software

MDPI

Article

Mapping Petri Nets onto a Calculus of Context-Aware Ambients

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

[1] School of Computer Science and Informatics, De Montfort University, Leicester LE1 9BH, UK; vasileios.germanos@dmu.ac.uk
[2] School of Computer and Information Engineering, Shanghai Polytechnic University, Shanghai 201209, China; zengwen@sspu.edu.cn
* Correspondence: fsiewe@dmu.ac.uk

Abstract: Petri nets are a graphical notation for describing a class of discrete event dynamic systems whose behaviours are characterised 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) has 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 prove that a Petri net and its corresponding CCA process are behavioural equivalent. 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 the behaviours of Petri nets.

Keywords: calculus of context-aware ambients; CCA; Petri nets; dining cryptographers problem; behavioural equivalence; simulation; ccaPL; formal methods

check for updates

Citation: Siewe, F.; Germanos, V.; Zeng, W. Mapping Petri Nets onto a Calculus of Context-Aware Ambients. *Software* **2024**, *3*, 284–309. https://doi.org/10.3390/software3030015

Academic Editor: Paolino Di Felice

Received: 4 June 2024
Revised: 25 June 2024
Accepted: 1 July 2024
Published: 18 July 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

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 [5] 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. Join-calculus [6] 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 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) [7,8], inspired by the Calculus of Mobile Ambient (MA) [9], 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 emerges as a suitable formalism to reason with the behaviours of pervasive systems and the Internet of Things [10].

It is customary in formal methods to compare the expressive powers of formalisms, i.e., whether one formalism can be mapped onto another. In [11], Mennicke presents operational Petri net semantics for join-calculus. An approach to mapping Petri nets to concurrent programs in CC++ is proposed in [12], 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, Ref. [13] proposes a mapping from Petri nets to the language of the B-Method in an effort to incorporate the Petri nets' graphical notation into a software development method based on the B-language. A mapping of Petri nets to DEVS (Discrete Event System Specification) is presented in [14], so DEVS-based platforms can be used to analyse systems specified in Petri nets.

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

- We propose an algorithm that transforms any Petri net into a CCA process (Section 2.3).
- We prove that a Petri net and its corresponding CCA process are behavioural equivalents (Section 2.3.3). This shows 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 (Section 2.3.4). 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 (Section 2.3.4 and Appendix A).
- The proposed approach is illustrated using a case study of the dining cryptographers problem (Section 3).

The remainder of the paper is structured as follows. Section 2 presents the proposed algorithm for mapping Petri nets onto CCA and discusses the efficiency and scalability of the algorithm. The notion of behavioural equivalence is also introduced in this section. The case study is presented in Section 3. In Section 4, we discuss the results. Section 5 concludes the paper and proposes future work.

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 [2,15]. 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, and never between places or between transitions. A place may contain a non-negative integer number of marks called *tokens*. An example of a Petri net is depicted in Figure 1. Any distribution of tokens over the places will represent a configuration of the net called a *marking*.

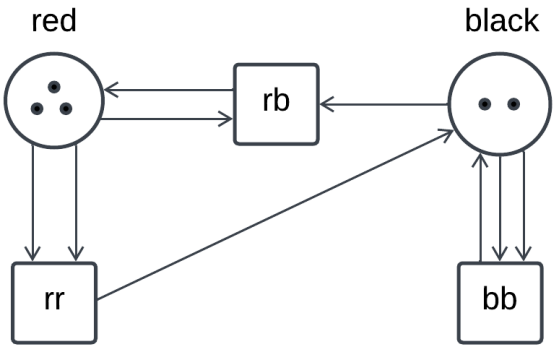


Figure 1. An example of a Petri net [16].

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 $P \cap T = \emptyset$.
- $I : P \times T \to \mathbb{N}$ 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 shown in Figure 1 models the following ball game [16]. The tokens in the places red and black represent red and black balls in an urn, respectively. As long as there are at least two balls in the urn, a person takes two balls from the urn. If the balls are of the same colour, then a black ball is returned; otherwise, a red ball is returned. Transition rb fires if the person takes two balls of different colours, transition rr fires if two red bails are taken and transition bb fires if two black balls are taken. This Petri net is formally defined as follows:*

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

Let us denote by $Input(t)$ (with respect to $Output(t)$) the set of all input places (with respect to all output places) of a transition t, i.e., $Input(t) = \{p \in P \mid I(p, t) > 0\}$ and $Output(t) = \{p \in P \mid O(t, P) > 0\}$. Transitions are the active components of a Petri net. A transition t is enabled if each of its input places p contains at least $I(p, t)$ tokens, as formulated in (1), where m is the marking of the Petri net.

$$enabled(t) = \forall p \in Input(t) : m(p) \geq I(p, t). \tag{1}$$

A transition may execute (or fire) 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 (2), where $m(p)$ is the marking of a place p before the transition is executed and $m'(p)$ is the marking of a place p after the transition is executed.

$$m'(p) = m(p) - I(p, t) + O(t, p). \tag{2}$$

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

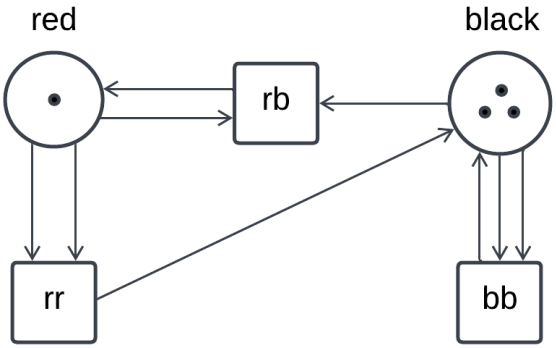


Figure 2. Petri net of Figure 1 after the execution of the transition rr.

