)$.
  - If $\sigma \neq \theta$, let $str(10^{n+1} + x)$ = " $1\sigma\overline{x'}$" and $str(10^{n+1} + y)$ = "$1\theta\overline{y'}$", for some strings $\overline{x'}$ and $\overline{y'}$. It follows that "$1\sigma\overline{x'}$" > "$1\theta\overline{y'}$" $\Leftrightarrow \sigma > \theta \Leftrightarrow \sigma \times 10^n > \theta \times 10^n \Leftrightarrow \sigma \times 10^n + x' > \theta \times 10^n + y' \Leftrightarrow x > y$.

Therefore, we conclude that $x \geq y \Leftrightarrow str(10^{n+1} + x) \geq str(10^{n+1} + y)$.

□

2.3.2. Modelling Transitions

Similarly to a place, a transition can be modelled as an ambient. Indeed, Algorithm 1, in the lines 9-25, creates for each transition $\mathtt{t} \in T$ an ambient named t of the form described in Table 8. The ambient uses the context-expression $state(p, x)$ defined in Table 4 to check if a place ambient p has a child ambient x. Note that the name of a child ambient of a place is encoded to reflect the number of tokens of that place and satisfies the property stated in Theorem 1. In Table 8, the variables p1, ..., pn denote the input places of the transition and q1, ..., qk denote the input and output places of the transition. The behaviour of the ambient is an iterative process, which waits until the semaphore `lock` is released, then obtains the semaphore `lock` (line 2 in Table 8). It then reads the markings of the transition's input places (line 3) and checks if all the input places are enabled (line 4), in which case the number of tokens of each place connected to it is updated according to (1) (line 5). Finally, the process releases the semaphore `lock` and gets ready for further execution (lines 6 and 7). An example of transition ambient is given in Table 9 for the transition *rr* of the Petri net in Figure 1.

Table 8. General form of a transition ambient

```
1. t[
2.   !< not lockOn() >lock::send(t).
3.   find _M_p1: state(p1,_M_p1) for ...
     find _M_pn: state(pn,_M_pn) for if
4.     <_M_p1 >=_(1000 + I(p1,t)) and ... and _M_pn >=_(1000 + I(pn,t)) >
5.     q1::send(-I(q1,t)+O(t,q1)).q1::recv(). ...
        .qk::send(-I(qk,t)+O(t,qk)).qk::recv().
6.     lock::send(end).0
7.     else lock::send(not_enabled).0
8.     fi.0
9. ]
```

 doi:10.20944/preprints202404.1400.v1

Table 9. The transition ambient for the transition *rr* of Figure 1

```
rr[
   !< not lockOn() >lock::send(rr).
   find _M_red: state(red,_M_red) for if
      < _M_red>=_1002 > red::send(-2).red::recv().
      black::send(1).black::recv().lock::send(end).0
      else lock::send(not_enabled).0
      fi.0
]
```

In summary, the CCA process generated by Algorithm 1 for a Petri net is the parallel composition of the semaphore `lock`, all the place ambients and all the transition ambients. For example, the CCA process generated for the Petri net of Figure 1 is presented in Table 10.

Table 10. The CCA process genertaed by Algorithm 1 for the Petri net in Figure 1

```
BEGIN_DECLS
   def lockOn() = { somewhere (lock[on[0] | true] | true) }
   def state(p,x) = { somewhere (p[x[0] | true] | true) }
   //display code
   //display congruence
   mode random
   length=100
END_DECLS
lock[
   ! recv().::recv(t).{ on[0] | t::recv(x).del on.send().0 }
   | send().0
]
|
red[
   send(3).0
   | !recv(n).let zz=_+(1000+n) in ::recv(v).del zz.let w= n+v,
     y=_+(1000+n+v) in send(w).::send().y[0]
   | _1003[0]
]
|
black[
   send(2).0
   | !recv(n).let zz=_+(1000+n) in ::recv(v).del zz.let w= n+v,
     y=_+(1000+n+v) in send(w).::send().y[0]
   | _1002[0]
]
|
rb[
   !< not lockOn() >lock::send(rb).find _M_red: state(red,_M_red) for
   find _M_black: state(black,_M_black) for if
      < _M_red>=_1001 and _M_black>=_1001 > red::send(0).red::recv().
      black::send(-1).black::recv().lock::send(end).0
      else lock::send(not_enabled).0
      fi.0
]
|
rr[
   !< not lockOn() >lock::send(rr).find _M_red: state(red,_M_red) for if
      < _M_red>=_1002 > red::send(-2).red::recv().black::send(1).
      black::recv().lock::send(end).0
      else lock::send(not_enabled).0
      fi.0
]
|
bb[
   !< not lockOn() >lock::send(bb).find _M_black: state(black,_M_black) for if
      < _M_black>=_1002 > black::send(-1).black::recv().lock::send(end).0
      else lock::send(not_enabled).0
      fi.0
]
```

 doi:10.20944/preprints202404.1400.v1

2.3.3. Complexity of the Algorithm

It is important to discuss the complexity of the algorithm in terms of the execution time, and the size of the CCA process produced in output in comparison to the size of the Petri net received in input. For a Petri net of n places and m transitions, Algorithm 1 takes $O(n)$ time to generate the place ambients (see the loop in line 5) and $O(nm)$ time to generate the transition ambients (see the loop in line 9). Therefore the overall time complexity of Algorithm 1 is $O(\tilde{n}^2)$, where $\tilde{n} = max(n, m)$; i.e. the execution time is in the order of the square of the size of the input. Besides, the algorithm creates for each place two ambients (see lin 5): a place ambient with a child ambient as depicted in Table 6. However, a single ambient is created for each transition in line 9 of Algorithm 1. The general form of a transition ambient is depicted in Table 8. Finally, a unique semaphore ambient named *lock* (see (2)) is created in line 3. Thus, the total number of ambients created is $2 \times n + m + 1$, and so the size of the CCA process generated by Algorithm 1 is in the order of $O(\tilde{n})$, i.e. the size of the CCA process generated in output grows linearly with the size of the Petri net in input. The proposed algorithm is thus efficient and scalable. An implementation of the algorithm in Python (version 3.10.11) is given in Appendix A so the mapping of a Petri net onto a CCA process can be done automatically at a click of a button.

