Analysing Petri Nets in a Calculus of Context-aware Ambients

François Siewe
Cyber Technology Institute
De Montfort University
Leicester LE1 9BH, U.K.
fsiewe@dmu.ac.uk

Vasileios Germanos
Cyber Technology Institute
De Montfort University
Leicester LE1 9BH, U.K.
vasileios.germanos@gmail.com

Wen Zeng
Cyber Technology Institute
De Montfort University
Leicester LE1 9BH, U.K.
wen.zeng.wz@gmail.com

***Abstract*—This paper proposes an approach to analysing and verifying Petri nets using a Calculus of Context-aware Ambients (CCA). We propose an algorithm that transforms a Petri net into a CCA process. This demonstrates that any system that can be specified in Petri nets can also be specified in CCA. Besides, the system can be analysed and verified using the CCA verification tools. We illustrate the practicality of our approach using a case study of the dining cryptographers problem.**

***Index Terms*—CCA, Petri nets, verification, ccaPL**

I. Introduction

To this date, there is a large number of mathematical formalisations for concurrency theory, and none of them can claim to have priority over others. Petri nets [1] were the first formalism for modelling interacting sequential processes. Then, Robin Milner developed the Calculus of Communicating Systems (CCS), and later the π-calculus [4] which extends CCS with notions of mobility. CCS along with CSP [2], ACP [3] and π-calculus belong to the family of process calculi. The Calculus of Context-aware Ambients (CCA) [5], [23] is inspired from the Calculus of Mobile Ambient (MA) [22] and provides new constructs to enable processes to be aware of the environment in which they are being executed.

This paper proposes an approach to the analysis and verification of Petri nets in CCA. The contributions of the paper is threefold: (i) We propose an algorithm that transforms a Petri net into a CCA process (Sect. IV). 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. (ii) We demonstrate the practicality of our approach using a case study of the dining cryptographers problem (Sect. V). First a dining cryptographers protocol is modelled using a Petri net (Sect. V-A). Then the Petri net is translated into a CCA process. (iii) Finally, we verify the correctness of the dining cryptographers protocol using the ccaPL tool [23] (Sect. V-B).

II. Overview of Petri Nets

Petri nets are a graphical formalism to describe systems whose dynamics are characterised by concurrency, synchronisation, mutual exclusion and conflict [6], [7]. A Petri net consists of *places*, *transitions*, and *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 Fig 1. Any distribution of tokens over the places will represent a configuration of the net called a *marking*. Therefore, a Petri net can be defined formally as a tuple (S, T, F, M_0), where

- S is a finite nonempty set of *places*
- T is a finite nonempty set of *transitions*
- $F \subseteq (S \times T) \cup (T \times S)$ is a set of arcs
- $M_0 : S \to \mathbb{N}$ is the initial marking.

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.

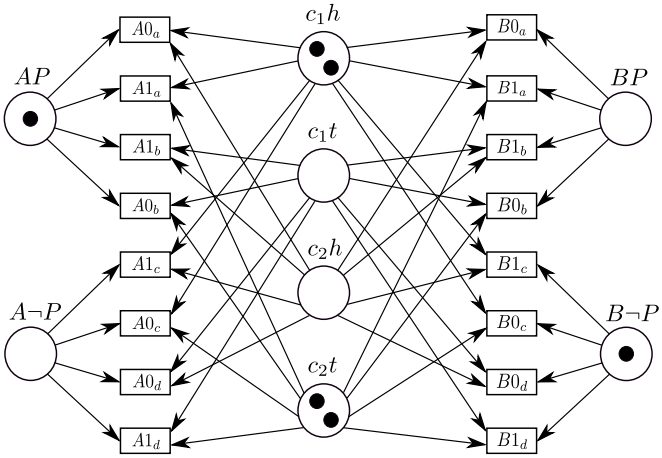

Fig. 1. A Petri net for the Dining Cryptographers problem

Transitions are the active components of a Petri net. A transition may execute if it is enabled, i.e. there are at least one token in all of its input places. The execution of a transition is atomic and consumes one token from each input place, and creates one token in each output place. The execution of Petri nets is non-deterministic: when multiple transitions are enabled at the same time, they will execute in any order.

III. Overview of CCA

Table I 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.