2.2. Overview of CCA

CCA [7,8] is a process calculus for specifying and reasoning regarding the behaviour of context-aware, mobile and concurrent systems. CCA is based on a single notion of *ambient*, which conceptualises an entity in which a computation can happen. An ambient is represented in the form $n[P]$, where n is the name of the ambient and P is a process describing the behaviour of the ambient. We can also say that the process P is executed by the ambient n. An ambient can contain other ambients, called child ambients, forming a hierarchy of ambients. For example, in $n[P \mid m[Q]]$, n is the parent ambient of m and m is a child ambient of n. Two ambients that have the same parent are siblings. There is a special ambient called *root*, which represents the root of the ambient hierarchy. Any ambient in CCA is a descendent of the root ambient. Ambients can be composed in parallel, using the parallel operator "|", to form a concurrent system. During their execution, ambients can communicate through a hand-shake message passing mechanism; i.e., an ambient (the sender) can send a message to another ambient (the receiver) and the sender and the receiver must synchronise for the communication to happen, as explained in Section 2.2.2. An ambient can be mobile; the mobility capabilities "`in`" and "`out`", described in Section 2.2.2, allow an ambient to move inside a sibling ambient and outside its parent ambient, respectively. An ambient can also be aware of its context so that its behaviour can change in response to changes in the context. This is undertaken using context expressions and context-guarded processes (see Section 2.2.4). A context expression is a predicate over the state of the ambient hierarchy. In distributed systems, each component of a system can be modelled as an ambient, e.g., sensors and computers. An example is given in Section 2.2.5.

CCA is formally described by the grammar in Table 1, 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.

Table 1. Syntax of CCA.

P, Q	::=		Processes	κ	::=		Context Expressions
		$\mathtt{0}$	inactivity			$\mathtt{0}$	empty context
		$P \mid Q$	parallel composition			`true`	true
		$\{P\}$	block			`false`	false
		$(\mathtt{new}\ n)\ P$	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
		$\alpha\ x(z_1, \ldots, z_\ell)$	process abstraction call			ϵ	locally

2.2.1. Processes

The syntax of a process P is given in Table 1. The process `0`, also known as the *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 $(\mathtt{new}\ n)\ P$ 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, and 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, and then proceeds non-deterministically like one of the $< \kappa_j > M_j.P_j$ processes 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 P process 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 with 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 is $\ell \geq 0$. A process abstraction is a mechanism to give a name, say x, to a process P and to later use that name anywhere to refer to the process P, just the same way as functions and procedures are used in programming languages.

2.2.2. Capabilities

A capability is an elementary action that an ambient can perform. The syntax of a capability M is given in Table 1. 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`$(y_1, \ldots, y_\ell)$ can receive a list of names from a location α for 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., $n[0]$). A capability 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, for which C is a context. For example, suppose a system is modelled by the process $P \mid n[Q \mid m[R \mid S]]$. Therefore, 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	::=	$0 \mid \odot \mid n[C] \mid C \mid P \mid (\texttt{new}\ n)\ C$

2.2.4. Context Expressions

The syntax of a process expression κ is given in Table 1. The formal semantics of context expressions (CEs) with respect to the context model of Table 2 are 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 contexts.

Table 3. Satisfaction relation for context expressions.

	$\models$	`true`		
C	$\models$	$n = m$	iff	$n = m$
C	$\models$	`0`	iff	$C = \mathtt{0}$
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 contexts, while the CE `false` holds for no context. A CE $n = m$ holds if the names n and m are identical. The CE `0` holds for the empty context `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 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.

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

Context expressions are used in CCA to specify context-aware processes. We recall in Table 5 the formal semantics of context-aware processes, where σ is a substitution of names, $\equiv$ is the congruence relation of processes and $\longrightarrow$ is the reduction relation of processes. The notation $val(e)$ represents the value of an expression e. The complete formal semantics of CCA can be found in [7,8].

2.2.5. An Example

This example illustrates how to model a distributed system in CCA. Consider a simple access control system that consists of a RFID tag, RFID reader, a garage door and a server. The system controls access to the garage through the door. When the reader detects an RFID tag, it reads the ID (aka identification) number of the tag and sends an access request to the server. The server checks if the ID number is valid (i.e., permitted to access), in which case the reader shows a green light and the garage door opens. If the ID number is invalid, the reader shows a red light (to mean that the access request is denied) and the door stays closed. Let us assume that a valid ID number is a value between 100 and 200, exclusive. Each component (RFID tag, RFID reader, garage door and the server) of this distributed system can be modelled as an ambient in CCA.

Table 5. Semantics of 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 $C \models \kappa\{x_1 \leftarrow n_1, \ldots, x_\ell \leftarrow n_\ell\}$
(R6)	$C(\texttt{let } x_1 = e_1, \ldots, x_\ell = e_\ell \texttt{ in } P) \equiv C(P\{x_1 \leftarrow val(e_1), \ldots, x_\ell \leftarrow val(e_\ell)\})$
(R7)	$\texttt{skip}.P \longrightarrow P$
(R8)	$\texttt{del } n.P \mid n[0] \longrightarrow P$
(R9)	$!P \longrightarrow P \mid !P$
(R10)	$\texttt{recv}(y_1, \ldots, y_\ell).P \mid \texttt{send}(z_1, \ldots, z_\ell).Q \longrightarrow P\{y_1 \leftarrow z_1, \ldots, y_\ell \leftarrow z_\ell\} \mid Q$, for $\ell \geq 0$
(R11)	$n[:: \texttt{recv}(y_1, \ldots, y_\ell).P \mid R] \mid m[:: \texttt{send}(z_1, \ldots, z_\ell).Q \mid S] \longrightarrow$ $n[P\{y_1 \leftarrow z_1, \ldots, y_\ell \leftarrow z_\ell\} \mid R] \mid m[Q \mid S]$, for $\ell \geq 0$
(R12)	$n[:: \texttt{recv}(y_1, \ldots, y_\ell).P \mid R] \mid m[n :: \texttt{send}(z_1, \ldots, z_\ell).Q \mid S] \longrightarrow$ $n[P\{y_1 \leftarrow z_1, \ldots, y_\ell \leftarrow z_\ell\} \mid R] \mid m[Q \mid S]$, for $\ell \geq 0$
(R13)	$n[m :: \texttt{recv}(y_1, \ldots, y_\ell).P \mid R] \mid m[:: \texttt{send}(z_1, \ldots, z_\ell).Q \mid S] \longrightarrow$ $n[P\{y_1 \leftarrow z_1, \ldots, y_\ell \leftarrow z_\ell\} \mid R] \mid m[Q \mid S]$, for $\ell \geq 0$
(R14)	$n[m :: \texttt{recv}(y_1, \ldots, y_\ell).P \mid R] \mid m[n :: \texttt{send}(z_1, \ldots, z_\ell).Q \mid S] \longrightarrow$ $n[P\{y_1 \leftarrow z_1, \ldots, y_\ell \leftarrow z_\ell\} \mid R] \mid m[Q \mid S]$, for $\ell \geq 0$

Indeed, an RFID tag is a device that stores an ID number. The ID number is sent to an RFID reader when the tag is in the range of the reader. In CCA, an RFID tag and an RFID reader can be represented as ambients. The boundary of the reader ambient delimits the range of the reader, i.e., an RFID tag is in the range of a reader if that tag is a child ambient of the reader. An RFID tag that contains the ID number 166 can be modelled in CCA by the ambient in Listing 1. It is assumed that initially the RFID tag is not in the range of the RFID reader, and then it moves into the range of the reader using the capability "`in`" (described in Section 2.2).

