Scalable Computing: Practice and Experience
Volume 6, Number 4, pp. 71–81. http://www.scpe.org
ISSN 1895-1767

PETRI NETS AS EXECUTABLE SPECIFICATIONS OF HIGH-LEVEL TIMED PARALLEL SYSTEMS

FRANCK POMMEREAU*

Abstract. We propose to use *high-level Petri nets* as a model for the semantics of *high-level parallel systems*. This model is known to be useful for the purpose of verification and we show that it is also *executable* in a parallel way. Executing a Petri net is not difficult in general but more complicated in a *timed context*, which makes necessary to *synchronise* the *internal time* of the Petri net with the *real time* of its environment. Another problem is to relate the execution of a Petri net, which has its own semantics, to that of its environment; *i. e.*, to properly handle *input/output*.

This paper presents a parallel algorithm to execute Petri nets with time, enforcing the even progression of internal time with respect to that of the real time and allowing the exchange of information with the environment. We define a class of Petri nets suitable for a *parallel execution machine* which preserves the *step sequence semantics* of the nets and ensures time consistent executions while taking into account the solicitation of its environment. The question of the efficient verification of such nets has been addressed in a separate paper [14], the present one is more focused on the practical aspects involved in the execution of so modelled systems.

Key words. Petri nets, parallelism, real-time, execution machines.

1. Introduction. *Petri nets* are widely used as a model of *concurrency*, which allows to represent the occurrence of *independent* events. They can be as well a model of *parallelism*, where the *simultaneity* of the events is more important. Indeed, when we consider their *step sequence semantics*, an execution is represented by a sequence of *steps*, each of them being the simultaneous occurrences of some transitions. Within this semantics, the execution of a step may be replaced by that of any of its linearisation (total or partial). This can be viewed as possible executions of the same program on parallel machines with different numbers of processors. In this context, the choice of executing one step or another becomes a question of scheduling (this is usually solved non-deterministically by the Petri net semantics). Petri nets are thus suitable for *specifying* and *verifying* systems in models for which the portability is an important concern.

Our main goal in this paper is to show that Petri nets are also suitable for the *execution* of the modelled systems. We thus consider high-level Petri nets for modelling high-level parallel systems, with the aim to allow both verification and execution of the specification. The question of the efficient verification of such nets has been addressed in a separate paper [14], the present one is more focused on the practical aspects involved in the execution of so modelled systems.

There are at least two reasons for having executable specifications. First, it allows for prototyping and testing at early stage of the design: there may be no need to have an implementation in order to see how the program behaves when its model can already be executed. Second, if the execution of the specification can be made (or happens to be) efficient enough, there is no need to consider any further implementation. This completely saves from the risk of introducing errors on the way from specification to implementation: the verified model and the executed program are exactly the same object. It may be objected that Petri nets are suitable for modelling but really not for programming. This is true. However, Petri nets like those used in this paper are widely used has a semantical domain for parallel programming languages or process algebra with concurrent semantics. For instance, the semantics of the parallel language B(PN)^2 [3] is defined in terms of Petri nets similar to those used in this paper. It features most usually expected high-level constructs for programming languages, in particular: nested declaration of typed variables and FIFO communication channels; communication through shared variables or channels; atomic actions; control flow constructs including parallelism; procedures with parameters passed by value or by reference and allowing recursive and parallel calls [10]; exceptions whose propagation can carry arbitrary value [11]; or Ada-like tasking with suspend/resume or abort capability [12]. Moreover, it can be easily extended with real-time constructs using the same approach to timed system as presented in the following, see [13, § 7.3]. Another example is the *Causal Time Calculus* defined in [14] which is a process algebra with timing features having a step based semantics. Both these formalisms could be applied to massively parallel problems, allowing to leave Petri nets in the background while working with much more pleasant and convenient notations.

*LACL, université Paris 12 — 61, avenue du général de Gaulle — 94010 Créteil, France —pommereau@univ-paris12.fr

Executing a Petri net is not difficult when we consider it alone, *i. e.*, in a *closed world*. But as soon as the net is *embedded in an environment*, the question becomes more complicated. The first problem comes when the net is timed: we have to ensure that its time reference matches that of the environment. The second problem is to allow an exchange of information between the net and its environment. Both these questions are addressed in this paper.

The *causal time* approach is a way to introduce timing features in an otherwise untimed model [7], in particular Petri nets. The idea behind causal time is to *use the expressive power of the model* in order to give an *explicit representation of clocks* in the modelled systems. In the case of high-level Petri nets, it is possible to introduce *counters* and a distinguished *tick transition* whose role is to simultaneously increment them. These counters thus become the timing reference and can be used as clock-watches by the processes as in [15, 6, 13, 14]. It was shown in [6, 14] that the causal time approach is highly relevant since it is simple to put into practice and allows for *efficient verification* through model checking. This paper shows that this approach is also relevant when *concrete execution* are considered. For the purpose of verification, the hypothesis of the closed world is assumed: the Petri net which models a system is considered alone, without any reference to something external to it. The situation differs if we consider the execution of such a Petri net in an *environment* which has its *own time reference*. Indeed, the tick transition of a Petri net may causally depend on the progression of other transitions in the net, which results in the so called *deadline paradox* [7]: "tick is disabled until the system progresses". In a closed world, this statement is logically equivalent to "the system is forced to progress before the next tick", which solves the deadline paradox. But, in the case of an open world, one may wonder how even is the progression of the causal time with respect to that of the *real time*, which is the time imposed by the environment.

Moreover, if the Petri net has to communicate with its environment, one may ask how the net can receive information from the environment and send back appropriate responses. Producing output is rather simple since the net is not disturbed; but reading input (*i. e.*, changing the behaviour of the net in reaction to the changes in the environment) is more difficult and may not be always possible.