Processes: The process $\mathbf{0}$, aka *inactivity process*, does nothing and terminates immediately. The process $P|Q$ denotes the parallel composition of the processes P and Q. The process $(\texttt{new}\ n)\ P$ 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 copy of P, i.e. $!P$ is equivalent to $P|!P$. The process $n[P]$ denotes an ambient named n whose behaviour is described by the process P. A process of the form $\{P\}$ behaves just like P. A context expression 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 $<\ \texttt{true}\ > M.P$. The *selection* $\texttt{if}\ \ < \kappa_1 > M_1.P_1\ \ldots\ < \kappa_\ell > M_\ell.P_\ell\ \texttt{fi}$ 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. The process '$\texttt{let}\ x_1 = e_1, \ldots, y_\ell = e_\ell\ \texttt{in}\ P$' 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 $\ell \geq 0$.

Capabilities: Ambients exchange messages using the output capability $\alpha\ \texttt{send}(z_1, \ldots, z_\ell)$ to send a list of names $z_1, \ldots, z_\ell$ to a location α, and the input capability $\alpha\ \texttt{recv}(y_1, \ldots, y_\ell)$ to receive a list of names from a location α into the variables $y_1, \ldots, y_\ell$. 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. The capability $\texttt{del}\ n$ deletes an empty n (i.e. $n[\mathbf{0}]$).

Context model: In CCA, a context is modelled as a process with a hole in it. The hole (denoted by $\odot$) in a context C represents the position of a 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 $P \mid n[Q \mid \odot]$. A context-expression (CE, for short) is a formula representing some property of a context model.

Context expressions: The CE `true` always holds. A CE $n = m$ holds if the names n and m are identical. 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 $\kappa_1|\kappa_2$ 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.

IV. Translating a Petri Net into a CCA Process

The Algorithm 1 translates a Petri net (see Sect. II) into a CCA process. The initial marking M_0 assigns to each place a number of tokens. During the execution of the Petri net, the number of tokens of a place increases by 1 when an input transition of the place is fired and decreases by 1 when an output transition to the place is executed. We use a semaphore to guarantee that the dynamic change of the number of tokens in a place is performed in an atomic fashion. This semaphore is described by the ambient `update` in (1). The semaphore is unavailable when the ambient contains the child ambient `on`[`0`] and is available otherwise. Initially, the semaphore is available.

```
update[
 ! recv(). :: recv(s).{on[0] | s :: recv().del on.send().0}
 | send().0
]
```
(1)

The behaviour of the ambient `update` can be explained as follows. A place ambient `s` enters the critical section by performing a capability of the form `update` :: `send`(`s`), and leaves the critical section by executing the capability `update` :: `send`(). The execution of the former capability leads to the creation of a child ambient `on`[`0`] in the ambient `update` (i.e. the semaphore becomes unavailable), and the execution of the latter capability leads to the deletion of that child ambient so the semaphore becomes available again. We use the context-expression `updateOn`() defined in (2) to check if the semaphore `update` is available or not.

$$\texttt{updateOn}() = \texttt{somewhere update}[\texttt{on}[0] \mid \texttt{true}] \quad (2)$$

Similarly to a place, we use a semaphore `lock` defined in (3) to guarantee that the execution of a transition is atomic. The ambient `lock` interacts with a transition ambient exactly the same way as the ambient `update` interacts with a place ambient.

```
lock[
 ! recv(). :: recv(s).{on[0] | s :: recv().del on.send().0}
 | send().0
]
```
(3)

The context-expression in (4) checks if the semaphore `lock` is available or not.

$$\texttt{lockOn}() = \texttt{somewhere lock}[\texttt{on}[0] \mid \texttt{true}] \quad (4)$$

The following subsections explain how places and transitions are modelled as ambients.