Listing 1. The ambient representing the RFID tag.

```
RFID_tag[
     !@send(166).0
   | in RFID_reader.0
]
```

An RFID reader can be specified, as in Listing 2. It reads the value of an RFID tag and requests the access control service (`ac_service`) from the server. Upon receipt of a response (`status`) from the server, the reader shows a green light (`status=valid`) to indicate that

the access request is authorised, and a red light (`status=invalid`) if the access request is denied.

Listing 2. The ambient representing the RFID reader.

```
RFID_reader[
     !recv().#recv(id).server::send(RFID_reader, ac_service, id).
         server::recv(status).send().0
   | send().0
]
```

The server can be modelled as the ambient in Listing 3. It receives a service request from a client and executes the requested service. A service is modelled as a process abstraction. In this example, there is only one service: the access control service `ac_service`.

Listing 3. The ambient representing the server.

```
server[
     proc ac_service(client, door, id) {
        if
           < (id>100) and (id<200) > client::send(valid).
                 door::send(open).send().0
        else
          client::send(invalid).door::send(close).send().0
        fi
     }
   |
     !recv().::recv(client, service, id).service(client, garage_door, id).0
   | send().0
]
```

The last component of the system to specify is the garage door. The door can be controlled by the server and is modelled by the ambient in Listing 4. It closes if it receives the command "`close`" from the server and opens if it receives the command "`open`". The current state of the door (closed/opened) is checked using the context expression $state(x)$ defined in Table 4.

Listing 4. The ambient representing the garage door.

```
garage_door[
     !recv().server::recv(x).
        if
           < x = close and state(opened) > del opened.send().closed[0]
           < x = open and state(closed) > del closed.send().opened[0]
        else
           send().0
        fi
   | send().0
   | closed[0] // initially the door is closed.
]
```

The overall access control system is modelled in Listing 5 as the parallel composition of its four components described in Listings 1–4.

Listing 5. The CCA process representing the access control system of Section 2.2.5.

```
RFID_tag[
     !@send(166).0
   | in RFID_reader.0
]
|
RFID_reader[
     !recv().#recv(id).server::send(RFID_reader, ac_service, id).
         server::recv(status).send().0
   | send().0
]
|
server[
     proc ac_service(client, door, id) {
        if
           < (id>100) and (id<200) > client::send(valid).
                door::send(open).send().0
        else
          client::send(invalid).door::send(close).send().0
        fi
     }
   |
     !recv().::recv(client, service, id).service(client, garage_door, id).0
   | send().0
]
|
garage_door[
     !recv().server::recv(x).
        if
           < x = close and state(opened) > del opened.send().closed[0]
           < x = open and state(closed) > del closed.send().opened[0]
        else
           send().0
        fi
   | send().0
   | closed[0] // initially the door is closed.
]
```

Since CCA specifications are executable, the simple access control system can be analysed using the CCA simulator ccaPL, and the simulation outputs are illustrated in Figures 4 and 5 in Section 3.1. The next section presents an algorithm for translating a Petri net into an equivalent CCA process.

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

Algorithm 1 translates a Petri net (see Section 2.1) into a CCA process. The algorithm takes as input a Petri net (P, T, I, O, m_0) and returns in output a CCA process in the variable *cca_str*. In CCA, Petri net places and transitions are modelled as ambients, as explained in Sections 2.3.1 and 2.3.2. We call a *place ambient* an ambient that models a Petri net place. Likewise, we call a *transition ambient* an ambient that models a Petri net transition. A semaphore is used to guarantee that the execution of a transition ambient is atomic, i.e., at most one transition ambient can be executed at a time. In order to execute, a transition

ambient must obtain the semaphore; otherwise, the transition ambient must wait until the semaphore is released by another transition ambient. This semaphore is described in CCA by the ambient `lock` defined in Listing 6. The semaphore is in the *obtained* state if the ambient contains a child ambient `on[0]` and is in the *released* state otherwise.

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*

```
Space = "   ";
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";
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";
// Generate the place ambients
for p ∈ P do
    xx = 1000 + m0(p);
    cca_str = cca_str + " | \n" + p + "[\n send(" + m0(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";
// Generate the transition ambients
for t ∈ T do
    var = "";
    cond = "";
    trouve = False;
    foreach p ∈ P such that I(p,t) ≠ 0 do
        val = 1000 + I(p,t);
        if not trouve then
            cond = cond + "_M_" + p + ">=_" + val;
            trouve = True;
        else
            cond = cond + " and _M_" + p + ">=_" + val;
        var = var + "find _M_" + p + ": state(" + p + ",_M_" + p + ") for ";
    cca_str = cca_str + " | \n" + t +"[\n" + Space + "! < not lockOn() > lock::send(" +
      t + ")." + var + "if \n" + Space + Space + "< " + cond + " >"
    foreach p ∈ P such that I(p,t) ≠ 0 or O(t,p) ≠ 0 do
        val = −I(p,t) + O(t,p);
        cca_str = cca_str + p + "::send(" + val + ")." + p + "::recv().';
    cca_str = cca_str + "lock::send(end).0\n" + Space + Space + "else
      lock::send(not_enabled).0\n" + Space + Space + "fi.0\n]\n";
```

Listing 6. The ambient representing the semaphore `lock`.