In this paper, we define a *parallel execution machine* whose role is to run a Petri net with a tick transition in such a way that the ticks occur evenly with respect to the real time. We show that this can be ensured under reasonable assumptions about the Petri net. The other role of the machine is to allow the communication between the Petri net and the environment and we will identify favourable situations, very easy to obtain in practice, in which the reaction to a message is ensured within a short delay. An important property of our execution machine will be that it will preserve the step sequence semantics of the Petri net: this machine can be seen as an implementation of the Petri net execution rule including additional constraints related to the environment (real time and communication).

In the perspective of direct execution of the modelled systems, it becomes natural to provide parallel executions of the model of a parallel system. So, our goal in proposing a parallel execution machine is more related to a question of consistency than to that of speedup. The question of the speed of our execution machine will thus be intentionally left out of the topics of this paper. However, our definitions will leave enough free space to investigate in this direction and we will come back to this discussion at the end of the paper.

1.1. Execution machines. Defining an execution machine is the usual way to show that an abstract model, defined under assumptions which may be considered as unrealistic, can be used for concrete executions. For instance, the family of synchronous languages (*e. g.*, Esterel [2]), relies on the *synchronous hypothesis* which states that the reaction to a signal is instantaneous. This leads to consider an infinitely fast computer in the abstract model. Several execution machines for these languages have been defined (see, *e. g.*, [1, 5]); in all cases, the solution to remove the synchronous hypothesis makes use of a compilation stage which produces finite automata in which a whole chain of action/reaction is collapsed on a single transition. This allows a correct implementation of the instantaneous reaction assuming a computer *fast enough* with respect to the delays that the environment can observe. However, this breaks the causality relation between events and leads to reject some systems which may be considered on the abstract level but are concretely impossible to implement.

Similar concerns arise in the case of Petri nets with causal time; in particular, we have to reject systems which allow runs of unbounded length between two consecutive ticks. (Such behaviours are often called *Zeno runs*.) Concerning the question of reacting to the solicitation of the environment, it is easy to introduce specific constructs in a Petri net in order to ensure that a signal will be always taken into account very efficiently,

provided that the environment is not "too demanding". This is to say that we will need a computer fast enough with respect to its environment, exactly like for synchronous languages.

1.2. Organisation of the paper. The sequel is organised as follows. The section 2 introduces the basic notions related to Petri nets and their semantics. The section 3 then defines the class of Petri nets we are interested in and gives the assumptions which must be considered in order to allow their real-time execution. The section 4 shows how such nets can be compiled into a form suitable for their execution. Then, the section 5 defines the execution machine itself. We finally conclude in the section 6, introducing discussions about the efficiency of an implementation.

2. Basic definitions about Petri nets. This section briefly introduces the class of Petri nets and the related notions that will be used in the following.

2.1. Multisets. A *multiset* over a set X is a function $\mu : X \to \mathbf{N}$. We denote by $\mathsf{mult}(X)$ the set of all finite multisets μ over X, *i. e.*, such that $\sum_{x\in X} \mu(x) < \infty$. We write $\mu \leq \mu'$ if the domain X of μ is included in that of μ', and if $\mu(x) \leq \mu'(x)$, for all $x \in X$. An element $x \in X$ belongs to μ, denoted $x \in \mu$, if $\mu(x) > 0$. The sum and difference of multisets, and the multiplication by a non-negative integer are respectively denoted by $+$, $-$ and $*$ (the difference is defined only when the second argument is smaller or equal to the first one). A subset of X may be treated as a multiset over X, by identifying it with its characteristic function, and a singleton set can be identified with its sole element. A finite multiset μ over X may be written as $\sum_{x\in X} \mu(x) * x$ or $\sum_{x\in X} \mu(x) * \{x\}$, as well as in extended set notation, *e. g.*, $\{a_1, a_1, a_2\}$ denotes a multiset μ such that $\mu(a_1) = 2$, $\mu(a_2) = 1$ and $\mu(x) = 0$ for all $x \in X \setminus \{a_1, a_2\}$.

2.2. Labelled Petri nets. Let $\mathbb{S}$ be a set of *actions symbols*, $\mathbb{D}$ a finite set of *data values* (or just *values*) and $\mathbb{V}$ a set of *variables*. For $A \subseteq \mathbb{S}$ and $X \subseteq \mathbb{D} \cup \mathbb{V}$, we denote by $A \otimes X$ the set $\{a(x) \mid a \in A, x \in X\}$. Then, we define $\mathbb{A} \stackrel{\text{df}}{=} \mathbb{S} \otimes (\mathbb{D} \cup \mathbb{V})$ as the set of *actions* (with parameters). These four sets are assumed pairwise disjoint.

Definition 2.1. *A* labelled marked Petri net *is a tuple* $N = (S, T, \ell, M)$ *where:*

- S *is a nonempty finite set of* places*;*
- T *is a nonempty finite set of* transitions*, disjoint from* S*;*
- ℓ *defines the* labelling *of places, transitions and* arcs*, i. e., elements of* $(S \times T) \cup (T \times S)$*, as follows:*
  - *for* $s \in S$*, the labelling is* $\ell(s) \subseteq \mathbb{D}$ *which defines the tokens that the place is allowed to carry (often called the* type *of* s*),*
  - *for* $t \in T$*, the labelling is* $\ell(t) \stackrel{\text{df}}{=} \alpha(t)\gamma(t)$ *where* $\alpha(t) \in \mathbb{A}$ *and* $\gamma(t)$ *is a boolean expression called the* guard of t*,*
  - *for* $(x, y) \in (S \times T) \cup (T \times S)$*, the labelling is* $\ell(x, y) \in \mathsf{mult}(\mathbb{D} \cup \mathbb{V})$ *which denotes the tokens flowing on the arc during the execution of the attached transition. The empty multiset* $\emptyset$ *denotes the absence of arc;*
- M *is a* marking *function which associates to each place* $s \in S$ *a multiset in* $\mathsf{mult}(\ell(s))$ *representing the tokens held by* s*.*

Notice that $\alpha(t)$ could be a finite multiset of actions. This would be a trivial extension but would lead to more complicated definitions; we choose to restrict ourselves to single actions in order to streamline the presentation.

