When to Move to Transfer Nets On the limits of Petri nets as models for process calculi

Gianluigi Zavattaro

When to Move to Transfer Nets

On the limits of Petri nets as models for process calculi

Gianluigi Zavattaro

Dep. of Computer Science and Engineering, Mura A. Zamboni 7, 40127 Bologna
Focus Team, University of Bologna & INRIA, Italy
gianluigi.zavattaro@unibo.it

Abstract. Pierpaolo Degano has been an influential pioneer in the investigation of Petri nets as models for concurrent process calculi (see e.g. the well-known seminal work by Degano–De Nicola–Montanari also known as DDM88). In this paper, we address the limits of classical Petri nets by discussing when it is necessary to move to the so-called Transfer nets, in which transitions can also move to a target place all the tokens currently present in a source place. More precisely, we consider a simple calculus of processes that interact by generating/consuming messages into/from a shared repository. For this calculus classical Petri nets can faithfully model the process behavior. Then we present a simple extension with a primitive allowing processes to atomically rename all the data of a given kind. We show that with the addition of such primitive it is necessary to move to Transfer nets to obtain a faithful modeling.

1 Introduction

The study of the relationship between two relevant computational models like process calculi and Petri nets has attracted a lot of attention within the concurrency theory community since the second half of the 80s. One of the initial motivations for associating to process calculi a Petri net semantics was to enrich the formers with a truly concurrent semantics, instead of the interleaving semantics usually given in terms of a labeled transition system. In particular, one of the most influential work along this line of research is the seminal work by Degano, De Nicola, and Montanari [7] which was inspired by the observation that in Milner's CCS [15], under the classical interleaving semantics, *"causal dependencies remain non-recoverable (for instance the behavior of $\alpha|\beta + \alpha.\beta$ and that of $\alpha|\beta$ cannot be differentiated)"*. Another motivation for equipping process calculi with Petri net semantics is to resort to analysis or decidability results well-known for Petri nets. For instance, in [4] Petri nets were used to prove the decidability of termination in a CCS-like process calculus with asynchronous communication via a shared repository of data. In fact, Petri nets represent one of the most interesting models for infinite state systems in which properties like reachability, coverability, boundedness, as well as many others, are still decidable (see [10] for a nicely written and comprehensive survey about decidability results for Petri nets).

In this paper we focus on a specific research problem that we have encountered in several papers dedicated to the study of the decidability of properties, like *termination* and *divergence*, in several classes of process calculi. By termination in this paper we mean the existence of a completed finite computation, while by divergence we mean the existence of an infinite computation. The proof technique that we frequently adopted is based on translations from the process calculi of interest to Petri nets, in order to resort to already known decidability results for Petri nets.

In the already mentioned paper [4] we considered a calculus of processes communicating via a common data repository by means of output, input, read, and test for absence primitives. We first proved that, if data are guaranteed to be in the data space immediately after the execution of an output operation, the calculus is Turing complete (hence termination is undecidable). On the contrary, if output operations are asynchronous, in the sense that emitted data become available only after an unpredictable delay, the calculus is no longer Turing powerful because termination turns out to be decidable. The proof exploited a non-trivial Petri net semantics for the asynchronous version of the calculus.

In other papers we had to consider extended versions of Petri nets. For instance, in [5] we considered a similar calculus with processes communicating via a common data space, but with a *notify* primitive instead of the test for absence. The notify operation allows processes to register their interest in the emission of a given kind of datum; when such a datum is produced, all the registered processes receive a corresponding notification. For this calculus, we considered Petri nets with Transfer arcs, that are arcs able to move all the tokens present in a source place to a corresponding target place. On the contrary, in [6] we considered Petri nets with Reset arcs, which are used to remove all the tokens currently available in a place. In that paper, we considered a timed version of shared data space communication, in which data have an associated time out and must be cancelled when they expire. More recently, in [8] we considered again Transfer Petri nets, but in the rather different context of BioAmbients, a calculus where processes are placed inside nested locations and can execute operations for entering, exiting or merging ambients.

Intuitively, in that papers we had to move to extended versions of Petri nets due to the difficulty in the definition of appropriate encodings of the considered calculi into classical Petri nets. The increased expressive power of Transfer or Reset nets w.r.t. classical Petri nets has been investigated in [9]. In particular, we have that properties like reachability and termination are decidable for classical Petri nets while this is not the case for Transfer and Reset nets; properties like boundedness is decidable for classical and Transfer nets while this is not the case for Reset nets; and finally properties like divergence or coverability are decidable for all of these classes of nets.[1]

[1] In [9] a slightly different terminology is used: termination refers to the guarantee that all the computations completes, thus corresponding to the negation of the property that we call divergence in this paper.