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

The behaviour of the ambient `lock` can be explained as follows. Initially, the semaphore is in the released state, i.e., no transition ambient is being executed. The ambient waits (us-

ing the capability :: recv(t)) until a transition ambient t is willing to obtain the semaphore, and then creates a child ambient on[0] to indicate that the semaphore is obtained by the transition ambient t. The semaphore remains in the obtained state until it is released by the transition ambient 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., if 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 lines 5–7 of Algorithm 1 are explained in Section 2.3.1 and the lines 9–25 are explained in Section 2.3.2.

2.3.1. Modelling Places

In this section, we show how a Petri net place can be modelled as an ambient in CCA. For each place $p \in P$, an ambient (called *place ambient*) of the same name is created in line 7 of Algorithm 1. The general form of this place ambient is given in Listing 7. Initially, a place ambient contains $m_0(p) = x$ tokens. This is represented in the place ambient in Listing 7 by the process "send(x).0". A child ambient of the place ambient 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 tokens in the place ambient and satisfies the property stated in Theorem 1. Therefore, the names of the child ambients of the place ambients can be compared instead of the number of tokens of these place ambients. The behaviour of a place ambient can be summarised as follows. A place ambient (see Listing 7) reads its current number of tokens, calculates the name of its child ambient and then waits until a transition ambient it is connected to starts executing (see ":: recv(v)"). It then deletes its current child ambient, updates its number of tokens according to (2) and creates a child ambient that reflects its new number of tokens. It also signals to the running transition ambient 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 Listing 8. The loop in lines 5–7 of Algorithm 1 creates such a place ambient for each place $p \in P$ and composes them in parallel.

Listing 7. 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]
]
```

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

Definition 2. *The marking (also known as the number of tokens) of a place ambient is a non-negative integer number x such that the name of its child ambient is "_" $+ str(1000 + x)$. For example, the marking of the place ambient* red *in Listing 8 is 3 and that of the place ambient* black *is 2.*

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

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

Listing 8. Examples of 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]
]
```

The proof of Theorem 1 can be carried out by induction on n, as follows.

Proof. Let $\|z\|$ denote the number of digits 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, $x \geq y \Rightarrow str(10 + x) =$"$1x$" $\geq str(10 + y) =$"$1y$".

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

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 digits $\sigma \geq 1$ 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 two 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 Petri net transition can be modelled as an ambient in CCA. Indeed, Algorithm 1, in the lines 9–25, creates for each transition $\mathtt{t} \in T$ an ambient (called

transition ambient) named `t` of the form described in Listing 9. The transition 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 ambient is encoded to reflect the number of tokens of that place ambient and to satisfy the property stated in Theorem 1. In Listing 9, the variables p1, ..., pn denote the names of the input place ambients of the transition ambient and q1, ..., qk denote the input or output place ambients of the transition ambient. The behaviour of the transition ambient is an iterative process, which waits until the semaphore `lock` is released and then obtains the semaphore `lock` (line 2 in Listing 9). It then finds the names of the child ambients of the input place ambients p1, ..., pn (line 3) and checks if the guard of the if-statement holds (line 4), in which case the number of tokens of each place ambient connected to it is updated according to (2) (line 5). Finally, the process releases the semaphore `lock` and readies itself for further execution (lines 6 and 7). The guard of the if-statement can be formulated as in (3), where t (with respect to p) denotes a transition (with respect to a place) and t_a (with respect to p_a) the transition ambient of t (with respect to the place ambient of p).

$$guard(t_a) = \forall p \in Input(t)\ \exists x : state(p_a, x) \wedge x \geq \text{“_”} + str(1000 + I(p,t)). \tag{3}$$

Definition 3. *We say that a transition ambient t_a is enabled if $guard(t_a)$ holds.*

An example of a transition ambient is given in Listing 10 for the transition *rr* of the Petri net in Figure 1.

Listing 9. 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. ]
```

Listing 10. Example of 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 Listing 11. Table 6 summarises the mapping of the Petri net concepts to CCA concepts.

Listing 11. Example of the CCA process generated 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
]
```

Table 6. Mapping of Petri net concepts to CCA concepts.

Petri Net Concepts	CCA Concepts
a place	an ambient of the form in Listing 7, called place ambient .
a transition	an ambient of the form in Listing 9, called transition ambient.
the number of tokens (also known as marking) of a place	the name of the child ambient of a place ambient encodes the number of tokens of the place ambient, as in Definition 2.
a Petri net	the parallel composition of the semaphore ambient `lock` defined in Listing 6 with all the place ambients and all the transition ambients. An example is given in Listing 11

2.3.3. Behavioural Equivalence

In this section, we prove that the behaviour of a Petri net and that of its corresponding CCA process, generated by Algorithm 1, are equivalent according to Definition 4.

Definition 4 (Behavioural Equivalence)**.** *A Petri net* (P, T, I, O, m_0) *is behaviourally equivalent to its corresponding CCA process generated by Algorithm 1 if:*

- *Each place* $p \in P$ *and its corresponding place ambient* p_a *have the same initial marking, i.e.,*

$$\forall p \in P : m_0(p) = m_0(p_a).$$

- *A transition* $t \in T$ *is enabled if and only if its corresponding transition ambient* t_a *is enabled, i.e.,*

$$\forall t \in T : enabled(t) \Leftrightarrow guard(t_a).$$

- *For each transition* $t \in T$*, the execution of its corresponding transition ambient* t_a *is atomic, i.e., two transition ambients cannot execute at the same time.*
- *For each transition* $t \in T$*, the execution of its corresponding transition ambient* t_a *updates the markings of the place ambients, as in (2).*

In the following lemmas, we assume a Petri net (P, T, I, O, m_0) and its corresponding CCA process generated by Algorithm 1. For each place $p \in P$, we let p_a denote the corresponding place ambient, and for each transition $t \in T$, we let t_a denote the corresponding transition ambient.

Lemma 1. $\forall p \in P : m_0(p) = m_0(p_a)$.

Proof. By construction, Algorithm 1 creates for each place $p \in P$ a place ambient p_a that has a child ambient named $\text{“_”} + str(1000 + m_0(p))$ (see lines 6 and 7 in Algorithm 1). From Definition 2, we can conclude that $m_0(p) = m_0(p_a)$. □

Lemma 2. $\forall t \in T : enabled(t) \Leftrightarrow guard(t_a)$.