We adopt the standard rules about representing Petri nets as directed graphs with the following simplifications: the names of some nodes (especially places) may not be given; the two components of transition labels are depicted separately; true guards are omitted as well as brackets around sets; arcs may be labelled by expressions as a shorthand (see the example given in the figure 2.1).

Fig. 2.1. *On the left, a Petri net which actually denotes that given on the right, with* $\eta \geq 0$*,* $\{0, \ldots, \eta\} \subseteq \mathbb{D}$*,* $\{x, y\} \subseteq \mathbb{V}$ *and* $\tau \in \mathbb{S}$*.*

2.3. Step sequence semantics. A *binding* is a function $\sigma : \mathbb{V} \to \mathbb{D}$ which associates concrete values to the variables appearing in a transition and its arcs. We denote by $\sigma(E)$ the evaluation of the expression E bound by σ.

Let (S, T, ℓ, M) be a Petri net, and $t \in T$ one of its transitions. A binding σ is *enabling* for t at M if the guard evaluates to true, *i. e.*, $\sigma(\gamma(t)) = \top$, and if the evaluation of the annotations on the adjacent arcs respects the types of the places, *i. e.*, for all $s \in S$, $\sigma(\ell(s,t)) \in \mathsf{mult}(\ell(s))$ and $\sigma(\ell(t,s)) \in \mathsf{mult}(\ell(s))$.

A *step* corresponds to the simultaneous execution of some transitions, it is a multiset

$$U = \{(t_1, \sigma_1), \ldots, (t_k, \sigma_k)\}$$

such that $t_i \in T$ and σ_i is an enabling binding of t_i, for $1 \leq i \leq k$. U is *enabled* if the marking is sufficient to allow the flow of tokens required by the execution of the step, *i. e.*, for all $s \in S$

$$M(s) \;\geq \sum_{(t,\sigma)\in U} U((t,\sigma)) * \sigma(\ell(s,t)).$$

It is worth noting that if a step U is enabled at a marking, then so is any sub-step $U' \leq U$. A step U enabled by M may be *executed*, leading to the new marking M' defined for all $s \in S$ by

$$M'(s) \stackrel{\text{df}}{=} M(s) \;- \sum_{(t,\sigma)\in U} U((t,\sigma)) * \sigma(\ell(s,t)) \;+ \sum_{(t,\sigma)\in U} U((t,\sigma)) * \sigma(\ell(t,s)).$$

This is denoted by $M[U\rangle M'$ and this notation naturally extends to sequences of steps. The empty step, denoted by $\emptyset$, is always enabled and we have $M[\emptyset\rangle M$. A marking M' is *reachable* from a marking M if their exists a sequence of steps ω such that $M[\omega\rangle M'$; we will say in this case that M enables ω. Notice that M is reachable from itself through a sequence of empty steps.

The *step sequence semantics* is defined as the set containing all the sequences of steps enabled by a net. This semantics is based on transitions identities but the relevant information is generally the labels of the executed transitions. The *labelled step* associated to a step U is defined as $\sum_{(t,\sigma)\in U} U((t,\sigma)) * \sigma(\alpha(t))$, which allows to naturally define the *labelled step sequence semantics* of a Petri net. In the sequel we will consider only this semantics and omit the word "labelled".

2.4. Safety. A Petri net (S, T, ℓ, M) is *safe* if any marking M' reachable from M is such that, for all $s \in S$ and all $d \in \ell(s)$, $M'(s)(d) \leq 1$, *i. e.*, any place holds at most one token of each value. The class of safe Petri nets (including models abbreviating them) is very interesting:

- from a theoretical point of view, safe Petri nets never have auto-concurrency of transitions, which allows for efficient verification techniques [8];
- from a pragmatical point of view, safe Petri nets corresponds to the class of finite state Petri nets (as shown in [4], bounded Petri nets can be reduced to safe Petri nets while preserving their concurrent semantics), which correspond to realistic systems, *i. e.*, those that can be implemented on a concrete computer;
- from a practical point of view, this class was shown expressive enough to model most interesting problems from the real world. For instance, the semantics procedures, exceptions or tasks preemption in the language B(PN)² do not require more than safe Petri net.

Another nice property of safe Petri nets, directly related to our purpose, is that they have finitely many reachable markings, each of which enabling finitely many steps whose sizes are bounded by the number of transitions in the net. For all these reasons, as in the previous works about causal time [15, 6, 13, 14], we restrict ourselves to safe Petri nets.

3. Petri nets with causal time: CT-nets. We are now in position to define the class of Petri nets we are actually interested in; it consists in safe Petri nets, with several restrictions, for which we will define some specific vocabulary related to the occurrence of ticks. We assume that there exists $\tau \in \mathbb{S}$, used in the labelling of the tick transition.

Definition 3.1. *A* Petri net with causal time *(*CT-net*) is a safe labelled Petri net* $N \stackrel{\text{df}}{=} (S, T, \ell, M)$ *in which there exists a unique* $t_\tau \in T$*, called the* tick transition *of* N*, such that:*

- $\alpha(t_\tau) \in \{\tau\} \otimes (\mathbb{D} \cup \mathbb{V})$*;*

- $\alpha(t) \notin \{\tau\} \otimes (\mathbb{D} \cup \mathbb{V})$ *for all* $t \in T \setminus \{t_\tau\}$;
- t_τ *has at least one incoming arc labelled by a singleton.*

A tick-step *is a step* U *of* N *which involves the tick transition, i. e., such that* $\tau(d) \in U$ *for a* $d \in \mathbb{D}$.

Thanks to the safety and the last restriction on t_τ, any tick-step contains exactly one occurrence of the tick transition. On the other hand, one may notice that this definition is very liberal and allows to define nets in which the tick transition is not tight to increment counters but may produce any other effect not related to time. Fortunately, we do not need a more restrictive definition, which lets us free to experiment different approaches in the future.