A. Modelling Places

In this section, we show how a place can be modelled as an ambient in CCA. For each place $\texttt{s} \in S$ we create an ambient of the same name like in the lines 7-19 in Algorithm 1. Initially, a place `s` contains $M_0(\texttt{s}) = \texttt{n}$ tokens. This is represented in line 8 by the process "`send`(`n`).0". We say that a place is enabled if it contains one or more tokens, i.e. $\texttt{n} > 0$; otherwise the

TABLE I
SYNTAX OF CCA

P, Q	::=	$\mathbf{0} \mid$ '$P \mid Q$' $\mid$ (new n) $P \mid !P \mid n[P] \mid \{P\} \mid < \kappa > M.P \mid$ if $< \kappa_1 > M_1.P_1 \ \ldots \ < \kappa_\ell > M_\ell.P_\ell$ fi $\mid$ let $x_1 = e_1, \ldots, x_\ell = e_\ell$ in P
M	::=	α recv$(y_1, \ldots, y_\ell) \mid \alpha$ send$(z_1, \ldots, z_\ell) \mid$ del n
α	::=	$@ \mid n@ \mid \# \mid n\# \mid :: \mid n :: \mid \epsilon$
κ	::=	true $\mid$ false $\mid$ this $\mid n = m \mid n[\kappa] \mid$ not $\kappa \mid$ '$\kappa_1 \mid \kappa_2$' $\mid \kappa_1$ and $\kappa_2 \mid \kappa_1$ or $\kappa_2 \mid$ next $\kappa \mid$ somewhere κ

place is disabled. This is modelled in CCA by the presence or the absence of the child ambient enabled[0]. Initially, the child ambient exists if $\mathtt{n} > 0$, like in the lines 17-18. Therefore we define a context-expression enabled(s) as in (5).

$$\mathtt{enabled(s)} = \mathtt{somewhere\ s[enabled[0] \mid true]} \quad (5)$$

For each output transition t of the place s, i.e. $(\mathtt{s}, \mathtt{t}) \in F$, the two processes in (6) and (7) are created in line 11 and line 12 respectively.

$$\begin{array}{l} < \mathtt{n} = 1 \text{ and not } \mathtt{updateOn}() > \mathtt{t} :: \mathtt{send}(). \\ \quad \mathtt{del\ enabled.update} :: \mathtt{send}().\mathtt{send}(0).0 \end{array} \quad (6)$$

$$\begin{array}{l} < \mathtt{n} > 1 \text{ and not } \mathtt{updateOn}() > \mathtt{t} :: \mathtt{send}(). \\ \quad \mathtt{let\ x} = \mathtt{n} - 1 \mathtt{\ in\ update} :: \mathtt{send}().\mathtt{send(x)}.0 \end{array} \quad (7)$$

When the place has only 1 token remaining and the semaphore update is available, the process (6) sends the token to the output transition t and sets the place's number of tokens to 0. The child ambient enabled is deleted to reflect that change. The process (7) is executed when the place has more than 1 token. It sends 1 token to the transition t and decreases by 1 the place's number of tokens.

As for each input transition t of the place s, i.e. $(\mathtt{t}, \mathtt{s}) \in F$, the two processes in (8) and (9) are created in line 14 and line 15 respectively.

$$\begin{array}{l} < \mathtt{n} = 0 \text{ and not } \mathtt{updateOn}() > \mathtt{t} :: \mathtt{recv}(). \\ \quad \mathtt{update} :: \mathtt{send}().\mathtt{send}(1).\mathtt{enabled}[0] \end{array} \quad (8)$$

$$\begin{array}{l} < \mathtt{n} > 0 \text{ and not } \mathtt{updateOn}() > \mathtt{t} :: \mathtt{recv}(). \\ \quad \mathtt{let\ x} = \mathtt{n} + 1 \mathtt{\ in\ update} :: \mathtt{send}().\mathtt{send(x)}.0 \end{array} \quad (9)$$

The process (8) is executed when the place has no token. It receives 1 token from the input transition t, sets the place's number of tokens to 1, and then creates the child ambient enabled[0] as the place has become enabled. When the place has 1 or more tokens, the process (9) can be executed to receive an additional token from an input transition t.

B. Modelling Transitions

Similarly to a place, a transition can be modelled as an ambient of the same name. Indeed, Algorithm 1 creates for each transition $\mathtt{t} \in T$ an ambient t as described in the lines 23-35. The behaviour of the ambient is an iterative process, which checks if both the update and the lock semaphores are available (line 25) and that each of the transition's input places is enabled (lines 26-27). This process then enters the critical section with the semaphore lock (line 29) and receives a token from each of the transition's input places (lines 30-31). Finally, the process sends a token to each of the transition's output places (lines 32-33) and releases the semaphore lock (line 34).

In summary, the process generated by Algorithm 1 for a Petri net is the parallel composition of the two semaphores (update and lock), all the place ambients and all the transition ambients.