Proof. For $t \in T$,

$enabled(t) = \forall p \in Input(t) : m(p) \geq I(p,t)$
$\Leftrightarrow$ $\forall p \in Input(t) : \text{“_”} + str(1000 + m(p)) \geq \text{“_”} + str(1000 + I(p,t))$, from Theorem 1.
$\Leftrightarrow$ $\forall p \in Input(t) : \text{“_”} + str(1000 + m(p_a)) \geq \text{“_”} + str(1000 + I(p,t))$, from Lemma 1.
$\Leftrightarrow$ $\forall p \in Input(t) : state(p_a, \text{“_”} + str(1000 + m(p_a))) \wedge \text{“_”} + str(1000 + m(p_a)) \geq \text{“_”} + str(1000 + I(p,t))$, from Table 4 and Definition 2.
$\Leftrightarrow$ $\forall p \in Input(t)\ \exists x : state(p_a, x) \wedge x \geq \text{“_”} + str(1000 + I(p,t))$, from predicate logic.
$\Leftrightarrow$ $guard(t_a)$.
□

Lemma 3. *The execution of a transition ambient is atomic, i.e., two transition ambients cannot execute at the same time.*

Proof. A semaphore `lock` (defined in Listing 6) is used to guarantee that two transition ambients cannot execute at the same time. The execution of a transition ambient is guarded by a guarded prefix “`< not lockOn() >lock::send(t)`”, as shown in Listing 9. Therefore, the execution starts only if the context expression “`not lockOn()`” holds, i.e., if the semaphore is not in the obtained state, according to the semantic rule (R1) in Table 5 and the definition of the context expression “`lockOn()`” in Table 4. When the semaphore is not in the obtained state, a transition ambient can obtain the semaphore by sending its name to the semaphore using the semantic rules (R9), (R10), (R1) and (R12) in sequence. Once the semaphore has been obtained, it is released only when it receives a message from the transition ambient t that obtained it, according to the continuation process “`t::recv().del on`”

in Listing 6. This continuation is executed by applying the semantic rule (R14) followed by (R8). This happens only when the transition ambient t has reached Line 6 (or Line 7) of Listing 9, and is willing to send a message to the semaphore `lock` using the semantic rule (R14). Therefore, no two transition ambients can execute at the same time. □

Lemma 4. *The execution of an enabled transition ambient updates the markings of the place ambients, as in (2).*

Proof. The general form of a transition ambient is given in Listing 9, and that of a place ambient is shown in Listing 7. Therefore, the execution of an enabled transition ambient t (i.e., (3) holds for this transition ambient) is carried out as follows, using the semantic rules of Table 5.

- In Line 2 of Listing 9, the rules (R9) and (R12) are applied to obtain the semaphore `lock`.
- In Line 3, the rule (R5) is applied to find the name of the child ambient of each input place ambient.
- In Line 4, the condition of the if-statement is evaluated to be true, since the transition ambient is enabled (see (3)). Therefore, Line 5 is executed next, according to the semantic rule (R3).
- In line 5, for each input or output place ambient p of the transition ambient t, the process "`p::send(-I(p,t)+O(t,p)).p::recv()`" is executed to send (using the rule (R12)) to p the value $-I(p,t)+O(t,p)$, and then it must wait for the acknowledgement (using the rule (R13)) that p has successfully updated its marking. Note that the value $-I(p,t)+O(t,p)$ is a constant for each pair $\{p,t\}$.
- In Line 6, the transition ambient releases the semaphore `lock` using the semantic rule (R14).

In the meantime, each input or output place ambient p (specified as in Listing 7) runs in parallel to t and behaves as follows.

- The rules (R9) and (R10) are applied to receive in the variable n the current marking of p.
- Rule (R6) is applied to calculate the name of the current child ambient of p in the variable zz.
- Rule (R12) is used to receive the value $-I(p,t)+O(t,p)$ sent by the running transition ambient t in the variable v.
- Rule (R8) is applied to delete the current child ambient.
- Rule (R6) is applied to calculate the new value ($w = n + v$) of the marking of p in the variable w, as in (2), and to calculate (at the same time) the new name of the child ambient of p ($y =$ "_" $+ str(1000 + w)$) in the variable y.
- Rules (R9) and (R10) are applied to set the new marking of the place ambient to w.
- Rule (R13) is used to tell the running transition ambient that the marking of p has been updated.
- Finally, the new child ambient named y is created.

We conclude that the execution of an enabled transition ambient updates the makings of the place ambients, as in (2). □

Theorem 2. *A Petri net (P, T, I, O, m_0) and its corresponding CCA process generated by Algorithm 1 are behaviourally equivalent.*

Proof. The proof of Theorem 2 is straightforward from Definition 4 and the Lemmas 1–4. □

2.3.4. 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 as the output in comparison to the size of the Petri net received as the 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. In addition, the algorithm creates two ambients for each place (see lin 5): a place ambient with a child ambient, as depicted in Listing 7. 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 Listing 9. Finally, a unique semaphore ambient named lock (see Listing 6) 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 as the output grows linearly with the size of the Petri net as the 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 carried out automatically at the 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 7, 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.