3. Results

In this section, the proposed algorithm is validated through experiments using a case study of the dining cryptographers problem. The experiments are carried out using the CCA simulator ccaPL.

3.1. Overview of the CCA Simulator ccaPL

The syntax of a ccaPL program is given in Table 11, where `P` is a process and `k` is a context-expression defined in Table 1; `<Id>` stands for an identifier (i.e. a name), `e` for empty string, and `<Val>` is a non-negative integer number. It follows that a ccaPL program is composed of a declaration

Table 11. Syntax of a ccaPL program

```
<Program> ::= <DeclarationBlock> P
<DeclarationBlock> ::= e | ‘‘BEGIN_DECLS’’ <DeclarationList> ‘‘END_DECLS’’
<DeclarationList> ::= e | <Declaration><DeclarationList>
<Declaration> ::= ‘‘def’’ <Id> ‘‘(’’ <ParamList> ‘‘) = {’’  k  ‘‘}’’
                | ‘‘mode random’’ | ‘‘display code’’ | ‘‘length = ’’ <Val>
                | ‘‘display congruence’’
<ParamList> ::= e | <Id> <Params>
<Params> ::= e | ‘‘,’’ <Id> <Params>
```

block (optional) and a body, which is a CCA process. The declaration block starts with the keyword `BEGIN_DECLS` and ends with the keyword `END_DECLS`. In between these keywords, one can add the definitions of context-expressions using the keyword `def`, and the declarations of *execution directives*. Note that ccaPL programs are case-sensitive. An example of ccaPL program is given in Table 10. The execution directives control how the parallel processes of a program are executed. By default, at each execution step the process to be executed is chosen deterministically based on two criteria: how long the process has been willing to execute (FIFO[1]), and in case of conflict the sequential order as they appear in the program text is used. The execution directive `mode random` changes the execution mode to a random selection of the processes to be executed. The directive `display code` forces the program code to be displayed after each execution step. By default, only reduction steps are shown in the execution traces; with the directive `display congruence`, the congruence steps are also shown

[1] First in First out

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 23 April 2024 doi:10.20944/preprints202404.1400.v1

in the execution traces. The directive `length = xx` stops the execution of the program after `xx` steps. Comments can be added anywhere in a program text using the prefix `//` for a single line comment or the pair `/*` and `*/` for a multiline comment, just like in the Java programming language. The ccaPL tool can generates three types of execution output as shown in Figure 3: a textual execution trace, a communication graph, and a behaviour graph.

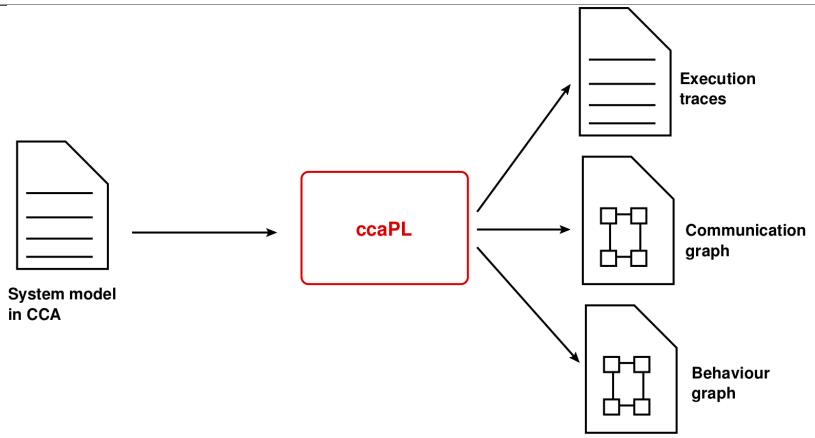


Figure 3. The ccaPL tool

3.1.1. Textual Execution Trace

The execution trace is a text describing the execution steps. An example of textual execution trace is given in Figure 4. Each execution step trace is prefixed by the symbol `-->` (resp. `<-->`) for a reduction step (resp. for a congruence step), followed by the explanation of the execution step between a pair of curly brackets `{` and `}`. For example, the explanation of a message passing step has the form `{child to parent: A ===(X)===> B}`, meaning that a message `X` is sent by a child ambient `A` to a parent ambient `B`. Notations such as `Child to parent`, `Parent to child`, `Sibling to sibling`, and `local` provide information about the relationship between the sender `A` and the receiver `B`. In particular, `local` means the sender is the receiver (i.e. `A` and `B` are the same ambient). An explanation of the form `{binding: n -> X}` corresponds to the execution of a statement of the form `find n:k for P` and means that the value (i.e. name) `X` has been found for the variable `n` such that the context-expression `k` holds in the current context. The variable `n` will then be replaced by the name `X` in the process `P` (see the semantic rule R5 in Table 5). The remaining execution step explanations are straightforward. To generate a textual execution trace, use the following command line, where `myprog.cca` is your program file.

```
java -jar ccaPL.jar -e myprog.cca
```

```
1. --->  {Local: ac_server ===()===> ac_server}
2. --->  {Local: RFID_reader ===()===> RFID_reader}
3. --->  {Local: door ===()===> door}
4. --->  {ambient "RFID_tag" moves into ambient "RFID_reader"}
5. --->  {Child to parent: RFID_tag ===(166)===> RFID_reader}
6. --->  {Sibling to sibling: RFID_reader ===(RFID_reader,ac_service,166)===> ac_server}
7. --->  {local call to the abstraction "ac_service" in the ambient "ac_server"}
8. --->  {Sibling to sibling: ac_server ===(valid)===> RFID_reader}
9. --->  {Sibling to sibling: ac_server ===(open)===> door}
10. --->  {Local: RFID_reader ===()===> RFID_reader}
```