V. THE DINING CRYPTOGRAPHERS PROBLEM

The standard dining cryptographers problem [8] 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 [9], [10] 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.

A. 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 Fig 1. 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 (10).

$$\{AP, B\neg P\}, \{A\neg P, BP\}, \{A\neg P, B\neg P\} \quad (10)$$

Algorithm 1: From a Petri net to a CCA process

input : A Petri net $N = (S, T, F, M_0)$
output: A CCA process

println((1)); //semaphore "update"
println(" | ");
println((3)); //semaphore "lock"
// Generate the place ambients
for $s \in S$ **do**
    println(" | ");
    println(s+"[");
    println("send("+$M_0(s)$+").0");
    println(" | ! recv(n).if ");
    foreach $t \in T$ *such that* $(s, t) \in F$ **do**
        println("< n=1 and not updateOn()> " + t + "::send().del enabled.update::send().send(0).0");
        println("< n>1 and not updateOn()> " + t + "::send().let x = n - 1 in update::send().send(x).0 ");
    foreach $t \in T$ *such that* $(t, s) \in F$ **do**
        println("< n=0 and not updateOn()> " + t + "::recv().update::send().send(1).enabled[0]");
        println("< n>0 and not updateOn()> " + t + "::recv().let x = n + 1 in update::send().send(x).0");
    println(" fi");
    if $M_0(s) > 0$ **then**
        println(" | enabled[0]");
    println("] ");
// Generate the transition ambients
for $t \in T$ **do**
    println(" | ");
    println(t+"[");
    println(" send().0");
    print(" | ! recv().< not updateOn() and not lockOn()");
    foreach $s \in S$ *such that* $(s, t) \in F$ **do**
        print(" and enabled(" + s+ ")");
    print(" > ");
    print(" lock::send(" + t + ")");
    foreach $s \in S$ *such that* $(s, t) \in F$ **do**
        print("."+ s +"::recv().update::send(" + s + ")");
    foreach $s \in S$ *such that* $(t, s) \in F$ **do**
        print("."+ s +"::send().update::send(" + s + ")");
    println(" .lock::send( ).send().0");
    println("] ");

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 multisets in (11).

$$\{c_1h, c_1h, c_2h, c_2h\}, \{c_1h, c_1h, c_2t, c_2t\}, \{c_1t, c_1t, c_2h, c_2h\}, \{c_1t, c_1t, c_2t, c_2t\} \quad (11)$$

The cross product of the cryptographer markings in (10) and the coin markings in (11) 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.

B. Verification of the Dining Cryptographers Protocol

Algorithm 1 was applied to transform the Petri net in Fig 1 into a CCA process. Due to the paper length requirement, we have not given the CCA process in this paper. However, the ambients corresponding to the place AP and the transition $A0_a$ are given in Table II and Table III, respectively. The ambients for the other places and transitions are similar. The CCA process is analysed using the CCA simulation tool, ccaPL, to ascertain that the Petri net in Fig 1 implements correctly the dining cryptographers protocol. To achieve this, we verify that the behaviour of the Petri net satisfies the protocol for each of the 12 possible initial markings. The results show that the Petri net is a correct implementation of the dining cryptographers protocol. For illustration, we consider the following three scenarios in the case where coin 1 is head and coin 2 is tail, i.e. $\{c1h, c1h, c2t, c2t\}$.

a) Scenario 1: Alice paid the bill, i.e. $\{AP, B\neg P\}$. The execution of the system in ccaPL generates the execution graph in Fig 2. The ambients involved in the execution are listed at the top of the graph, and the arrows represent the communications between the ambients. The graph shows that the only transitions fired in this case are A1a and B0c. This means that Alice says the coins 'agree' and Bob says the coins 'disagree', which is the expected output of the protocol.

b) Scenario 2: Bob paid the bill, i.e. $\{A\neg P, BP\}$. The corresponding execution graph in Fig 3 shows that the only transitions fired in this case are A0c and B1a. This means that Alice says the coins 'disagree' and Bob says the coins 'agree', which is the correct output of the protocol.

c) Scenario 3: Alice and Bob did not pay the bill, i.e. $\{A\neg P, B\neg P\}$. The corresponding execution graph in Fig 4 shows that the only transitions fired in this case are A0c and B0c. This means that both Alice and Bob say the coins 'disagree', which is conform to the protocol.