From a formal point of view, there are cases that strictly require to move from classical to extended Petri nets to equip a process calculus with a faithful Petri net semantics. To clarify this specific point, in this paper we fully formalize in a simplified setting one of these cases. More precisely, we identify a basic calculus of processes performing input and output operations on a shared data space, extended with a primitive for renaming all the data of a given kind. We first show that for the basic version of the calculus without renaming, it is possible to define a faithful encoding by using classical Petri nets. By faithful encoding, here we mean that there exists a one-to-one correspondence between process reductions in the calculus and transition firings in the Petri net. Then we move to the version of the calculus with the renaming primitive, and we show that termination is undecidable for this version of the calculus. This undecidability result shows that there exists no recursive encoding from the calculus to classical Petri nets that preserves and reflects at least termination. Then we consider Transfer Petri nets, and we show that with this extended version of Petri nets it is again possible to define a faithful encoding. This also proves that divergence, as well as boundedness and coverability, are decidable for the calculus with the renaming primitive.

Structure of the paper. In Section 2 we define the DS calculus, the initial version of our language for processes communicating via input and output operations on a common repository, and we present a faithful modeling of the DS calculus into classical Petri nets. In Section 3, inspired by the *copy-collect* primitive proposed in [17], we define the RenDS calculus that includes a new primitive $ren(a, b)$ that renames to b all the instances of a in the data space. For this calculus we prove the undecidability of termination (hence also the impossibility to equip RenDS with a termination preserving classical Petri net semantics) and then we show a faithful modeling in terms of Transfer Petri nets. Section 4 draws some concluding remarks.

2 The DS calculus

In this section we present the syntax and the semantics of a simple calculus of processes communicating by introducing and consuming data into/from a shared repository.

Definition 1 (Processes). *Let Name, ranged over by a, b, ..., be a denumerable set of names. Processes are defined by the following grammar:*

$$\begin{array}{lcl} \alpha & ::= & in(a) \quad | \quad out(a) \\ P & ::= & \sum_{i \in I} \alpha_i.P_i \quad | \quad !\alpha.P \quad | \quad P|P \end{array}$$

The basic process actions are $in(a)$ and $out(a)$ denoting the consumption/emission of one instance of datum a from/into the shared data space. The term $\sum_{i \in I} \alpha_i.P_i$ denotes a process ready to perform any of the action α_i, and then proceed by executing the corresponding continuation P_i. We use $\mathbf{0}$ to denote such process

in case $I = \emptyset$, and we will usually omit trailing $\mathbf{0}$. The replicated process $!\alpha.P$ performs an initial action α and then spawns the continuation P by keeping $!\alpha.P$ in parallel. Two parallel processes P and Q are denoted with $P|Q$.

Example 1. As an example, we consider a simple producer-consumer system:

$$!in(prod).out(job).in(done).\big(\ out(prod) + out(end)\ \big)\ |$$
$$!in(cons).\big(\ in(job).out(done).out(cons) + in(end)\ \big)$$

The producer process is triggered by a *prod* datum; it produces a *job* request, waits for the corresponding *done* message, and then nondeterministically decides whether to continue with another job production phase or complete by emitting the message *end*. The consumer process is triggered by a *cons* datum; it consumes a *job* request, produces the corresponding *done* messages, and repeats until an *end* message is received instead of a *job* request.

A system includes also a shared data space where data are stored and consumed.

Definition 2 (Systems). *A system is a pair $\langle P, \mathcal{S}\rangle$ where P is a process and $\mathcal{S}$ is a multiset over Name.*

In the following, $\uplus$ stands for multiset union and with $\mathcal{S}(a)$ we denote the number of instances of a in the multiset $\mathcal{S}$.

Example 2. Let P be the process defined in Example 1. The system

$$\big\langle P, \{prod, cons\}\big\rangle$$

represents the initial state of the produced-consumer system where the data space contains the two *prod* and *cons* data necessary to initially trigger the producer and consumer processes, respectively.

In order to define the operational semantics of systems we first define a labeled transition system on processes which indicates the possible input and output actions, and then we define a transition relation on systems which defines the effect of the execution of process actions on the shared data space.

Definition 3 (Process semantics). *The semantics of processes is defined by a labeled transition system on processes with two kinds of labels: $in(a)$ and $out(a)$. The transition system is the least one satisfying the axioms and rules reported in Table 1.*

The `PRE` rule simply allows a sum process to execute one of its initial action and then continue with the corresponding continuation. `REPL` allows $!\alpha.P$ to execute α, spawn an instance of the continuation P, and keep $!\alpha.P$ in parallel. Finally, `PAR` allows a parallel process to execute an action.

We can now complete the definition of the operational semantics taking into account systems.

$$\texttt{PRE}: \frac{j \in I}{\sum_{i \in I} \alpha_i.P_i \xrightarrow{\alpha_j} P_j} \qquad \texttt{REPL}: !\alpha.P \xrightarrow{\alpha} !\alpha.P \mid P \qquad \texttt{PAR}: \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q}$$

Table 1. The transition system for processes (symmetric rule of PAR omitted).

$$\frac{P \xrightarrow{in(a)} P'}{\langle P, \mathcal{S} \uplus a \rangle \to \langle P', \mathcal{S} \rangle} \qquad \frac{P \xrightarrow{out(a)} P'}{\langle P, \mathcal{S} \rangle \to \langle P', \mathcal{S} \uplus a \rangle}$$

Table 2. The reduction relation for systems (brackets in singletons are omitted).