The figure 3.1 shows a toy CT-net that will be used as a running example. In this net, the role of the tick transition t_τ is to increment a counter located in the top-right place. When the transition t_1 is executed, it resets this counter and picks in the top-left place a value which is bound to the variable m. This value is transmitted to the transition t_2 which will be allowed to execute when at least m ticks will have occurred. Thus, m specifies the minimum number of ticks between the execution of t_1 and that of t_2. At any time, the transition t_3 may randomly change the value of this minimum while emitting a visible action $u(x)$ where x is the new value. Notice that the maximum number of ticks between the execution of t_1 and that of t_2 is enforced by the type of the place connected to t_τ which specifies that only tokens in $\{0, \ldots, \eta\}$ are allowed (given $\eta > 0$).

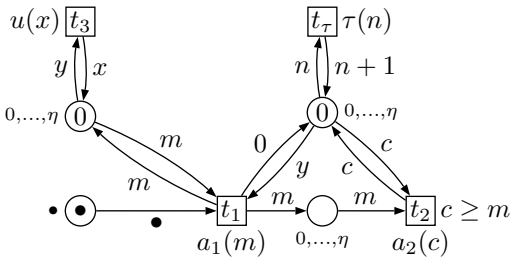

FIG. 3.1. *An example of a CT-net, where* $\eta > 0$, $\{a_1, a_2, u, \tau\} \subseteq \mathbb{S}$, $\{c, n, m, x, y\} \subseteq \mathbb{V}$ *and* $\{0, \ldots, \eta\} \cup \{\bullet\} \subseteq \mathbb{D}$.

Assuming $\eta \geq 5$, a possible execution of this CT-net is:

$$\{\tau(0)\}\ \{u(2)\}\ \{a_1(2)\}\ \{\tau(0), u(1)\}\ \{\tau(1)\}\ \{u(5)\}\ \{\tau(2)\}\ \{\tau(3)\}\ \{a_2(4), u(0)\}\ \{\tau(4)\}\ .$$

3.1. Tractability. A CT-net (S, T, ℓ, M) is *tractable* if there exists an integer $\delta \geq 2$ such that, for all marking M' reachable from M, any sequence of at least δ nonempty steps enabled by M' contains at least two tick-steps. In other words, the length of an execution between two consecutive ticks is bounded by δ whose smallest possible value is called the *maximal distance between ticks*.

This notion of tractable nets is important because it allows to distinguish those nets which can be executed on a realistic machine: indeed, an intractable net may have potentially infinite runs between two ticks (so called *Zeno runs*), which cannot be executed on a finitely fast computer without breaking the evenness of ticks occurrences.

For example, the CT-net of our running example is intractable because the transition t_3 can be executed infinitely often between two ticks: in the execution given above, the step $\{u(5)\}$ could be repeated an arbitrary number of times. In the rest of this paper, we restrict ourselves to tractable CT-nets.

3.2. Input and output. The communication between a CT-net and its environment is modelled using some of the actions in transitions labels. We distinguish for this purpose two finite disjoint subsets of $\mathbb{S}$: $\mathbb{S}_i$ is the set of *input action symbols* and $\mathbb{S}_o$ is that of *output actions symbols*. We assume that $\tau \notin \mathbb{S}_i \cup \mathbb{S}_o$. We also distinguish a nonempty set $\mathbb{D}_{io} \subseteq \mathbb{D}$ representing the values allowed for input and output. Intuitively, the distinguished symbols correspond to communication ports on which values from $\mathbb{D}_{io}$ may be exchanged between the execution machine and its environment. Thus the execution of a transition labelled by $a_o(d_o) \in \mathbb{S}_o \otimes \mathbb{D}_{io}$ is seen as the sending of the value d_o on the output port a_o. Conversely, if the environment sends a value $d_i \in \mathbb{D}_{io}$ on the input port $a_i \in \mathbb{S}_i$, the net is expected to execute a step containing the action $a_i(d_i)$. In general, we cannot ensure that such a step is enabled, in the worst case, it may happen that no transition has a_i in its label. Fortunately, we show now that a net can easily be designed in order to ensure that such an input message is always correctly handled.

A naive way to achieve this result is to use self-loops, like the transition t_3 in the figure 3.1. In this example, if we assume $u \in \mathbb{S}_i$ and $\{0, \ldots, \eta\} \supseteq \mathbb{D}_{io}$, any requested communication on u can always be handled. Unfortunately, self-loops lead to intractable nets since such transitions can always be arbitrarily repeated (remember the step $\{u(5)\}$ above). Actually, a self-loop indicates that the CT-net is expected to be able to respond instantaneously to all the messages that the environment would send on the corresponding port, which is not a realistic assumption. Indeed, if the number of such messages sent in a given amount of real time is not bounded, then a finitely fast computer cannot avoid to miss some of them. So, in the following, we assume that the environment may not produce more than one message on each input port between two ticks, which will lead to the notion of tick-reactiveness. This assumption is equivalent to say that we require the CT-net to be executed on a computer fast enough with respect to its environment; so, this is actually one of the classical conditions that must be assumed while defining an execution machine.

Let $A \subseteq \mathbb{S}_i$ be a nonempty set of input action symbols, we denote by $\mathsf{req}(A)$ the set of potential requests on A, which contains all the sets of the form $\{a_1(d_1), \ldots, a_k(d_k)\}$ where $\{a_1, \ldots, a_k\} \subseteq A$ and $(d_1, \ldots, d_k) \in {\mathbb{D}_{io}}^k$ for all $k \geq 1$. Each element of $\mathsf{req}(A)$ is potentially a step of a CT-net.