TABLE II
THE AMBIENT MODELLING THE PLACE AP

```
AP[
   send(1).0
  | ! recv(n).if
        < n = 1 and not (updateOn()) > A0a::send().del enabled.update::send().send(0).0
        < n > 1 and not (updateOn()) > A0a::send().let x = (n - 1) in update::send().send(x).0
        < n = 1 and not (updateOn()) > A0b::send().del enabled.update::send().send(0).0
        < n > 1 and not (updateOn()) > A0b::send().let x = (n - 1) in update::send().send(x).0
        < n = 1 and not (updateOn()) > A1a::send().del enabled.update::send().send(0).0
        < n > 1 and not (updateOn()) > A1a::send().let x = (n - 1) in update::send().send(x).0
        < n = 1 and not (updateOn()) > A1b::send().del enabled.update::send().send(0).0
        < n > 1 and not (updateOn()) > A1b::send().let x = (n - 1) in update::send().send(x).0
    fi
  | enabled[ 0 ]
]
```

TABLE III
THE AMBIENT MODELLING THE TRANSITION $A0_a$

```
A0a[
   send().0
  | ! recv().< not updateOn() and not lockOn() and enabled(AP) and enabled(c1h) and enabled(c2h) >
      lock::send(A0a).AP::recv().update::send(AP).c1h::recv().update::send(c1h).
      c2h::recv().update::send(c2h).lock::send().send().0
]
```

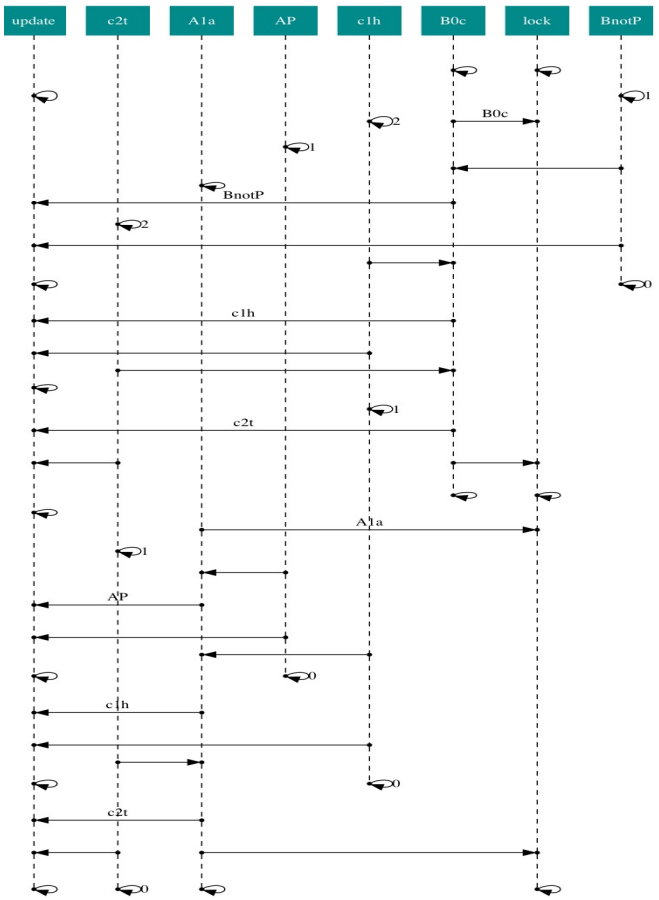

Fig. 2. Alice paid and Bob did not pay ($\{AP, B\neg P\}$)