Definition 4 (System semantics). *The semantics of systems is defined by the minimal transition system satisfying the rules in Table 2.*

The transitions for systems simply allows processes to consume and introduce data from/to the shared data space.

Example 3. Let $\langle P, \{prod, cons\} \rangle$ be the producer-consumer system defined in Example 2. According to the transition system in Definition 4, such system can generate sequences of emissions and consumptions of *prod* and *cons* messages, combined with the emissions and consumptions of *job* and *done* data. Such sequences of actions could be either infinite, or –in case they are maximal, i.e., they cannot be extended– terminating with the production and consumption of an *end* datum. In this last case, the data space is guaranteed to be finally empty because all the *job* requests are consumed, as well as all the corresponding *done* acknowledgement, and also the final *end* message is removed.

We now consider a Petri net semantics for this simple calculus. We first recall the classical definition of Petri nets, then we discuss how to use them to model the behavior of systems of our DS calculus.

Definition 5 (Petri nets). *A* Petri net *is a tuple* $N = (S, T, \boldsymbol{m_0})$*, where* S *and* T *are finite sets of* places *and* transitions*, respectively. A finite multiset over the set* S *of places is called a* marking*, and* $\boldsymbol{m_0}$ *is the initial marking. Given a marking* $\boldsymbol{m}$ *and a place* p*, we say that the place* p *contains a number of* tokens *equal to the number of instances of* p *in* $\boldsymbol{m}$ *(written* $\boldsymbol{m}(p)$*). A transition* $t \in T$ *is a pair of markings denoted with* ${}^\bullet t$ *and* $t^\bullet$ *(the preset and postset of the transition, respectively). A transition* t *(also denoted with* ${}^\bullet t \rightarrowtail t^\bullet$*) can fire in the marking* $\boldsymbol{m}$ *if* ${}^\bullet t \subseteq \boldsymbol{m}$ *(where* $\subseteq$ *is multiset inclusion); upon transition firing the new marking of the net becomes* $\boldsymbol{n} = (\boldsymbol{m} \setminus {}^\bullet t) \uplus t^\bullet$ *(where* $\setminus$ *and* $\uplus$ *are the difference and union operators for multisets, respectively). This is written as* $\boldsymbol{m} \mapsto \boldsymbol{n}$*.*

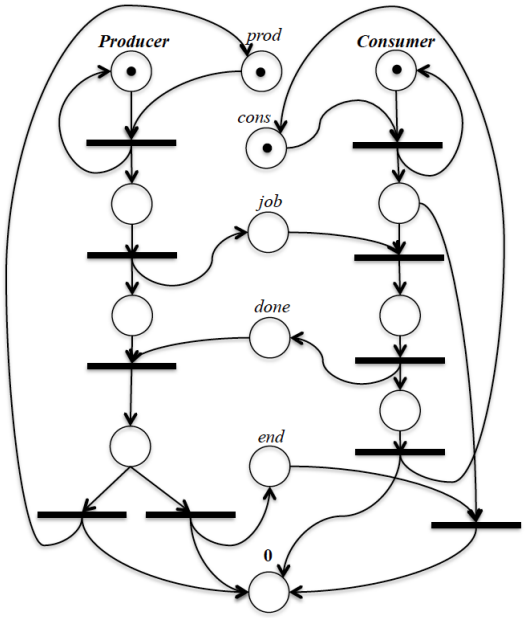

Fig. 1. Petri net for the producer-consumer example

$$dec(\sum_{i\in I} \alpha_i.P_i) = \{\sum_{i\in I} \alpha_i.P_i\} \qquad dec(!\alpha.P) = \{!\alpha.P\} \qquad dec(P|Q) = dec(P) \uplus dec(Q)$$

Table 3. Process decomposition function

Petri nets are graphically depicted by representing places with circles and transitions with rectangles. Edges connect circles to rectangles: an edge from a circle to a rectangle indicates a place in the preset of a transition, while an edge from a rectangle to a circle indicates a place in the postset of a transition. Dots inside circles represent tokens inside places.

Example 4. In Figure 1 we depict a Petri net representing the behavior of the producer-consumer system defined in Example 3. The behavior of the producer process is reported on the left, while the consumer is on the right. Places in the middle of the figure represent the possible data in the data space (*prod*, *cons*, *job*, *done* and *end*) and the trailing empty **0** process.