A CT-net (S, T, ℓ, M) is *once-reactive* to $A \subseteq \mathbb{S}_i$ iff: either, it enables only the empty step; or, there exists a step $U' \notin \mathsf{req}(A)$ such that $M[U'\rangle M''$ and, for all $U \in \mathsf{req}(A)$, we have $M[U\rangle M'$ and the CT-net (S, T, ℓ, M') is once-reactive to $A \setminus \{a \in A \mid \exists d \in \mathbb{D}_{io}, a(d) \in U\}$. Intuitively, this inductive definition states that, for all input port $a_i \in \mathbb{S}_i$, the CT-net can react to any request on a as soon as it comes, after what it may miss them. On the other hand, the CT-net is never forced to execute an action involving an input port in A (thanks to the step U'). At any time, the CT-net may terminate its execution with a deadlock.

A CT-net (S, T, ℓ, M) is *tick-reactive* to $A \subseteq \mathbb{S}_i$ iff it is once-reactive to A and, for all sequence of steps $U_1 \cdots U_k$ such that U_k is a tick-step and $M[U_1 \cdots U_k\rangle M'$, then the CT-net (S, T, ℓ, M') is tick-reactive to A. This definition is also inductive and states that a tick-reactive CT-net is almost like a once-reactive net except that its capability to react is fully restored after each tick. This guarantees that one message on a may always be handled between two ticks, which exactly matches our assumption. It turns out that it is easy to transform a reactive CT-net with self-loops into a tick-reactive one. It is enough to add one place for each self-loop with the type $\{\circ, \bullet\}$ and marked with $\bullet$, and arcs such that each occurrence of the self-loop consumes the $\bullet$ and replace it with a $\circ$, so it cannot occur twice; on the other hand, each occurrence of the tick-transition must reset to $\bullet$ the token in the added places. This way, self-loops cannot be repeated with at least one tick in between. As we can see, it is easy to construct a tick-reactive net; for instance, the figure 3.2 shows a modified version of our running example which is tick-reactive to $\{u\}$ and tractable (now, the step $\{u(5)\}$ could not be repeated at will).

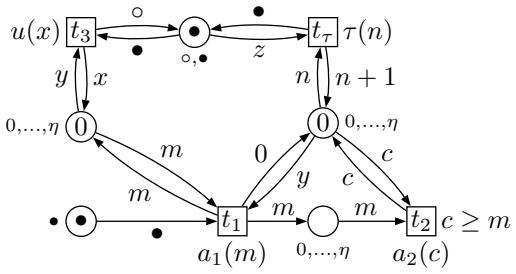

FIG. 3.2. *The tick-reactive version of the running example, where* $z \in \mathbb{V}$ *and* $\{\circ, \bullet\} \subset \mathbb{D}$.

3.3. Consistency. We denote by $U[a]$ the number of occurrences of the action symbol a in a step U, *i. e.*, $U[a] \stackrel{\text{df}}{=} \sum_{a(x) \in U} U(a(x))$. A step U is *consistent* if $U[a] \leq 1$ for all $a \in \mathbb{S}_i \cup \mathbb{S}_o$. A CT-net is *consistent* if its step sequence semantics only involve consistent steps. Inconsistent steps are those during the execution of which several communications take place on the same port. Since the transitions executed by a single step occur simultaneously, this means that several values may be sent or received on the same port at the same time. This is certainly something which is not realistic and so, we restrict ourselves to consistent CT-nets in the following.

The nets given in the figures 3.1 and 3.2 are both consistent. But, assuming $a_2 \in \mathbb{S}_i \cup \mathbb{S}_o$, it would not be the case if we would replace $u(x)$ by $a_2(x)$ in the label of the transition t_3 since we could have and execution with the step $\{a_2(4), a_2(0)\}$ which is not consistent.

4. Compilation of CT-nets: CT-automata. The aim of this section is to show how to transform a tractable and consistent CT-net into a form more suitable for the execution machine. This corresponds to a compilation producing an automaton (non-deterministic in general), called a *CT-automaton*, whose states are the reachable markings of the net and whose transitions correspond to the steps allowing to reach one marking from another. It should be remarked that this compilation is not strictly required but allows to simplify things a lot, in particular in an implementation of the machine: with respect to its corresponding CT-net, a CT-automaton has no notion of markings, bindings, enabling, etc., which results in a much simpler model. Another reason to introduce this compilation stage is that it can be used to check if the net of interest is really a safe, tractable and consistent CT-net; moreover, it is an almost necessary step to compute the value of δ (the maximal distance between ticks) which will be used during the execution. So, as we cannot avoid a computation at least equivalent to this compilation stage, we turn it into an advantage for the execution which can be made much simpler and more efficient.

In order to record only the input and output actions in a step U of a CT-net, we define the set of the *visible actions in* U by $\lfloor U \rfloor \stackrel{\text{df}}{=} U \cap (((\mathbb{S}_i \cup \mathbb{S}_o) \otimes \mathbb{D}_{io}) \cup (\{\tau\} \otimes \mathbb{D}))$. Because of the consistency, $\lfloor U \rfloor$ could not be a multiset.

DEFINITION 4.1. *Let $N = (S, T, \ell, M)$ be a tractable and consistent CT-net, the* CT-automaton of N *is the finite automaton $\mathcal{A}(N) \stackrel{\text{df}}{=} (S_{\mathcal{A}}, T_{\mathcal{A}}, s_{\mathcal{A}})$ where:*

- *$S_{\mathcal{A}}$ is the set of states defined as the set of all the reachable markings of N;*
- *the set of transitions is $T_{\mathcal{A}} \subseteq S_{\mathcal{A}} \times L_{\mathcal{A}} \times S_{\mathcal{A}}$, where $L_{\mathcal{A}} \stackrel{\text{df}}{=} \{A \subseteq ((\mathbb{S}_i \cup \mathbb{S}_o) \otimes \mathbb{D}_{io}) \cup (\{\tau\} \otimes \mathbb{D})\}$, and is defined as the set of all the triples (M', A, M'') such that $M', M'' \in S_{\mathcal{A}}$ and there exists a nonempty step U of N such that $M[U\rangle M'$ and $A = \lfloor U \rfloor$;*
- *$s_{\mathcal{A}} \stackrel{\text{df}}{=} M \in S_{\mathcal{A}}$ is the initial state of $\mathcal{A}(N)$, i. e., the initial marking of N.*

The following holds by definition but should be stressed since it states that a CT-net and the corresponding CT-automaton have exactly the same executions.