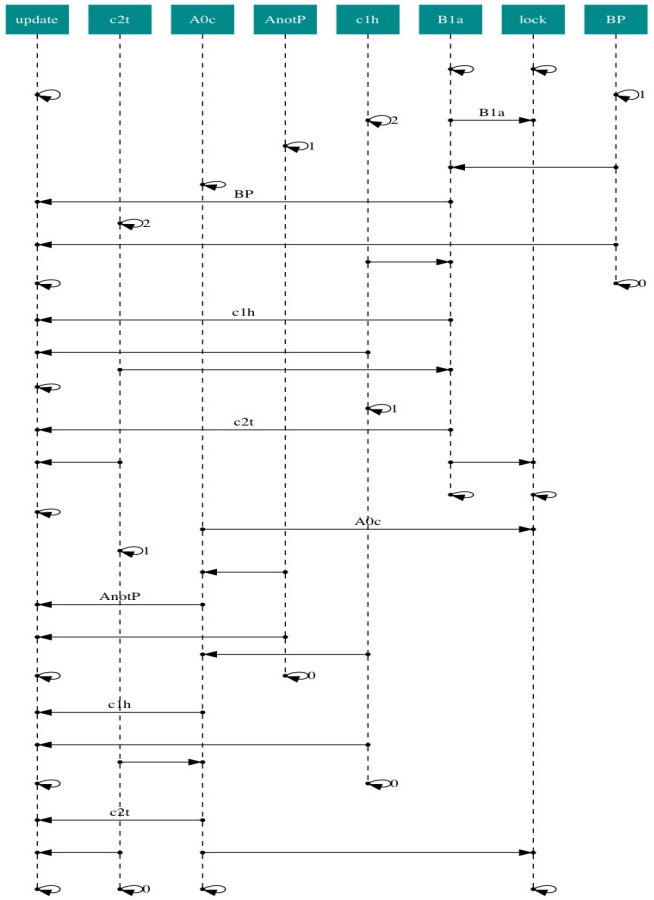

Fig. 3. Alice did not pay and Bob paid ($\{A\neg P, BP\}$)

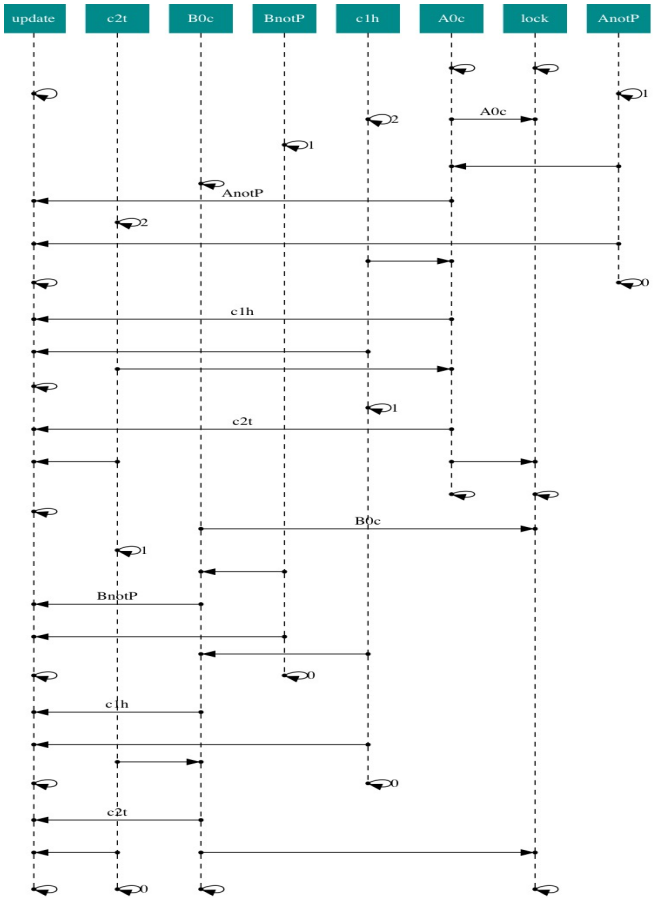

Fig. 4. Alice did not pay and Bob did not pay ($\{A\neg P, B\neg P\}$)

VI. Related Work

There has been a substantial amount of research that adapts and relates features of process algebras to Petri nets. Petri Box calculus [11], [12], 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. [13] proposed a translation from Condition/Event (C/E) nets to Circal process algebra based on a binary composition and hiding operators. Moreover, in [14]–[16], frameworks that endow Petri nets with labelled transition systems are presented, applying techniques come from process algebras. In particular, in [15], 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 [20]; with application to the verification of mobile systems [21]. For instance, [17], [18] proposed a translation of CCS into Petri nets, while [19] presented a distributed semantics for π-calculus, based on Petri nets.

VII. Conclusion

This paper presented an approach to the analysis and verification of Petri nets in CCA. An algorithm was proposed that transforms a Petri net into a CCA process. That process can then be analysed and verified using the CCA verification tools. The approach was applied to a real-world case study of the dining cryptographers problem.