We now discuss how to translate systems of the DS calculus into Petri nets. The idea is to represent sequential processes and data by means of tokens inside corresponding places. Sequential processes are of two possible kinds: $\sum_{i\in I} \alpha_i.P_i$ and $!\alpha.P$. Parallel processes will be represented by a multiset of tokens, one for

$$\texttt{IN}: \quad \{\textstyle\sum_{i\in I} \alpha_i.P_i, a\} \rightarrowtail dec(P_j) \quad j \in I, \alpha_j = in(a)$$

$$\texttt{OUT}: \quad \{\textstyle\sum_{i\in I} \alpha_i.P_i\} \rightarrowtail dec(P_j) \uplus \{a\} \quad j \in I, \alpha_j = out(a)$$

$$\texttt{REPIN}: \quad \{!in(a).P, a\} \rightarrowtail dec(P) \uplus \{!in(a).P\}$$

$$\texttt{REPOUT}: \quad \{!out(a).P\} \rightarrowtail dec(P) \uplus \{a\} \uplus \{!out(a).P\}$$

Table 4. Petri net transitions

each sequential process composed in parallel. Formally, let P be a process, with $dec(P)$ we denote the multiset defined in Table 3. The possible Petri net transitions, denoted with $\mathcal{T}$, are defined in Table 4. The execution of an $in(a)$ action consumes a token from place a, while an $out(a)$ action produces such a token. After execution of the action, tokens are produced in the places corresponding to the process continuation. Notice that the transitions involving the replicated processes $!in(a).P$ or $!out(a).P$ consume and then reproduce the corresponding tokens; in this way the tokens remain available for future transitions involving those replicated processes.

Definition 6. *Let $\langle P, \mathcal{S}\rangle$ be a system. We define the Petri net $Net(P, \mathcal{S}) = (P, T, \boldsymbol{m_0})$ as follows:*

- $S = \{Q \mid Q \text{ is a sequential process in } P\} \cup \{a \mid a \text{ occurs in } \mathcal{S} \text{ or in } P\}$
- $T = \{c \rightarrowtail p \in \mathcal{T} \mid dom(c) \subseteq S\}$
- $\boldsymbol{m_0} = dec(P) \uplus \mathcal{S}$

Example 5. It is easy to see that the Petri net in Figure 1 corresponds to $Net(P, \{prod, cons\})$ where $\langle P, \{prod, cons\}\rangle$ is the system in Example 2.

The strict correspondence between the process calculus and the Petri net semantics is formalized as follows. We omit the proof of this correspondence result because standard.

Proposition 1. *Let $\langle P, \mathcal{S}\rangle$ be a system, and $Net(P, \mathcal{S}) = (P, T, \boldsymbol{m_0})$ be the corresponding Petri net. Let Q be a process composed of sequential processes occurring in P, and $\mathcal{V}$ be a multiset of data occurring in P or in $\mathcal{S}$. We have that $\langle Q, \mathcal{V}\rangle \rightarrow \langle Q', \mathcal{V}'\rangle$ if and only if $dec(Q) \uplus \mathcal{V} \mapsto dec(Q') \uplus \mathcal{V}'$ in $Net(P, \mathcal{S})$.*

3 The RenDS calculus: shared data space with renaming

We now consider an extension of the DS calculus with a primitive for renaming all the data of a given kind. This renaming mechanism is inspired by the *copy-collect* primitive proposed in [17]. In that paper, a language with multiple data

spaces is considered, and the *copy-collect* primitive is used to move all the data matching a given pattern from a source space to a target space. As in DS we have only one data space, we adapt this primitive by considering an operation $ren(a, b)$ which changes to b all the instances of datum a in the data space. We call RenDS this extended calculus.

The syntax of processes is the same as in Definition 1 with the addition of a new action:

$$\alpha \quad ::= \quad \cdots \quad | \quad ren(a, b)$$

Example 6. We now consider an alternative version of the producer-consumer example in which the producer can repeatedly produce job requests without waiting for the indication that the previous job request has been accomplished. When the consumer starts, only the job requests already issued will be served while subsequent requests will remain pending.

$$!in(prod).out(job).\big(\ in(done) \mid out(prod)\ \big) \ \mid$$
$$in(cons).ren(job, todo).\big(\ !in(todo).out(done) \mid in(prod)\ \big)$$

The producer process is triggered by a *prod* datum; it produces a *job* request and then waits for the *done* message, but in parallel reproduces the *prod* datum to repeatedly issue an arbitrary number of requests. The consumer process is triggered by a *cons* datum; as a first action renames all the *job* requests in *todo*, and then it serves the *todo* activities. Subsequent *job* requests will remain pending. The consumer has also the ability to stop the producer by consuming the *prod* datum.

The semantics for processes is defined as in Table 1 with the addition of the label $ren(a, b)$, while the semantics of systems is defined by the two rules in Table 2 with the addition of the following one for the renaming primitive:

$$\frac{P \xrightarrow{ren(a,b)} P' \qquad \mathcal{S}'(a) = 0 \qquad \mathcal{S}'(b) = \mathcal{S}(a) + \mathcal{S}(b) \qquad \forall c \notin \{a, b\}.\mathcal{S}'(c) = \mathcal{S}(c)}{\langle P, \mathcal{S}\rangle \to \langle P', \mathcal{S}'\rangle}$$

Example 7. Let P be the process defined in Example 6. The initial system is $\langle P, \{prod, cons\}\rangle$ with *prod* and *cons* in the data space, to trigger the producer and the consumer, respectively. It is worth to note that this system can either have an infinite computation in which infinitely many *job* requests are issued, or it terminates in such a way that the final system will contain the same number of instances of the $in(done)$ process and of the *job* datum, representing the pending requests issued after the the consumer transforms the current *job* requests into *todo* data.