PROPOSITION 4.2. *Let $N \stackrel{\text{df}}{=} (S, T, \ell, M)$ be a tractable and consistent CT-net, M' be a reachable marking of N and $(S_{\mathcal{A}}, T_{\mathcal{A}}, M) \stackrel{\text{df}}{=} \mathcal{A}(N)$.*

1. *If $M'[U\rangle M''$ for a nonempty step U then $(M', \lfloor U \rfloor, M'') \in T_{\mathcal{A}}$.*
2. *Conversely, if $(M'', A, M''') \in T_{\mathcal{A}}$ then there exists a nonempty step U such that $M''[U\rangle M'''$ and $\lfloor U \rfloor = A$.*

As an example, the figure 4.1 shows the CT-automaton which corresponds to the tractable version of our running example (given in the figure 3.2). For the sake of compactness, we assumed $\eta \stackrel{\text{df}}{=} 1$ (the automaton for $\eta = 2$ has 105 states and this number grows to 277 for $\eta = 3$). Moreover, we assumed $\{a_1, a_2, u\} \subseteq \mathbb{S}_i \cup \mathbb{S}_o$.

5. The execution machine. We now describe the execution machine. In order to communicate with the environment, a symbol $a_o \in \mathbb{S}_o$ is considered as a port on which a value $d \in \mathbb{D}_{io}$ may be written, which is denoted by $a \leftarrow d$ (more generally, this is used for any assignment). Similarly, a symbol $a_i \in \mathbb{S}_i$ is considered as a port on which such a value, denoted by $a_i?$, may be read; we assume that $a_i? = \varnothing \notin \mathbb{D}_{io}$ when no communication is requested on a_i. Moreover, in order to indicate to the environment if a communication have been properly handled, we also assume that each $a \in \mathbb{S}_i$ may be marked "accepted" (denoting that the communication has been correctly handled), "refused" (denoting that the communication could not been handled), "erroneous" (denoting that a communication on this port was possible but with another value, or that a communication was expected but not requested) or not marked, which is represented by "no mark". We also use the notation $a_i \leftarrow \mathit{mark}$ when an input port is being marked.

Let $(S_{\mathcal{A}}, T_{\mathcal{A}}, s_{\mathcal{A}})$ be a CT-automaton and let Δ be a constant amount of time; we will see later on how Δ is defined since it depends on the definition of the execution machine. We will use three variables:

- Θ is a time corresponding to the occurrences of ticks;
- $s \in S_{\mathcal{A}}$ is the current state;
- $I \subseteq \mathbb{S}_i$ is the set of ports on which the environment asks a communication.

The behaviour of the machine is described by the algorithm given on the left of the figure 4.2 where the execution of a step (line 13) is detailed on the right of the figure. Several aspects of this algorithm should be commented:

- the statement "**now**" evaluates to the current time when it is executed;
- the "**for all**" loops are parallel loops;
- the execution of the line 8 can be parallelised also (see below);

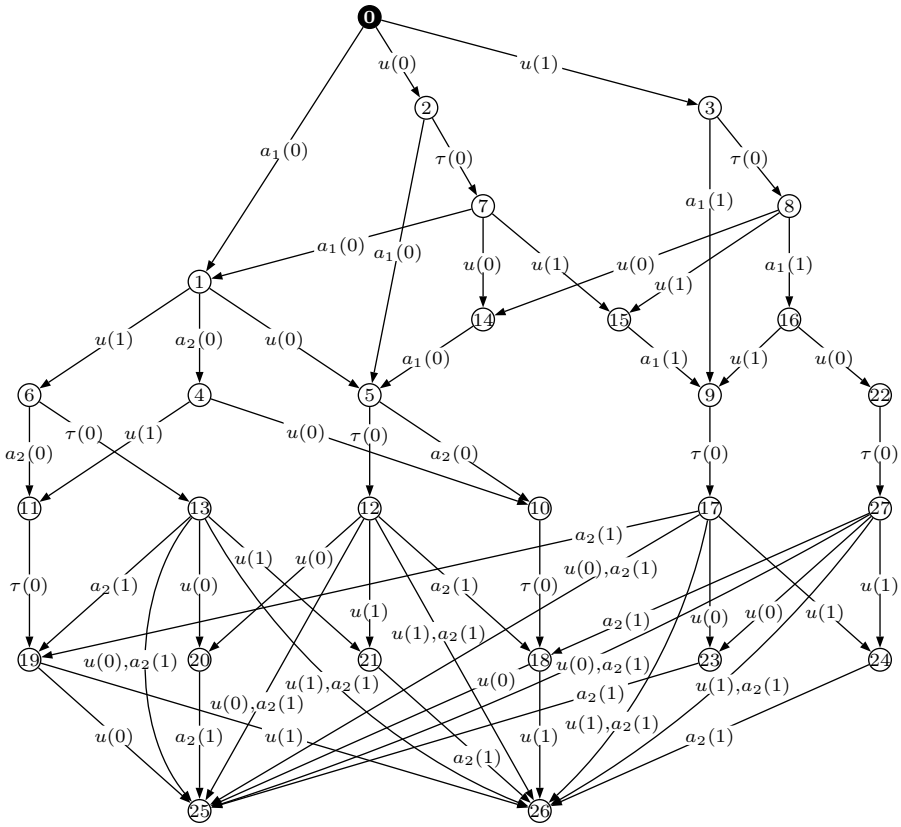

FIG. 4.1. *The CT-automaton of the CT-net given in the figure 3.2 (with* $\eta \stackrel{\text{df}}{=} 1$*), the initial state is numbered 0 and filled in black.*

```
1:  s ← s_A
2:  Θ ← now
3:  while s has successors do
4:     for all a ∈ 𝕊_i do
5:        a ← "no mark"
6:     end for
7:     I ← {a ∈ 𝕊_i | a? ≠ ∅}
8:     choose a transition (s, A, s')
9:     if A is a tick-step then
10:       wait until now = Θ + Δ
11:       Θ ← now
12:    end if
13:    execute(A, I)
14:    s ← s'
15: end while
```

```
procedure execute(A, I) :
17: for all a(d) ∈ A (a ≠ τ) do
18:    if a ∈ 𝕊_o then
19:       a ← d
20:    else if a ∈ 𝕊_i and a? = d then
21:       a ← "accepted"
22:    else
23:       a ← "erroneous"
24:    end if
25:    I ← I \ {a}
26: end for
27: for all a ∈ I do
28:    a ← "refused"
29: end for
```