References

[1] C.A. Petri, "Kommunikation mit Automaten," Ph.D. thesis, Fakultät für Mathematik und Physik, Technische Hochschule Darmstadt, Darmstadt, Germany, 1962.

[2] C.A.R Hoare, "Communicating Sequential Processes," Prentice-Hall, Inc., USA, 1985.

[3] J. C. M. Baeten, "Procesalgebra : een formalisme voor parallelle, communicerende processen," Kluwer, Deventer, 1986

[4] R. Milner, "Communicating and Mobile Systems: The π-Calculus," Cambridge University Press, USA, 1999.

[5] F. Siewe, H. Zedan, A. Cau, "The Calculus of Context-aware Ambients", J. Comput. System Sci., 77(4), pp. 597–620, 2011.

[6] M.A. Marsan, G. Balbo, G. Conte, S. Donatelli, G. Franceschinis, "Modelling with generalized stochastic Petri Nets," Wiley Series on Parallel Computing, 1995.

[7] T. Murata, "Petri nets: Properties, analysis and applications," vol. 77(4). Proceedings of the IEEE, pp. 541–580, 1989.

[8] D. Chaum, "The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability," vol. 1(1). J. Cryptol., Springer-Verlag, Berlin, Heidelberg, pp. 65–75, 1988.

[9] L. Mazaré, "Using unification for opacity properties," In Proceedings of the Workshop on Issues in the Theory of Security (WITS '04), pp. 165–176, 2004.

[10] J. W. Bryans, M. Koutny, P. Y. A. Ryan, "Modelling Opacity Using Petri Nets," vol. 121. Electr. Notes Theor. Comput. Sci. Proceedings of the 2nd International Workshop on Security Issues with Petri Nets and other Computational Models (WISP 2004), pp. 101–115, 2005.

[11] E. Best, R. R. Devillers, J G. Hall, "The box calculus: a new causal algebra with multi-label communication," Advances in Petri Nets: The DEMON Project, 1992.

[12] M. Koutny, J. Esparza, E. Best, "Operational Semantics for the Petri Box Calculus," Proceedings of the Concurrency Theory, CONCUR '94, Springer-Verlag, pp. 210–225, 1994.

[13] A. Cerone, "Implementing Condition/Event Nets in the Circal Process Algebra," Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering, FASE '02, Springer-Verlag, pp. 49–63, 2002.

[14] J. Leifer, R. Milner, "Transition systems, link graphs and Petri nets," vol. 16(6). Mathematical Structures in Computer Science, pp. 989–1047, 2006.

[15] R. Milner, J. Desel, W. Reisig, G. Rozenberg, "Bigraphs for Petri Nets," Lectures on Concurrency and Petri Nets: Advances in Petri Nets, Springer Berlin Heidelberg, 2004.

[16] V. Sassone, P. Sobociński, "A Congruence for Petri Nets," vol. 127(2). Electronic Notes in Theoretical Computer Science, Proceedings of the Workshop on Petri Nets and Graph Transformations (PNGT 2004), pp. 107–120, 2005.

[17] P. Degano, R. De Nicola, U. Montanari, "A distributed operational semantics for CCS based on condition/event systems," vol. 26. Acta Informatica, pp. 59–91, 1988.

[18] U. Goltz, "CCS and petri nets," Semantics of Systems of Concurrent Processes, Springer Berlin Heidelberg, pp. 334–357, 1990.

[19] N. Busi, R. Gorrieri, "A Petri net semantics for π-calculus," CONCUR '95: Concurrency Theory, Springer Berlin Heidelberg, pp. 145–159, 1995.

[20] V. Khomenko, R. Meyer, R. Hüchting, "A Polynomial Translation of pi-calculus FCPs to Safe Petri Nets," vol. 9(3). Logical Methods in Computer Science, 2013.

[21] V. Khomenko, V. Germanos, "Modelling and Analysis Mobile Systems Using π-calculus (EFCP)," Transactions on Petri Nets and Other Models of Concurrency X, Springer Berlin Heidelberg, pp. 153–175, 2015.

[22] L. Cardelli and A. Gordon, "Mobile Ambients," Theoretical Computer Science, VOL. 240, pp. 177–213, 2000.

[23] "ccaPL," http://www.cse.dmu.ac.uk/~fsiewe/CCA/cca.html, accessed 20 March 2020.