Figure 4. Textual execution trace

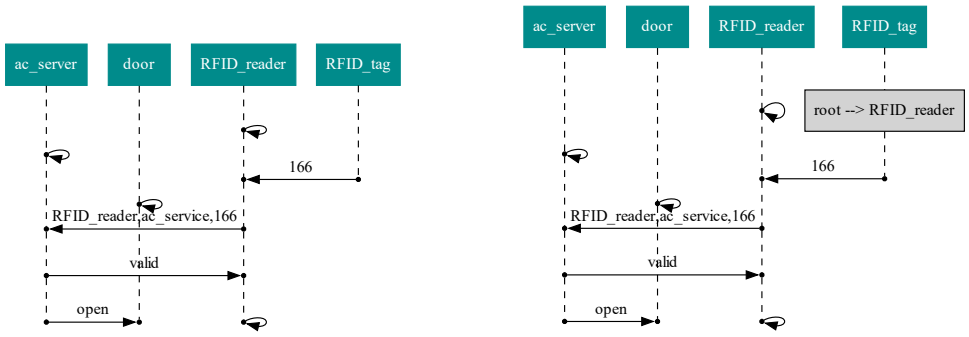
Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 23 April 2024 doi:10.20944/preprints202404.1400.v1

3.1.2. Communication Graph

The execution trace is a diagram showing the timeline of the communications between the ambients. An example of communication graph is given in Figure 5-a. The top row of the diagram is the list of the ambients being executed. The execution timeline of each ambient is denoted by a vertical dashed line, and the time increases from top to bottom. An arrow from one timeline (sender ambient) to another (receiver ambient) indicates a communication step between the corresponding ambients. This arrow is labelled with the message exchanged. A communication graph is created using the following command line, where `XXX` is the image format like ps, jpg, png, pdf, and so on.

```
java -jar ccaPL.jar -gXXX myprog.cca
```

The generated graph is stored in the file `myprog.XXX`.



(a) Communication graph (b) Behaviour graph

Figure 5. Graphical execution traces

3.1.3. Behaviour Graph

This is similar to a communication graph, but in addition it shows the movement steps using a grey box containing a text of the form `A --> B` on the timeline of an ambient to indicate that the ambient has moved from the location `A` to the location `B`. An example of behaviour graph is given in Figure 5-b. Recall that an ambient can move from one location to another by performing the capability `in` or the capability `out` (see Sect. 2.2). The following command line will generate a communication graph `myprog_0.XXX` and the corresponding behaviour graph `myprog_1.XXX`.

```
java -jar ccaPL.jar -gxXXX myprog.cca
```

Note that the generation of the graphical execution traces requires that the *Graphviz* package [18] be installed on your computer.

3.2. A Case Study: The Dining Cryptographers Problem

The standard dining cryptographers problem [19] consists of three diners and requires that the identity of the person who pays the bill (which may be one of the cryptographers or an external person) remains anonymous. In this section we consider a simplified version of the problem with just two diners. This version is also used in [20,21] and can be extended to three or more diners. Alice and Bob are two cryptographers who have a dinner in a restaurant. When it is time for the bill, they are informed by the waiter that the bill has already been paid. Both, Alice and Bob, would like to know whether the bill was paid by a third person, or it was one of them. However, if it is the second case, then they do not want an eavesdropper, Yves, on a neighbouring table to know which of them paid. Following is the protocol that they decided to use to solve this problem.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 23 April 2024 doi:10.20944/preprints202404.1400.v1

3.2.1. A Dining Cryptographers Protocol

Firstly, they toss two coins that are visible to both of them. At the same time, they ensure that Yves cannot see either of them. If Alice paid, she lies about the parity of the two coins i.e. she calls 'agree' if she sees a head and a tail, and 'disagree' otherwise. If Alice did not pay, she tells the truth about the parity of the coins. The same applies for Bob. Now Alice and Bob both know whether one of them paid. In case their calls are the same they know that a third person paid, otherwise it must have been one of them – in this example they actually both know which. On the other hand, Yves can only tell whether or not one of Alice and Bob paid, but not which one. It should be noted that Yves also knows about the protocol. A possible encoding of the protocol using a Petri net is depicted in Figure 6. The two places at the left of the net represent Alice's initial state: having paid is shown by placing a single token in place AP, and having not paid is shown by placing a single token in place $A\neg P$. The initial state of Bob is represented by the places at the right. The three possible initial markings for Alice and Bob are given in (3).