We now consider the problem of modeling with Petri nets the processes of the new calculus RenDS. Intuitively, this translation is not easy to be defined due to the impossibility to perform in the Petri net "global" actions that act atomically on all the tokens currently present in a place. In fact, in classical nets, transitions

always consume the same amount of tokens. On the contrary, an operation like $ren(a, b)$ has an effect which is dependent on the current system state, because all the a data must atomically be renamed into b.

In order to formalize this negative result, i.e., classical Petri nets are not sufficiently expressive to model the new calculus with renaming, we proceed as follows. We start from the observation that the existence of a terminating computation is decidable for Petri nets [13], then we prove that termination is undecidable in RenDS. Hence we can conclude that there exists no computable termination preserving encoding of RenDS into classical Petri nets. It is worth to observe that for the DS calculus in Section 2, the presented encoding into Petri nets obviously preserves termination (trivial corollary of Proposition 1), hence termination is decidable in DS.

We prove the undecidability of termination in RenDS by reduction from the halting problem in Random Access Machines (RAMs). A RAM [18], denoted in the following with R, is a computational model composed of a finite set of registers $r_1, \ldots, r_n$, that can hold arbitrary large natural numbers, and by a program composed by indexed instructions $(1 : I_1), \ldots, (m : I_m)$, that is a sequence of simple numbered instructions, like arithmetical operations (on the contents of registers) or conditional jumps. An internal state of a RAM is given by $(i, c_1, \ldots, c_n)$ where i is the program counter indicating the next instruction to be executed, and $c_1, \ldots, c_n$ are the current contents of the registers $r_1, \ldots, r_n$, respectively.

Without loss of generality, we assume that the registers contain the value 0 at the beginning and at the end of the computation, and that the execution of the program begins with the first instruction $(1 : I_1)$. The assumption on the initially empty registers is justified by the possibility to add to programs a prologue that introduces the desired values in the registers, while the assumption on the finally empty registers is justified by the possibility to add to programs a conclusion that decrements all the registers to 0 before halting. In other words, the initial configuration is $(1, 0, \ldots, 0)$. The computation continues by executing the other instructions in sequence, unless a jump instruction is encountered. The execution stops when the instruction *Halt* is reached. More formally, we indicate by $(i, c_1, \ldots, c_n) \rightarrow_R (i', c'_1, \ldots, c'_n)$ the fact that the configuration of the RAM R changes from $(i, c_1, \ldots, c_n)$ to $(i', c'_1, \ldots, c'_n)$ after the execution of the i-th instruction.

In [16] it is shown that the following two instructions are sufficient to model every recursive function:

- $(i : Succ(r_j))$: adds 1 to the content of register r_j;
- $(i : DecJump(r_j, s))$: if the contents of register r_j is not zero then decreases it by 1 and go to the next instruction, otherwise jumps to instruction s.

We start by presenting how to encode RAM instructions into processes of the RenDS calculus:

$$\begin{array}{lcl} [\![(i : Succ(r_j))]\!] & : & !in(p_i).out(r_j).out(p_{i+1}) \\ [\![(i : DecJump(r_j, s))]\!] & : & !in(p_i).\big(\ in(r_j).out(p_{i+1}) \ + \ ren(r_j, loop).out(p_s)\ \big) \\ [\![(i : Halt)]\!] & : & in(p_i).!in(loop).out(loop) \end{array}$$

The idea is to represent the content of the register r_j with a corresponding number of instances of the datum r_j in the data space. The program counter is modeled by a datum p_i indicating that the i-th instruction is the next one to be executed. The modeling of the i-th instruction always starts with the consumption of the p_i datum. An increment instruction on r_j simply produces one datum r_j, while a decrement consumes such a datum. A faithful modeling of a test for zero on r_j should be able to detect the absence of data r_j. As there are no primitives for performing such a test, we consider a nondeterministic modeling according to which a test for zero on r_j could be successful even if the data space contains some r_j instances. But if this occurs, we use the renaming primitive to atomically rename all the currently present r_j data into *loop* data. The presence of *loop* data forbids the possibility for the RAM modeling to terminate: in fact, the encoding of a $Halt$ instruction enters in an infinite loop in case there is at least one datum *loop* in the data space.

We now present the full definition of our encoding. Let R be a RAM with m instructions, and let $(i, c_1, \ldots, c_n)$ be one of its configurations. With

$$[\![(i, c_1, \ldots, c_n)]\!]_R \ = \langle \prod_{1 \leq i \leq m} [\![(i : I_i)]\!], \{p_i, \underbrace{r_1, \cdots, r_1}_{c_1 \ times}, \cdots, \underbrace{r_n, \cdots, r_n}_{c_n \ times}\} \rangle$$

we denote the system representing the configuration $(i, c_1, \ldots, c_n)$.