FIG. 4.2. *The main loop of the execution machine (on the left) and the execution of a step A with respect to requested inputs given by I (on the right).*

- each execution of the "**while**" loop performs a bounded amount of work, in particular the following numbers are bounded: the number of ports; the number of transitions outgoing from a state; the number of actions in each step. Assuming that choosing a transition requires a fixed amount of time (see below), Δ is the maximum amount of time required to execute the "**while**" loop $\delta - 1$ times;
- no tick is explicitly executed but its occurrence actually corresponds to the execution of the line 11.

PROPOSITION 5.1. *The algorithm presented in the figure 4.2 ensures an even occurrence of the ticks.*

Proof. Let θ be the value assigned to Θ when the line 2 is executed. A number of transitions (at most $\delta - 2$) is executed until a tick transition is chosen. All together, the duration of these executions requires is $D \leq \Delta$ so the line 10 waits during $\Delta - D$. Thus, the line 11, which corresponds to the tick, is executed at time $\theta + D + (\Delta - D) = \theta + \Delta$. By induction, we obtain that ticks are executed at times $\theta + k\Delta$ for $k \geq 1$. □

5.1. Choosing a transition. We still have to define how a transition may be chosen, in a fixed amount of time, in order to mark "accepted" as much as possible input ports in the set I of requested communications. In order to define a criterion of maximality, we assume that there exists a total order on $\mathbb{S}_i$. This corresponds to a priority between the ports: when several communications are requested but not all are possible, we first choose to serve those on the ports with the highest priorities. Then, given I, we define a partial order $\prec$ on the transitions outgoing from a state and the machine chooses one of the smallest transitions according to $\prec$. This choice may be random or driven by a scheduler. For instance, we may choose to execute steps as large as possible, or steps no larger than the number of processors, etc. The definition of a scheduling strategy is out of the scope of this paper; we just need to assume that the time needed to choose a transition is bounded (which should hold in the reasonable cases).

For each step A appearing on a transition outgoing from the current state, we define a vector $V_A \in \{0,1,2\}^{\mathbb{S}_i}$ which represents the marks on the input ports after A would be executed: the value 0 stands for "accepted" or "no mark", the value 1 for "refused" and the value 2 for "erroneous". Thus, the value of $V_A(a)$ can be found using the following table, where d and d' are distinct values in $\mathbb{D}_{io}$:

	$A[a] = 0$	$a(d) \in A$	$a(d') \in A$
$a? = d$	1	0	2
$a? = \varnothing$	0	2	2

Then, $A_1 \prec A_2$ if $V_{A_1} < V_{A_2}$ according to the lexicographic order on these vectors.

Again, it is clear that building these vectors and choosing the smallest one is feasible in a fixed amount of time since the number of transitions outgoing from a state is bounded. This is also feasible in parallel: all the V_A's can be computed in parallel (as well as all their components) and the selection of the smallest one is a logarithmic reduction.

Notice that if $\prec$ allows to define a total order on steps, it is not the case for the transitions since several transitions may be labelled by the same step. For instance, assuming $u \notin \mathbb{S}_i \cup \mathbb{S}_o$, the running example would give a CT-automaton similar to that of the figure 4.1 but in which all the actions $u(0)$ or $u(1)$ would have been deleted. In this case, the state 13 would have two outgoing transitions labelled by $\emptyset$ and three labelled by $a_2(1)$.

PROPOSITION 5.2. *Let $a \in \mathbb{S}_i$ be an input action symbol and N be a CT-net which is tick-reactive to $R \ni a$. Then, the execution of $\mathcal{A}(N)$ will never mark a as "erroneous" nor "refused".*

Proof. Let s be the current state of $\mathcal{A}(N)$ and (s, A, s') be the transition chosen by the execution machine. There are three cases.

(1) If $a? = \varnothing$, then it may be marked "erroneous" or not marked. In the former case, this means that $a(d) \in A$ for a $d \in \mathbb{D}_{io}$. Then, if $A = \{a(d)\}$, because of the tick-reactiveness, there must exist a transition (s, U', s'') which does not involve a (tick-reactiveness never forces the occurrence of an input action), otherwise, the transition (s, A', s'') with $A' \stackrel{\text{df}}{=} A \setminus \{a(d)\}$ must exists (since it corresponds to a sub-step). In both cases, we have $(s, U', s'') \prec (s, A, s')$ or $(s, A', s'') \prec (s, A, s')$ hence a contradiction with the fact that (s, A, s') was chosen. So, a must be not marked in this case.

(2) If $a? = d \neq \varnothing$ and the communication on a is marked "refused", this means that $A[a] = 0$. The tick-reactiveness ensures that there must exist a transition $(s, A \cup \{a(d)\}, s'')$ (by assumption, a cannot have been requested before since the previous tick), hence again a contradiction. So, a must be marked "accepted" in this case.

(3) If $a? = d \neq \varnothing$ and the communication on a is marked "erroneous", this means that $a(d') \in A$ for a $d' \in \mathbb{D}_{io} \setminus \{d\}$. But there must exist a transition $(s, (A \cup \{a(d)\}) \setminus \{a(d')\}, s'')$ (tick-reactiveness allows the occurrence for any value in $\mathbb{D}_{io}$), hence again a contradiction. So, a is also marked "accepted" here. □

Then, the next result shows that a communication requested on a port to which the CT-net is tick-reactive is always correctly handled (*i. e.*, accepted) within the current "**while**" loop, which is the best response time that one can expect from the presented algorithm.