$$\{AP, B\neg P\}, \{A\neg P, BP\}, \{A\neg P, B\neg P\} \tag{3}$$

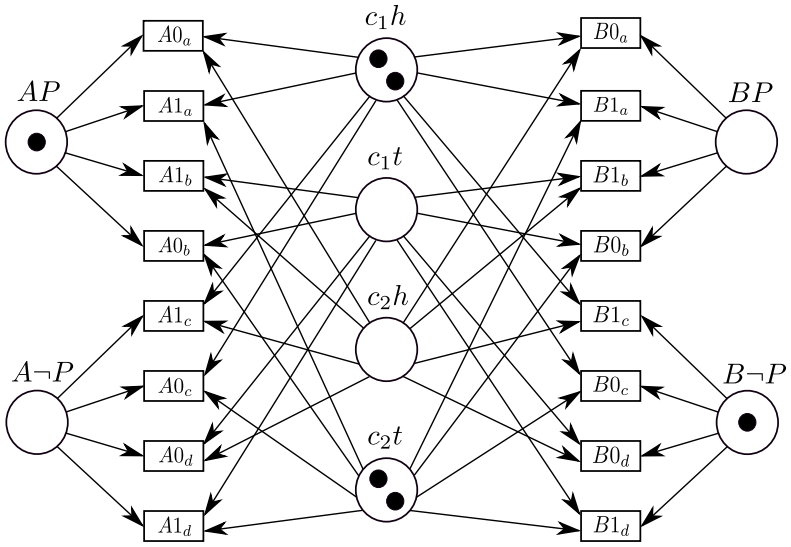


Figure 6. A Petri net representing the dining cryptographers protocol [21]

The top two places in the centre of the net represent the first coin: head is represented by placing two tokens in the place c_1h, and tail is represented by placing two tokens in the place c_1t. The bottom two places, c_2h and c_2t, represent the second coin. As it was mentioned, both Alice and Bob must see the coins. For this reason, the marked places must contain two tokens. As a result, the possible initial markings for the coins are the multi-sets in (4).

$$\begin{gathered}\{c_1h, c_1h, c_2h, c_2h\}, \{c_1h, c_1h, c_2t, c_2t\}, \\ \{c_1t, c_1t, c_2h, c_2h\}, \{c_1t, c_1t, c_2t, c_2t\}\end{gathered} \tag{4}$$

The cross product of the cryptographer markings in (3) and the coin markings in (4) denotes the set of all 12 possible initial markings. The eight transitions on the right represent the eight possible scenarios for Bob, given by two possibilities for each coin multiplied by the two possibilities for his own initial state. The transitions on the right represent Bob saying the coins 'disagree' ($B0$) or Bob saying the coins 'agree' ($B1$). Similarly for Alice on the left.

 doi:10.20944/preprints202404.1400.v1

3.2.2. Mapping the Petri Net of the Dining Cryptographers Protocol onto a CCA Process

Algorithm 1 was applied to map the Petri net in Figure 6 onto a CCA process. In this section it is shown that the CCA process and the Petri net behave the same way. To achieve this, the CCA process is executed in ccaPL for each of the 12 possible initial markings. The result is summarised in Table 12. It follows that the transitions enabled in the Petri net correspond exactly to the transition ambients executed by the CCA process. A sample of the execution traces, in the form of a communication graph, is given in Figure 7. This corresponds to the simulation of the case where Alice paid the bill and Bob did not (i.e. $\{AP, B\neg P\}$), and the toss of the two coins shows head for coin 1 and tail for the other (i.e. $\{c_1h, c_1h, c_2t, c_2t\}$). The execution trace indicates with a pair of consecutive arrows a transition that is not enabled. The current number of tokens of each place is shown as a label to a loop-arrow along the execution timeline (i.e. the vertical dashed line) of the place ambients. The execution trace of an enabled transition starts with an arrow labelled with the name of the transition and terminates with an arrow labelled *end*. Therefore, in Figure 7 the simulation shows that the only enabled transitions, and thus executed, for this case are the transitions A1a and B0c. This experiment can be reproduced using the Python implementation of Algorithm 1 given in Appendix A to generate the CCA process for the dining cryptographers protocol and then execute the CCA process in the simulation too ccaPL.