We now prove that our encoding is termination preserving, from which we conclude the undecidability of termination for the RenDS process calculus.

Theorem 1. *Let R be a RAM. We have that R terminates if and only if $[\![(1, 0, \ldots, 0)]\!]_R$ terminates.*

Proof. We start with the *only if* part. Assume R terminates. We have that $[\![(1, 0, \ldots, 0)]\!]_R$ can faithfully reproduce the terminating computation of R without producing any *loop* data. This computation of $[\![(1, 0, \ldots, 0)]\!]_R$ terminates because the encoding of the $Halt$ instruction definitely consumes the program counter datum, and remains blocked trying to consume a *loop* datum.

We now consider the *if* part. Assume that $[\![(1, 0, \ldots, 0)]\!]_R$ terminates. Every terminating computation completes by reaching the encoding of a $Halt$ instruction (all the other instructions are replicated and reproduce the program counter datum before terminating) and never produce any *loop* datum (otherwise the enconding of the $Halt$ instruction perform an infinite loop). The RAM R can execute an equivalent computation reaching a $Halt$ instruction because the increment and decrement instructions can be obviously mimicked, as well as the test for zero actions. In fact, such actions are surely executed when the tested register is empty, otherwise a *loop* datum would have been produced. □

As a trivial corollary, from the undecidability of the halting problem for RAMs we can conclude the undecidability of termination for the RenDS calculus.

The undecidability of termination implies the impossibility to define a termination preserving encoding of the RenDS calculus into Petri nets. We can however obtain a correspondence result if we move to Petri nets with Transfer arcs [12], allowing for the atomic movement of all the tokens currently present in a source place to a target place. Transfer nets represent an interesting extension of Petri nets; in [9] it has been proven that they are more expressive than classical Petri nets because reachability (as well as termination) is no longer decidable, while other properties like divergence, boundedness or coverability are still decidable.

Definition 7 (Petri nets with Transfer arcs). *A Petri net with Transfer arcs is defined as a Petri net $N = (S, T, \boldsymbol{m_0})$ with the difference that the transitions t in T are now triples, containing besides the preset ${}^\bullet t$ and the postset $t^\bullet$ also a partial function t_f from places to places (transitions are now denoted with ${}^\bullet t \overset{t_f}{\rightarrowtail} t^\bullet$). Given a transition ${}^\bullet t \overset{t_f}{\rightarrowtail} t^\bullet$ we assume that $dom(t_f) \cap {}^\bullet t = \emptyset$, i.e. the places in the preset of a transition t, cannot be source places for transfer arcs of t. As for Petri nets, a transition t can fire in the marking $\boldsymbol{m}$ if ${}^\bullet t \subseteq \boldsymbol{m}$; upon transition firing the new marking becomes $\boldsymbol{n}$ where*

$$\boldsymbol{n}(p) = \begin{cases} \boldsymbol{m}(p) - {}^\bullet t(p) + t^\bullet(p) + \sum_{p'.t_f(p')=p} \boldsymbol{m}(p') & \text{if } p \notin dom(t_f) \\ t^\bullet(p) + \sum_{p'.t_f(p')=p} \boldsymbol{m}(p') & \text{if } p \in dom(t_f) \end{cases}$$

Intuitively, for places that are not sources of transfer arcs, besides the usual preset and postset modifications, there is also the possibility to add tokens transferred from corresponding source places of transfer arcs. For places which are sources of transfer arcs, only the new introduced tokens must be taken into account because the previously present tokens are consumed by the corresponding transfer arc. Also in this case, the effect of the firing of a transition is written $\boldsymbol{m} \mapsto \boldsymbol{n}$.

Typically, transfer arcs are depicted as arcs from places to places, connected to the corresponding transition by a line.

Example 8. In Figure 2 we depict a Petri net representing the behavior of the producer-consumer system defined in Example 7. The behavior of the producer process is reported on the left, while the consumer is on the right. Places in the middle of the figure represent the possible data in the data space (*prod*, *cons*, *job*, *todo*, *done* and *end*) and the trailing empty **0** process. Notice the transfer arc from the *job* to the *todo* place, used by the consumer to take under consideration all and only those job requests that have been already issued when the consumer starts.