Table 7. 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>
```

It follows that a ccaPL program is composed of a declaration 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 a ccaPL program is given in Listing 11. 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 (First In First Out)) and, in case of conflict, the sequential order as they appear in the program text. 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 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 generate three types of execution output, as shown in Figure 3: a textual execution trace, a communication graph

and a behaviour graph. For illustration, the execution outputs for the simple access control system described in Section 2.2.5 are depicted in Figures 4 and 5.

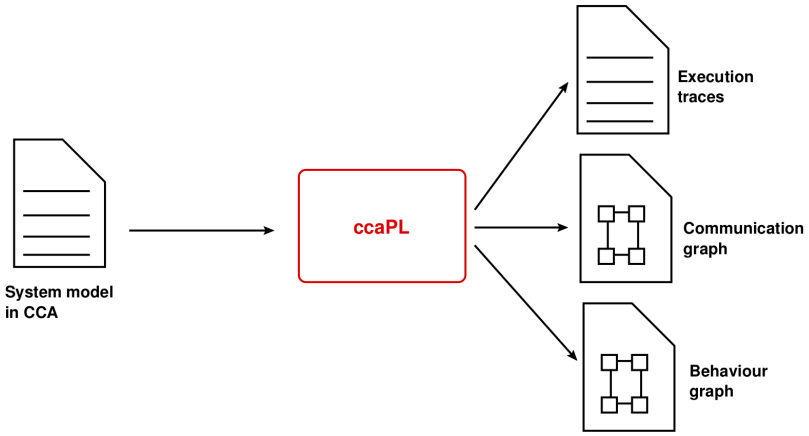
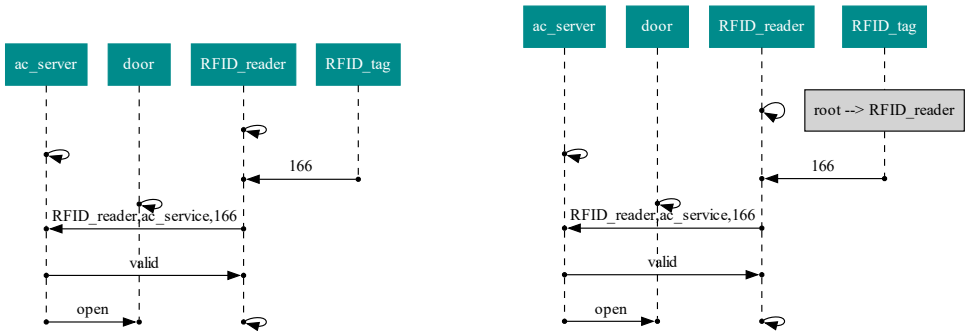


Figure 3. The ccaPL tool.

```
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.



(**a**) Communication graph (**b**) Behaviour graph

Figure 5. Graphical execution traces.

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 `--->` (with respect to `<--->`) for a reduction step (with respect to 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
```

3.1.2. Communication Graph

The execution trace is a diagram showing the timeline of the communications between the ambients. An example of the communication graph is given in Figure 5a. 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
```

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 Location `A` to Location `B`. An example of a behaviour graph is given in Figure 5b. Recall that an ambient can move from one location to another by performing the capability `in` or the capability `out` (see Section 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 [17] be installed on your computer.

3.2. A Case Study: The Dining Cryptographers Problem

The standard dining cryptographers problem [18] 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 [19,20], 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 whether 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. The following is the protocol that they decided to use to solve this problem.

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 on the right. The three possible initial markings for Alice and Bob are given in (4).

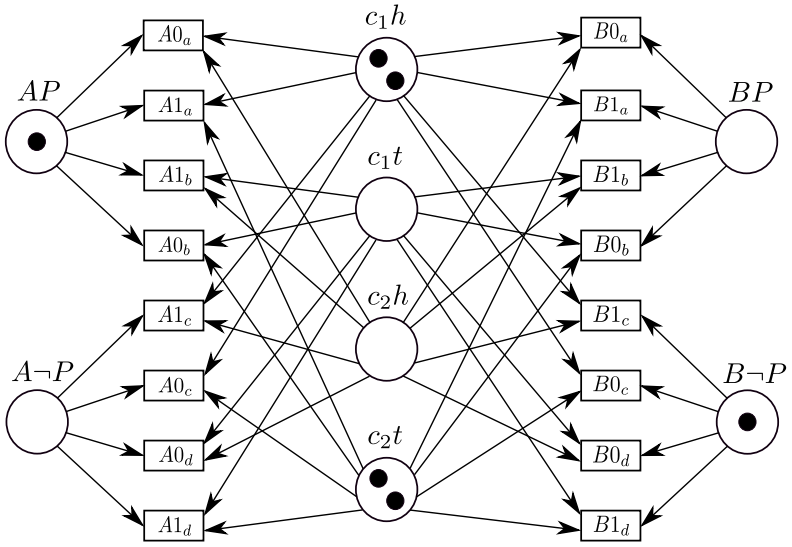


Figure 6. A Petri net representing the dining cryptographers protocol [20].

$$\{AP, B\neg P\}, \{A\neg P, BP\}, \{A\neg P, B\neg P\} \tag{4}$$

The top two places in the centre of the net represent the first coin: heads is represented by placing two tokens in the place c_1h, and tails 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 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 (5).

$$\begin{gathered}\{c_1h, c_1h, c_2h, c_2h\}, \{c_1h, c_1h, c_2t, c_2t\}, \\ \{c_1t, c_1t, c_2h, c_2h\}, \{c_1t, c_1t, c_2t, c_2t\}\end{gathered} \tag{5}$$

The cross-product of the cryptographer markings in (4) and the coin markings in (5) 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 that the coins 'disagree' ($B0$) or Bob saying that the coins 'agree' ($B1$), and it is similar for Alice on the left.

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 in 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 8. 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 in which Alice paid the bill and Bob did not (i.e., $\{AP, B\neg P\}$), and the toss of the two coins shows heads for coin 1 and tails 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 ambient that is not enabled (see (3)). The current number of tokens of each place ambient 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 ambient

starts with an arrow labelled with the name of the transition ambient and terminates with an arrow labelled *end*. Therefore, in Figure 7, the simulation shows that the only enabled transitions (and thus those executed) for this case are the transition ambients 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 the CCA process can be executed in the simulation tool ccaPL.