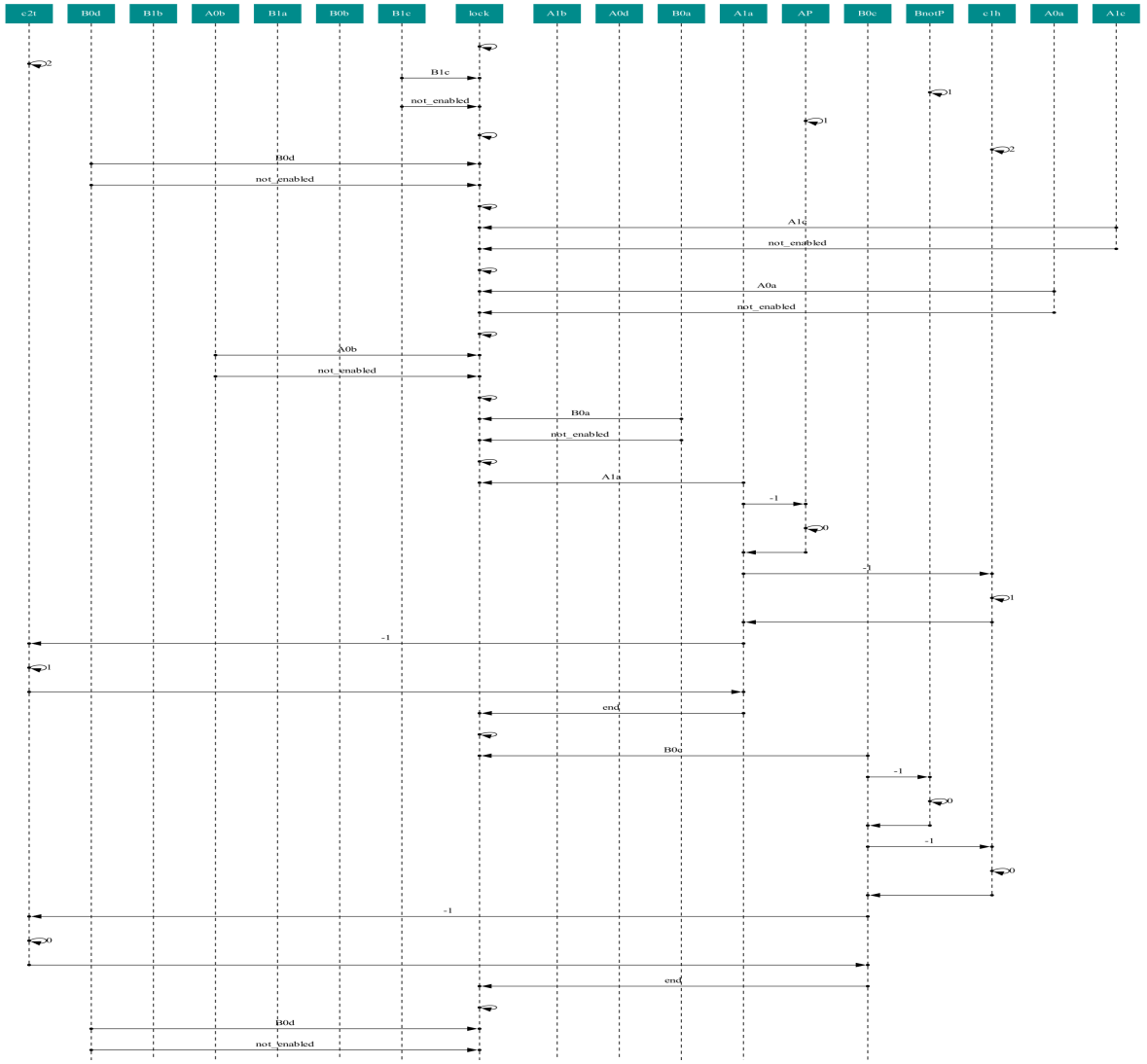


Figure 7. Execution trace for the case where Alice paid and Bob did not pay (i.e. $\{AP, B\neg P\}$) and coin 1 shows head and coin 2 shows tail (i.e. $\{c_1h, c_1h, c_2t, c_2t\}$)

 doi:10.20944/preprints202404.1400.v1

Table 12. Comparison of the behaviours of the Petri net in Figure 6 to that of its corresponding CCA process

Inputs		Outputs		
Coin markings	Cryptographer markings	Petri net Enabled transitions	CCA process Executed ambients	#
$\{c_1h, c_1h, c_2h, c_2h\}$	$\{AP, B\neg P\}$	$A0_a$, $B1_c$	A0a, B1c	1
	$\{A\neg P, BP\}$	$A1_c$, $B0_a$	A1c, B0a	2
	$\{A\neg P, B\neg P\}$	$A1_c$, $B1_c$	A1c, B1c	3
$\{c_1h, c_1h, c_2t, c_2t\}$	$\{AP, B\neg P\}$	$A1_a$, $B0_c$	A1a, B0c	4
	$\{A\neg P, BP\}$	$A0_c$, $B1_a$	A0c, B1a	5
	$\{A\neg P, B\neg P\}$	$A0_c$, $B0_c$	A0c, B0c	6
$\{c_1t, c_1t, c_2h, c_2h\}$	$\{AP, B\neg P\}$	$A1_b$, $B0_b$	A1b, B0b	7
	$\{A\neg P, BP\}$	$A0_d$, $B1_b$	A0d, B1b	8
	$\{A\neg P, B\neg P\}$	$A0_d$, $B0_d$	A0d, B0d	9
$\{c_1t, c_1t, c_2t, c_2t\}$	$\{AP, B\neg P\}$	$A0_b$, $B1_d$	A0b, B1d	10
	$\{A\neg P, BP\}$	$A1_d$, $B0_b$	A1d, B0b	11
	$\{A\neg P, B\neg P\}$	$A1_d$, $B1_d$	A1d, B1d	12

4. Discussion

There has been a substantial amount of research that adapts and relates features of process algebras to Petri nets. Petri Box calculus [22,23], for instance, is a process algebra based on CCS that presents a compositional semantics for high level constructs of concurrent programming languages with regard to Petri nets. [24] proposed a translation from Condition/Event (C/E) nets to Circal process algebra based on a binary composition and hiding operators. Moreover, in [25–27], frameworks that endow Petri nets with labelled transition systems are presented, applying techniques come from process algebras. In particular, in [25], the theory of bigraphs has been applied to C/E nets, by converting C/E nets to bigraphs and examining their behavioural theory. Moreover, there has been a significant work on translating process algebras to Petri nets [28]; with application to the verification of mobile systems [29]. For instance, [30,31] proposed a translation of CCS into Petri nets, while [32] presented a distributed semantics for π-calculus, based on Petri nets.

Powerful, usable and flexible contemporary systems are characterised by features such as *dynamic reconfigurability*, where nodes in networks dynamically can appear or vanish; and *logical mobility*, where connections in ad-hoc networks can be formed dynamically. These systems are also called reference passing systems (RPS) [28]. As the number of such systems constantly is increased, the correct functionality of such systems is of paramount importance to eliminate potential costly errors in the design phase.