We now discuss how to translate systems of the RenDS calculus into Petri nets with Transfer arcs. This translation is obtained as simple extension of the one in Definition 6. Sequential processes and the decomposition function is defined exactly as for the previous DS calculus. The unique difference is at the level of transitions: we simply add transitions to model the renaming of data from a to b,

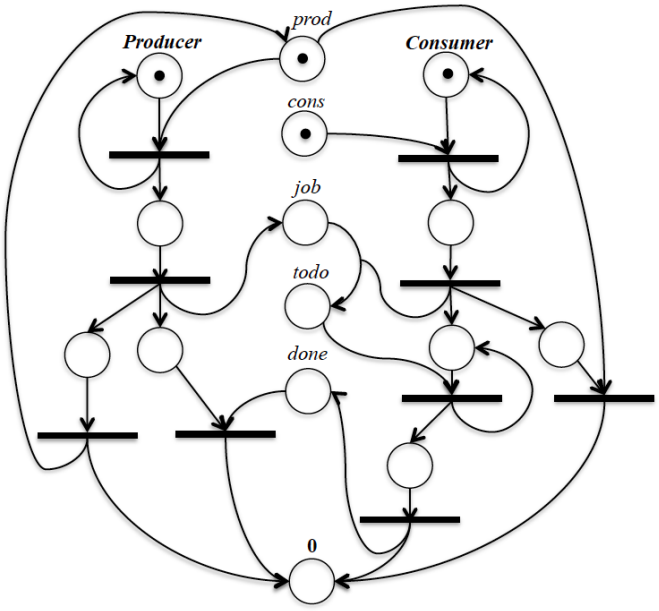

Fig. 2. Petri net for the producer-consumer example with renaming

with a transfer arc from the place a to the place b. All other actions are modeled as already done for the DS calculus. The new set of transitions, denoted with $\mathcal{T}_{ren}$, is defined in Table 4 plus the new rules in Table 5.

Definition 8. *Let $\langle P, \mathcal{S}\rangle$ be a system of the* RenDS *calculus. We define the Petri net with Transfer arcs $Net_{ren}(P, \mathcal{S}) = (P, T, \boldsymbol{m_0})$ as follows:*

- $S = \{Q \mid Q \text{ is a sequential process in } P\} \cup \{a \mid a \text{ occurs in } \mathcal{S} \text{ or in } P\}$
- $T = \{c \overset{t}{\rightarrowtail} p \in \mathcal{T}_{ren} \mid dom(c) \subseteq S\}$
- $\boldsymbol{m_0} = dec(P) \uplus \mathcal{S}$

Example 9. It is easy to see that the Transfer Petri net depicted in Figure 1 corresponds to $Net_{ren}(P, \{prod, cons\})$ where $\langle P, \{prod, cons\}\rangle$ is the system in Example 7.

We conclude by formalizing the correspondence between the operational semantics of the RenDS calculus and the corresponding Petri net with Transfer arcs. Also in this case we omit the proof of this correspondence result because standard.

Proposition 2. *Let $\langle P, \mathcal{S}\rangle$ be a system of the* RenDS *calculus, and $Net_{ren}(P, \mathcal{S}) = (P, T, \boldsymbol{m_0})$ be the corresponding Petri net with Transfer arcs. Let Q be a process composed of sequential processes occurring in P, and $\mathcal{V}$ be a multiset of*

$$\{\textstyle\sum_{i\in I}\alpha_i.P_i\} \overset{\{a\mapsto b\}}{\rightarrowtail} dec(P_j) \quad j\in I, \alpha_j = ren(a,b)$$

$$\{!ren(a,b).P\} \overset{\{a\mapsto b\}}{\rightarrowtail} dec(P) \uplus \{!ren(a,b).P\}$$

Table 5. Petri net transitions for renaming

data occurring in P or in $\mathcal{S}$. We have that $\langle Q, \mathcal{V}\rangle \rightarrow \langle Q', \mathcal{V}'\rangle$ if and only if $dec(Q) \uplus \mathcal{V} \mapsto dec(Q') \uplus \mathcal{V}'$ in $Net_{ren}(P, \mathcal{S})$.

4 Conclusion

The relationship between traditional concurrency models like process calculi and Petri nets has been one of those research topics, within the concurrency theory community, to which Pierpaolo Degano gave a fundamental initial contribution (see the seminal work [7]). A detailed description of the extremely vast literature concerning the relationship between process calculi and Petri nets is out of the scope of this paper. Here, we simply recall few relatively recent relevant papers.

In [1] a Petri net semantics is used to prove the decidability of termination (called convergence in that paper) in a version of CCS with replication instead of recursion, in which name restriction –usually called also name generation– cannot occur inside replication. This syntactic limitation guarantees that only boundedly many distinct names can be generated. In [14] a precise relationship between name passing calculi –in particular the π-calculus– and classical Petri nets has been established, by showing which are the precise restrictions to be imposed to name-generation and name-passing mechanisms in order to resort to a Petri net semantics. Open nets are instead used in [2] to equip with a net semantics an asynchronous version of CCS with replication and a limited form of restriction that cannot occur under the scope of a replication. Open nets are classical Petri nets including open places and the possibility for distinct nets to interact on open places. The advantage of Open nets is that they allow for a compositional definition of the net encoding. Moreover, they naturally support the modeling of restriction: free names (i.e. non restricted names) are modeled with open places while bound names (i.e. restricted names) are modeled with private places.

In this paper, we have focused on those cases in which in order to faithfully model a process calculus it is necessary to consider extended versions of Petri nets. This happens, for instance, when the process calculus includes global synchronization mechanisms. Classical Petri net transitions, in fact, always consume a predefined amount of tokens from the input places, thus the number of consumed tokens is independent from the current token distribution. Global synchronization mechanisms, on the contrary, are defined as functions depending on the current (global) state of the system.