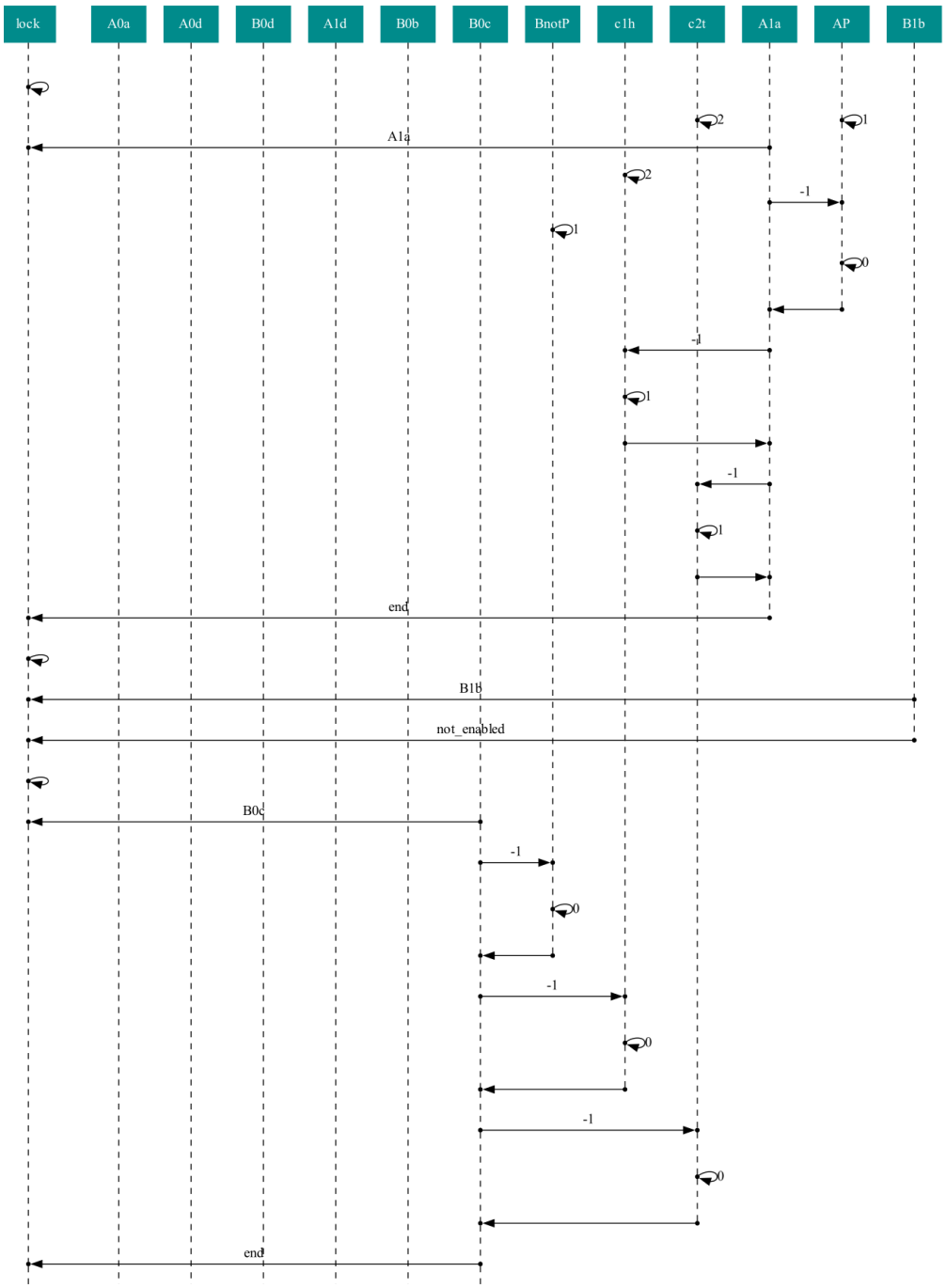


Figure 7. Execution trace for the case where Alice paid and Bob did not pay (i.e., $\{AP, B\neg P\}$) and coin 1 shows heads and coin 2 shows tails (i.e., $\{c_1h, c_1h, c_2t, c_2t\}$).

Table 8. Comparison of the behaviours of the Petri net in Figure 6 to those of its corresponding CCA process. The column # numbers the 12 possible initial markings of the Petri net.

Inputs		**Outputs**		
Coin	**Cryptographer**	**Petri Net**	**CCA Process**	
Markings	**Markings**	**Enabled Transitions**	**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 [21,22], for instance, is a process algebra based on CCS that presents compositional semantics for high-level constructs of concurrent programming languages with regard to Petri nets. Reference [23] proposed a translation from Condition/Event (C/E) nets to Circal process algebra based on a binary composition and hiding operators. Moreover, in [24–26], frameworks that endow Petri nets with labelled transition systems are presented, applying techniques come from process algebra. In particular, in [24], the theory of bigraphs is applied to C/E nets, by converting C/E nets to bigraphs and examining their behavioural theory. Moreover, there has been significant work in translating process algebra to Petri nets [27], with applications in the verification of mobile systems [28]. For instance, refs. [29,30] proposed a translation of CCS into Petri nets, while [31] presented 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 can dynamically 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) [27]. As the number of such systems is constantly increasing, the correct functionality of such systems is of paramount importance to eliminate potential costly errors in the design phase.

There are 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 [32,33]. 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 to apply some restrictions (e.g., finiteness of the control) that would ensure decidability, while preserving a reasonable modelling power.

In [34], the authors present a number of equivalence notions on Petri nets that can be used in the construction of algebraic models. In [29], the authors proposed a new set of inference rules for a subclass of Calculus of Communicating Systems (CSS). This allowed for a direct translation to a subclass of Petri nets, and more particularly to condition/event nets. Early work on translating Petri nets into CCA is presented in [35], 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 [36], 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 [37], different approaches for the modelling of parallel processes are examined. The work considers process algebra, 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 [32], 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 [27], the authors introduced a translation of FCP to safe Petri nets to formally verify mobile systems.

5. Conclusions

In formal methods, it is customary to compare the expressive power of various languages and provide ways of mapping one language to another. This paper has followed the same approach and proposes an algorithm that maps Petri nets onto CCA. It is formally proven that a Petri net and its corresponding CCA process (generated by the proposed algorithm) are behaviourally equivalent. This demonstrates that any system that can be specified in Petri nets can also be specified in CCA. Thus, CCA is at least as expressive as Petri nets. It follows from this work that a Petri net can now be analysed using CCA verification tools like ccaPL. To do so, a Petri net is first translated into a CCA process using the proposed algorithm, and then the CCA process is analysed using ccaPL. The algorithm is implemented in Python so that the translation of a Petri net into a CCA process can be performed automatically at the click of a button. A real-world case study of the dining cryptographers problem was used to illustrate the proposed approach.

In future work, we will investigate how the proposed algorithm can be extended to handle more advanced Petri net variants like coloured Petri nets. We will also look at whether, conversely, CCA can be mapped onto Petri nets.

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

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The original contributions presented in the study are included in the article, further inquiries can be directed to the corresponding author.