There is a number of formalisms that are suitable for modelling and verifying specifications that are characterised by concurrency and the ability to form dynamic logical connections between individual modules [33,34]. The major factors and trade-offs in selecting an appropriate formalism are its expressiveness and the tractability of the associated verification techniques. Formalisms that are expressive are Turing powerful and so not decidable in general. However, it is possible apply some restrictions (e.g., finiteness of the control) that would ensure decidability, while preserving a reasonable modelling power.

In [35], the authors present a number of equivalence notions on Petri nets that can be used in the construction of algebraic models. In [36], the authors proposed a new set of inference rules for a subclass of Calculus of Communicating Systems (CSS). This allows for a direct translation to a subclass of Petri nets, more particularly, to condition/event nets. Early work on translating Petri nets into CCA is presented in [37], but the proposed algorithm is limited to a subset of Petri nets without multiple arcs between a place and a transition. Our algorithm can handle Petri nets with multiple arcs between places and transitions, e.g. the Petri net in Figure 1. In [38], a notion of *net calculus* is introduced, which is defined through a place/transition Petri net. The authors proposed a calculus of nets, called SCONE. Based on this, relationships between SCONE and a subset of CCS are studied. In

 doi:10.20944/preprints202404.1400.v1

[39], different approaches for the modelling of parallel processes are examined. The work considers process algebras, like CCS and communicating sequential processes (CSP) as main representatives. It is shown that the construction of transition nets is possible for all CCS programs in which recursive calls start sequentially. In [33], Finite Control Processes (FCP), a subclass of π-calculus, have been proposed, where the system is defined as a parallel composition of sequential entities. In [28], the authors introduced a translation of FCP to safe Petri nets to formally verify mobile systems.

In formal methods it is customary to compare the expressive power of various languages and provide way of mapping one language to another. This paper has followed the same approach and proposes an algorithm that maps Petri nets onto CCA. The graphical language of Petri nets can then be combined uniformly with the text-based specification language of CCA to facilitate the specification and documentation of the design of complex distributed systems. The CCA tools such as ccaPL can also be used as additional tools to analyse the behaviours of Petri nets. The main limitation of this work is that the equivalence between a Petri net and its corresponding CCA process is not formally established, but through experiments. Thus in future work, we will investigate a formal proof of a behavioural equivalence between a Petri net and its corresponding CCA process. Reversely, we will also investigate whether CCA can be mapped onto Petri nets.

5. Conclusion

This paper demonstrates that any system that can be specified in Petri nets can also be specified in CCA. Therefore, CCA is at least as expressive as Petri nets. Indeed, an algorithm is proposed that transforms a Petri net into a process in CCA. It is established, through experiments, that a Petri net and its corresponding CCA process exhibit the same behaviours. The CCA process can then be analysed and verified using the CCA verification tools like the CCA simulator ccaPL. The algorithm is implemented in Python and therefore the translation of a Petri net into a CCA process can be done automatically at a click of a button. It follows from this work that the Petri net graphical notations can be used together with the CCA language in the specification of complex pervasive and IoT systems. A real-world case study of the dining cryptographers problem was used to illustrate the proposed approach.

Author Contributions: Conceptualization, F. Siewe and V. Germanos and W. Zeng; methodology, F. Siewe; software, F. Siewe; validation, F. Siewe, V. Germanos and W. Zeng; formal analysis, F. Siewe.; investigation, F. Siewe and V. Germanos. and W. Zeng; writing—review and editing, F. Siewe and V. Germanos and W. Zeng. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflicts of interest.

Appendix A. An implementation of Algorithm 1 in Python

The Python code below generates a CCA process in the file petrinet2cca.cca, which can then be executed in the CCA simulator ccaPL to analyse the behaviours of the corresponding Petri net.

```
# Mapping Petri net to CCA (March 2024, Python version 3.10.11)
# output file
PATH = "petrinet2cca.cca"
# Space: minim indentation
Space = "   "
# P is a finite set of places names
P = ["red", "black"]
# T is a finite set of transitions names
T = ["rb", "rr", "bb"]
# I is the input function; I: PxT-->N.
```

 doi:10.20944/preprints202404.1400.v1


```
# I = {place:{trans:nb_arcs, ...}, ...}
# nb_arcs: number of arcs from "place" to "trans"
I = {"red":{"rb":1, "rr":2}, "black":{"rb":1, "bb":2}}
# O is the output function; I: TxP-->N.
# O = {trans:{place:nb_arcs, ...}, ...}
# nb_arcs: number of arcs from "trans" to "place"
O = {"rb":{"red":1}, "rr":{"black":1}, "bb":{"black":1}}
# m0 is the initial marking. m0 = {place:nb_tokens, ...}
m0 = {"red":3, "black":2}

# Save output in a file
def display(cca_str):
    F = open(PATH, "w")
    i = 0; print(cca_str); F.write(cca_str)
    F.close()
# Return the number of arcs from a place to a transition.
def inputs(place,trans):
    if place in I.keys() and trans in I[place].keys():
        return I[place][trans]
    else:
        return 0
# Return the number of arcs from a transition to a place.
def outputs(trans, place):
    if trans in O.keys() and place in O[trans].keys():
        return O[trans][place]
    else:
        return 0
# The mapping function
def mapping():
    # cca_str: contain the CCA program generated
    cca_str =  "BEGIN_DECLS\n"+Space+\
    "def lockOn() = { somewhere (lock[on[0] | true] | true) }\n"+\
    Space+"def state(p,x) = { somewhere (p[x[0] | true] | true) }\n"+\
    Space+"//display code\n"+Space+"//display congruence\n"+Space+\
    "mode random\n"+Space+"length=200\nEND_DECLS\n"+ "lock[\n"+Space+\
    "! recv().::recv(t).{ on[0] | t::recv(x).del on.send().0 }\n"+Space+\
    "| send().0\n]\n"
    # Create all the place ambients
    for x in P:
        cca_str += "|\n"; cca_str += x + "[\n"+Space+"send("+str(m0[x])+\
        ").0\n"+Space+"| !recv(n).let zz=_+(1000+n) in ::recv(v).del zz."+\
        "let w= n+v, y=_+(1000+n+v) in send(w).::send().y[0]\n"
        cca_str += Space+"| _"+str(1000+m0[x])+"[0]\n"; cca_str += "]\n"
     # Create all the transitions ambients
    for t in T:
        var = ""; cond = ""; trouve = False;
        for x in I.keys():
            if t in I[x].keys():
                if not trouve:
                    cond += "_M_"+x+">=_"+str(1000+I[x][t])
```