In particular, we have formalized a simple data-centric calculus for which it is possible to define a faithful classical Petri net semantics, and then we extend it with a simple primitive that globally renames all the data of a given kind that are currently available. We prove that the addition of this global primitive strictly requires to move to an extended class of Petri nets (in this case we consider Transfer Petri nets) in order to define a faithful net modeling. Formally speaking, the encodings that we define between a process calculus and a Petri net have a one-to-one correspondence between reductions in the process calculus and transition firings in the Petri net. The impossibility to define an encoding, on the contrary, consider also weaker encodings in which only termination is preserved (i.e. a process terminates if and only if its encoding in the Petri net has a terminating computation).

Translating process calculi into Petri nets is useful because it allows for the application of Petri net analysis techniques, or decidability results, back to the initial process calculi. It is interesting to observe that there are cases in which also extended versions of Petri nets fail, like for instance in [3]. In that paper, a process calculus with replication and name generation is defined, for which it is possible to produce unboundedly many different active processes due to the dynamic generation of new names. The presence of unboundedly many different processes forbids the application of Petri nets; in fact, Petri nets only has a predefined finite amount of possibile places (and transitions). In that paper, the decidability of divergence was proved by resorting to Well Structured Transition Systems (WSTS) [11], a meta model which is more general than Petri nets (and their usual extensions) and for which a rich set of interesting properties like divergence, coverability, or those expressible by means of simple temporal logic, are proved to be decidable.

References

1. J. Aranda, F. D. Valencia, and C. Versari. On the expressive power of restriction and priorities in CCS with replication. In *Foundations of Software Science and Computational Structures, 12th International Conference, FOSSACS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5504 of *Lecture Notes in Computer Science*, pages 242–256. Springer, 2009.
2. P. Baldan, F. Bonchi, F. Gadducci, and G. V. Monreale. Modular encoding of synchronous and asynchronous interactions using open petri nets. *Sci. Comput. Program.*, 109:96124, 2015.
3. N. Busi, M. Gabbrielli, and G. Zavattaro. On the expressive power of recursion, replication and iteration in process calculi. *Mathematical Structures in Computer Science*, 19(6):1191–1222, 2009.
4. N. Busi, R. Gorrieri, and G. Zavattaro. On the expressiveness of linda coordination primitives. *Inf. Comput.*, 156(1-2):90–121, 2000.
5. N. Busi and G. Zavattaro. On the expressiveness of event notification in data-driven coordination languages. In *Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin,*

Germany, March 25 - April 2, 2000, Proceedings, volume 1782 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2000.
6. N. Busi and G. Zavattaro. Expired data collection in shared dataspaces. *Theor. Comput. Sci.*, 3(298):529–556, 2003.
7. P. Degano, R. De Nicola, and U. Montanari. A distributed operational semantics for CCS based on condition/event systems. *Acta Inf.*, 26(1/2):59–91, 1988.
8. G. Delzanno and G. Zavattaro. Reachability problems in bioambients. *Theor. Comput. Sci.*, 431:56–74, 2012.
9. C. Dufourd, A. Finkel, and P. Schnoebelen. Reset nets between decidability and undecidability. In *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings*, volume 1443 of *Lecture Notes in Computer Science*, pages 103–115. Springer, 1998.
10. J. Esparza and M. Nielsen. Decidability issues for petri nets - a survey. *Bulletin of the EATCS*, 52:244–262, 1994.
11. A. Finkel, J. Raskin, M. Samuelides, and L. V. Begin. Monotonic extensions of petri nets: Forward and backward search revisited. *Electr. Notes Theor. Comput. Sci.*, 68(6):85–106, 2002.
12. C. G. Petri nets with marking-dependent arc cardinality:Properties and analysis. In *ICATPN*, volume 815 of *LNCS*, page 179198. Springer, 1994.
13. E. W. Mayr. An algorithm for the general petri net reachability problem. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing, May 11-13, 1981, Milwaukee, Wisconsin, USA*, pages 238–246. ACM, 1981.
14. R. Meyer and R. Gorrieri. On the relationship between π-calculus and finite place/transition petri nets. In *CONCUR 2009 - Concurrency Theory, 20th International Conference, CONCUR 2009, Bologna, Italy, September 1-4, 2009. Proceedings*, volume 5710 of *Lecture Notes in Computer Science*, pages 463–480. Springer, 2009.
15. R. Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.
16. M. L. Minsky. *Computation: finite and infinite machines*. Prentice-Hall, Englewood Cliffs, 1967.
17. A. I. T. Rowstron and A. Wood. Solving the linda multiple rd problem using the copy-collect primitive. *Sci. Comput. Program.*, 31(2-3):335–358, 1998.
18. J. C. Shepherdson and J. E. Sturgis. Computability of recursive functions. *Journal of the ACM*, 10:217–255, 1963.