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.
# 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])
                    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). Ph.D. Thesis, University of Bonn, Bonn, Germany, 1962.
2. Murata, T. Petri nets: Properties, analysis and applications. *Proc. IEEE* **1989**, *77*, 541–580.
3. Hoare, C. *Communicating Sequential Processes*; Prentice Hall: Hoboken, NJ, USA, 1985.
4. Milner, R. *A Calculus of Communicating Systems*; Springer: Berlin/Heidelberg, Germany, 1980.
5. Milner, R. *Communication and Mobile Systems: The π-Calculus*; Cambridge University Press: Cambridge, UK, 1999.
6. Fournet, C.; Gonthier, G. The reflexive CHAM and the join-calculus. In Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg Beach, FL, USA, 1996; pp. 372–385. https://doi.org/10.1145/237721.237805.
7. Siewe, F.; Zedan, H.; Cau, A. The Calculus of Context-aware Ambients. *J. Comput. Syst. Sci.* **2011**, *77*, 597–620.
8. Siewe, F. ccaPL: A CCA Programming Environment. Available online: https://fsiewe.afrilocode.net/CCA/index.html (accessed on 23 November 2023).
9. Cardelli, L.; Gordon, A.D. Mobile Ambients. *Theor. Comput. Sci.* **2000**, *240*, 177–213.
10. 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. *Trans. Emerg. Telecommun. Technol.* **2022**, *33*, e3808.
11. Mennicke, S. *A Petri Net Semantics for the Join-Calculus*; Technical Report; TU Braunschweig: Braunschweig, Germany, 2012. Available online: https://www.researchgate.net/profile/Stephan_Mennicke/publication/225026105_A_Petri_Net_Semantics_for_the_Join-Calculus/links/0c960531eab88c058c000000/A-Petri-Net-Semantics-for-the-Join-Calculus.pdf (accessed on 24 February 2024).
12. Yao, W.; He, X. Mapping Petri nets to concurrent programs in CC++. *Inf. Softw. Technol.* **1997**, *39*, 485–495. https://doi.org/10.1016/S0950-5849(97)00006-2.
13. Korečko, T.; Sobota, B. Petri Nets to B-Language Transformation in Software Development. *Acta Polytech. Hung.* **2014**, *11*, 187–206.
14. 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), Sousse, Tunisia, 24–26 March 2013; pp. 1–6. https://doi.org/10.1109/ICITeS.2013.6624067.
15. Marsan, M.A.; Balbo, G.; Conte, G.; Donatelli, S.; Franceschinis, G. Modelling with generalized stochastic Petri nets. *ACM Sigmetr. Perform. Eval. Rev.* **1998**, *26*, 2.
16. van der Aalst, W.M.P. Putting high-level Petri nets to work in industry. *Comput. Ind.* **1994**, *25*, 45–54. https://doi.org/10.1016/0166-3615(94)90031-0.
17. AT&T Labs-Research. Graphviz Distribution. Available online: http://www.research.att.com/sw/tools/graphviz/download.html (accessed on 23 November 2023).
18. Chaum, D. The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability. *J. Cryptol.* **1988**, *1*, 65–75.
19. Mazaré, L. Using unification for opacity properties. In Proceedings of the Workshop on Issues in the Theory of Security (WITS'04), Barcelona, Spain, 3–4 April 2004; pp. 165–176.
20. Bryans, J.W.; Koutny, M.; Ryan, P.Y.A. Modelling Opacity Using Petri Nets. *Electr. Notes Theor. Comput. Sci.* **2005**, *121*, 101–115.
21. Best, E.; Devillers, R.; Hall, J.G. The box calculus: A new causal algebra with multi-label communication. In *Advances in Petri Nets 1992*; Rozenberg, G., Ed.; Springer: Berlin/Heidelberg, Germany, 1992; pp. 21–69.
22. Koutny, M.; Esparza, J.; Best, E. Operational Semantics for the Petri Box Calculus. In *CONCUR '94: Concurrency Theory*; Jonsson, B., Parrow, J., Eds.; Springer: Berlin/Heidelberg, Germany, 1994; pp. 210–225.

23. 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, Grenoble, France, 8–12 April 2002; pp. 49–63.
24. Milner, R. Bigraphs for Petri nets. In *Advanced Course on Petri Nets*; Springer: Berlin/Heidelberg, Germany, 2003; pp. 686–701.
25. Sassone, V.; Sobociński, P. A congruence for Petri nets. *Electron. Notes Theor. Comput. Sci.* **2005**, *127*, 107–120.
26. Leifer, J.J.; Milner, R. Transition systems, link graphs and Petri nets. *Math. Struct. Comput. Sci.* **2006**, *16*, 989–1047.
27. Khomenko, V.; Meyer, R.; Hüchting, R. A polynomial translation of pi-calculus FCPs to safe Petri nets. *Log. Methods Comput. Sci.* **2013**, *9*, 1–36.
28. Khomenko, V.; Germanos, V. Modelling and Analysis Mobile Systems Using-calculus (EFCP). In *Transactions on Petri Nets and Other Models of Concurrency X*; Springer: Berlin/Heidelberg, Germany, 2015; pp. 153–175.
29. Degano, P.; De Nicola, R.; Montanari, U. A distributed operational semantics for CCS based on condition/event systems. *Acta Inform.* **1988**, *26*, 59–91.
30. Goltz, U. CCS and Petri nets. In *LITP Spring School on Theoretical Computer Science*; Springer: Berlin/Heidelberg, Germany, 1990; pp. 334–357.
31. Busi, N.; Gorrieri, R. Distributed semantics for the π-calculus based on Petri nets with inhibitor arcs. *J. Log. Algebr. Program.* **2009**, *78*, 138–162.
32. Dam, M. Model Checking Mobile Processes. *Inf. Comput.* **1996**, *129*, 35–51.
33. Sangiorgi, D.; Walker, D. *The π-Calculus: A Theory of Mobile Processes*; Cambridge University Press: Cambridge, UK, 2001.
34. van Glabbeek, R.; Vaandrager, F. Petri net models for algebraic theories of concurrency. In Proceedings of the PARLE Parallel Architectures and Languages Europe, Eindhoven, The Netherlands, 15–19 June 1987; pp. 224–242.
35. 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), Madrid, Spain, 13–17 July 2020; pp. 1647–1652.
36. Gorrieri, R.; Montanari, U. On the implementation of concurrent calculi in net calculi: Two case studies. *Theor. Comput. Sci.* **1995**, *141*, 195–252.
37. Taubner, D.A. *Finite Representations of CCS and TCSP Programs by Automata and Petri Nets;* Springer: Berlin/Heidelberg, Germany, 1989.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.