 doi:10.20944/preprints202404.1400.v1


```
                    trouve = True
                else:
                    cond += " and _M_"+x+">=_"+str(1000+I[x][t])
                var += "find _M_"+x+": state("+x+",_M_"+x+") for "
        cca_str += "|\n"
        cca_str += t +"[\n"+Space+"!< not lockOn() >lock::send("+t+")."+\
                   var+"if \n"+Space+Space+"< "+cond+" > "
        for x in P:
            if (x in I.keys() and t in I[x].keys())\
            or (t in O.keys() and x in O[t].keys()):
                cca_str += x+"::send("+str(-inputs(x,t)+outputs(t,x))+\
                           ")."+x+"::recv()."
        cca_str += "lock::send(end).0\n"
        cca_str += Space+Space+"else lock::send(not_enabled).0\n"+\
        Space+Space+"fi.0\n]\n"
    return cca_str;
# Display and save output
display(mapping())
```

References

1. Petri, C.A. Kommunikation mit Automaten (Communication with Automata). Phd thesis, University of Bonn, 1962.
2. Murata, T. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE* **1989**, *77*, 541–580.
3. Hoare, C. *Communicating Sequential Processes*; Prentice Hall, 1985.
4. Milner, R. *Communication and Mobile Systems: The π-Calculus*; Cambridge University Press, 1999.
5. Fournet, C.; Gonthier, G. The reflexive CHAM and the join-calculus, New York, NY, USA, 1996; POPL '96, p. 372–385. https://doi.org/10.1145/237721.237805.
6. Siewe, F.; Zedan, H.; Cau, A. The Calculus of Context-aware Ambients. *Journal of Computer and System Sciences* **2011**, *77*, 597–620.
7. Siewe, F. ccaPL: A CCA Programming Environment. https://fsiewe.afrilocode.net/CCA/index.html (accessed: 23 November 2023).
8. Cardelli, L.; Gordon, A.D. Mobile Ambients. *Theoretical Computer Science* **2000**, *240*, 177–213.
9. Alfakeeh, A.S.; Al-Bayatti, A.H.; Siewe, F.; Baker, T. Agent-based negotiation approach for feature interactions in smart home systems using calculus of the context-aware ambient. *Transactions on Emerging Telecommunications Technologies* **2022**, *33*, e3808.
10. Siewe, F.; Yang, H. Privacy protection by typing in ubiquitous computing systems. *Journal of Systems and Software* **2016**, *120*, 133–153. https://doi.org/https://doi.org/10.1016/j.jss.2016.07.037.
11. Atbaiga, N.; Siewe, F. Formal Specification of a Context-aware Whiteboard System in CCA. In Proceedings of the In Proceedings of the Libyan International Conference on Electrical Engineering and Technology (LICEET 2018), Tripoli, Libya, 2018.
12. Mennicke, S. A Petri Net Semantics for the Join-Calculus. Technical report, Technical Report, TU Braunschweig. Available at https://www. tu-braunschweig . . . , 2012.
13. Yao, W.; He, X. Mapping Petri nets to concurrent programs in CC++. *Information and Software Technology* **1997**, *39*, 485–495. https://doi.org/https://doi.org/10.1016/S0950-5849(97)00006-2.
14. Korečko, t.; Sobota, B. Petri Nets to B-Language Transformation in Software Development. *Acta Polytechnica Hungarica* **2014**, *11*, 187–206.
15. Boukelkoul, S.; Redjimi, M. Mapping between Petri nets and DEVS models. In Proceedings of the 2013 3rd International Conference on Information Technology and e-Services (ICITeS), 2013, pp. 1–6. https://doi.org/10.1109/ICITeS.2013.6624067.

 doi:10.20944/preprints202404.1400.v1

16. Marsan, M.A.; Balbo, G.; Conte, G.; Donatelli, S.; Franceschinis, G. Modelling with generalized stochastic Petri nets. *ACM SIGMETRICS performance evaluation review* **1998**, *26*, 2.
17. van der Aalst, W.M.P. Putting high-level Petri nets to work in industry. *Computers in Industry* **1994**, *25*, 45–54. https://doi.org/https://doi.org/10.1016/0166-3615(94)90031-0.
18. AT&T Labs-Research. *Graphviz Distribution*. http://www.research.att.com/sw/tools/graphviz/download.html (accessed: 23 November 2023).
19. Chaum, D. The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability. *J. Cryptol.* **1988**, *1*, 65–75.
20. Mazaré, L. Using unification for opacity properties. In Proceedings of the In Proceedings of the Workshop on Issues in the Theory of Security (WITS'04), 2004, pp. 165–176.
21. Bryans, J.W.; Koutny, M.; Ryan, P.Y.A. Modelling Opacity Using Petri Nets. *Electr. Notes Theor. Comput. Sci.* **2005**, *121*, 101–115.
22. Best, E.; Devillers, R.; Hall, J.G. The box calculus: A new causal algebra with multi-label communication. In Proceedings of the Advances in Petri Nets 1992; Rozenberg, G., Ed., Berlin, Heidelberg, 1992; pp. 21–69.
23. Koutny, M.; Esparza, J.; Best, E. Operational Semantics for the Petri Box Calculus. In Proceedings of the CONCUR '94: Concurrency Theory; Jonsson, B.; Parrow, J., Eds., Berlin, Heidelberg, 1994; pp. 210–225.
24. Cerone, A. Implementing Condition/Event Nets in the Circal Process Algebra. In Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering, FASE '02; Jonsson, B.; Parrow, J., Eds., Springer-Verlag, 2002; pp. 49–63.
25. Milner, R. Bigraphs for Petri nets. In Proceedings of the Advanced Course on Petri Nets. Springer, 2003, pp. 686–701.
26. Sassone, V.; Sobociński, P. A congruence for Petri nets. *Electronic Notes in Theoretical Computer Science* **2005**, *127*, 107–120.
27. Leifer, J.J.; Milner, R. Transition systems, link graphs and Petri nets. *Mathematical Structures in Computer Science* **2006**, *16*, 989–1047.
28. Khomenko, V.; Meyer, R.; Hüchting, R. A polynomial translation of pi-calculus FCPs to safe Petri nets. *Logical Methods in Computer Science* **2013**, *9*.
29. Khomenko, V.; Germanos, V. Modelling and Analysis Mobile Systems Using-calculus (EFCP). In *Transactions on Petri Nets and Other Models of Concurrency X*; Springer, 2015; pp. 153–175.
30. Degano, P.; De Nicola, R.; Montanari, U. A distributed operational semantics for CCS based on condition/event systems. *Acta Informatica* **1988**, *26*, 59–91.
31. Goltz, U. CCS and Petri nets. In *LITP Spring School on Theoretical Computer Science*; Springer, 1990; pp. 334–357.
32. Busi, N.; Gorrieri, R. Distributed semantics for the π-calculus based on Petri nets with inhibitor arcs. *The Journal of Logic and Algebraic Programming* **2009**, *78*, 138–162.
33. Dam, M. Model Checking Mobile Processes. *Information and Computation* **1996**, *129*, 35–51.
34. Sangiorgi, D.; Walker, D. *The π-calculus: A Theory of Mobile Processes*; Cambridge University Press, 2001.
35. van Glabbeek, R.; Vaandrager, F. Petri net models for algebraic theories of concurrency. In Proceedings of the PARLE Parallel Architectures and Languages Europe; de Bakker, J.W.; Nijman, A.J.; Treleaven, P.C., Eds. Springer Berlin Heidelberg, 1987, pp. 224–242.
36. Degano, P.; Nicola, R.D.; Montanari, U. A distributed operational semantics for CCS based on condition/event systems. *Acta Informatica* **1988**, *26*, 59–91.
37. Siewe, F.; Germanos, V.; Zeng, W. Analysing Petri Nets in a Calculus of Context-Aware Ambients. In Proceedings of the 2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC). IEEE Computer Society, 2020, pp. 1647–1652.
38. Gorrieri, R.; Montanari, U. On the implementation of concurrent calculi in net calculi: two case studies. *Theoretical Computer Science* **1995**, *141*, 195–252.
39. Taubner, D.A. *Finite Representations of CCS and TCSP Programs by Automata and Petri Nets*; Springer Berlin, Heidelberg, 1989.

 doi:10.20944/preprints202404.1400.v1

Short Biography of Authors

Dr. François Siewe is a Reader in Software Engineering in the School of Computer Science and Informatics at De Montfort University (DMU) in Leicester in the UK. He received a Ph.D. degree in Computer Science from De Montfort University in 2005. He obtained a B.Sc. degree in Mathematics and Computer Science in 1990, a M.Sc. degree in 1991, a Diplome d'Etude Approfondie (DEA) degree in Computer Science in 1992, and a Doctorat de Troisième Cycle degree in Computer Science in 1997 from the University of Yaoundé I in Yaoundé, Cameroon. Prior to joining DMU, he was a Fellow at the United Nations University International Institute for Software Technology (UNU-IIST) in Macua in China, and a Lecturer at the University of Dschang in Cameroon. His research interests include software engineering, formal methods, cyber security, context-aware and pervasive computing, and Internet of Things (IoT).

Dr. Vasileios Germanos is a Senior Lecturer in Computer Science in the Scholl of Computer Science and Informatics at De Montfort University, in Leicester in the UK. He obtained his MSc and PhD, both in Computing Science, from Newcastle University, UK. His research include formal methods, concurrent systems and cyber security.

Dr. Wen Zeng is an Associate Professor in the School of Computer and Information Engineering at Shanghai Polytechnic University. She is an Honorary Senior Research Fellow at De Montfort University, U.K., and a Guest Member of Staff at Newcastle University, U.K. She received her PhD and MSc both from Newcastle University, U.K. After that, she became a Post-doc Research Associate at Newcastle University. In 2017, she joined the School of Computer Science and Informatics at De Montfort University to work as a Tenured Senior Lecturer in Cybersecurity. Her current research interests center on the system design and optimization, including distributed systems, Internet of Things with Cloud computing systems, information flow, Cybersecurity, data privacy and risk management, and big data.