PROPOSITION 5.3. *Let $a \in \mathbb{S}_i$ be an input action symbol and N be a CT-net which is tick-reactive to $R \ni a$. If $a? = d \neq \varnothing$ before the execution of the line 7 in the figure 4.2, then a is marked* "accepted" *after the line 13 has executed.*

Proof. Directly follows from how the machine chooses a transition and from the proposition 5.2. □

6. Concluding remarks. We defined a parallel execution machine which shows the adequacy of causal and real time by allowing time-consistent executions of causally timed Petri nets (CT-nets) in a real-time environment. We also shown that it was possible to ensure that the machine efficiently reacts to the solicitation of its environment by designing CT-nets having the property of tick-reactiveness, which is easy to construct. In order to obtain these results, several restrictions have been adopted:

- only *safe* Petri nets are considered;
- the nets must be *tractable*, *i. e.*, they are not allowed to have unbounded runs between two ticks;
- the nets must be *consistent*, *i. e.*, they cannot perform several simultaneous communications on the same port;
- the execution machine must be run on a computer *fast enough* to ensure that the environment cannot attempt more than one communication on a given port between two ticks.

We do not consider the tractability and consistency requirements as true restrictions since they actually correspond to what can be performed on a realistic machine. The last restriction is actually a prescription: in order to ensure a correct communication, one has to run the execution machine on a computer fast enough to execute ticks more often than the environment can produce input. Moreover, it should be noticed that the frequency of ticks is arbitrary. So, if the ticks of a CT-net are too much sparse with respect to the requested inputs, it is easy to multiply by a constant k all its timing constraints in the net so ticks will occur k times more often. Using non-safe Petri nets may be considered in the future, however, this would lead to the class of infinite state systems which does not seem realistic for the purpose of execution.

6.1. Future work. Petri nets like CT-nets have been used for a long time as a semantical domain for high-level programming languages and process algebras with step based semantics (see, *e. g.*, [3, 14]) and these techniques could be directly applied to massively parallel languages or formalisms. In this direction, we envisage to combine a n-ary parallel composition operation with symmetry reductions [9] allowing to the verification of very large systems while giving modelling support for kinds of SPMD systems.

6.2. Implementation issues. A preliminary version of this work proposed a sequential execution machine and a prototype has been successfully implemented in Ada; this allowed to show that the evenness of ticks was not only possible in the theory but also easy to achieve in an implementation. (The only "difficulty" was to obtains Δ using test runs at the starting of the machine.) A parallel implementation of the version presented here had been started but had to be delayed since it turned out that there were still need for a ground study. Indeed, several open questions are actually critical ones. Notice that if our goal is to perform testing or simulation, an implementation can be naive and may even be sequential. But in the perspective of direct execution of the modelled systems, the speedup becomes crucial and actually depends on the interaction between several parameters: the model of computation, the family of parallel machine targeted and the scheduling strategy (as discussed in the section 5.1). All these questions were left out of the current paper; we thus envisage further research on this subject with the goal to identify good combinations allowing to produce high-quality implementations of our execution machine. In particular: how to exploit the parallelism in the presented algorithm strongly depends on the computational model envisaged (which may itself depend on the target architecture); the question of storing the CT-automaton is also important if one targets a distributed memory architecture. Taking all these parameters into account may lead to several very different refinements of the algorithm proposed above, each specially dedicated to a particular class of parallel computer and parallel programming language or model.

Related to the goal of efficient executions, another interesting problem is to connect the input/output of the machine to the concrete computer in order to delegate some computation. Indeed, output actions may be considered as calls to computational primitives, while input actions could correspond to the receiving of the computed values. This introduces delays, externals to the model, which must be taken into account. This can be made by introducing further timing constraints in the model in order to reflect the execution times obtained from benchmarks or from real-time guarantees in the case of calls to real-time primitives. In this perspective, considering Petri nets with time becomes necessary.

6.3. Conclusion. We believe that the framework proposed in this paper can be used to build concrete parallel applications in which the control flow could be ensured by Petri nets while a large part of the computation would be delegated to dedicated primitives with known performances. Using Petri nets for both the modelling and the execution allows to verify and run the same object, saving from the risk to introduce errors on the way from a model to its implementation, while allowing executions even during the early stages of the design.

REFERENCES

[1] C. André F. Boulanger, A. Girault, *Software Implementation of Synchronous Programs*, ICACSD'2001, IEEE Computer Society, 2001.
[2] G. Berry, *The foundations of Esterel*, Language and Interaction: Essays in Honour of Robin Milner. MIT Press, 1998.
[3] E. Best and R. P. Hopkins, *B(PN)^2 — A basic Petri net programming notation.* PARLE'93. LNCS 694, Springer, 1993.
[4] E. Best and H. Wimmel, *Reducing k-safe Petri nets to pomset-equivalent 1-safe Petri nets*, ICATPN'00. LNCS 1825, Springer, 2000.
[5] E. Boufaïd, *Machines d'ex'ecution pour langages synchrones*, PhD Thesis, University of Nice-Sophia Antipolis, 1998.
[6] C. Bui Thanh, H. Klaudel and F. Pommereau, *Petri nets with causal time for system verification*, MTCS'02. ENTCS, Elsevier, 2002.
[7] R. Durchholz, *Causality, time, and deadlines*, Data & Knowledge Engineering, 6. North-Holland, 1991.
[8] J. Esparza, *Model checking using net unfoldings*, Science of Computer Programming, Elsevier, 1994.
[9] T. Junttila, *On the Symmetry Reduction Method for Petri Nets and Similar Formalisms*, PhD Thesis, Helsinki University of Technology, 2003
[10] H. Klaudel, *Compositional High-Level Petri nets Semantics of a Parallel Programming Language with Procedures*, Sciences of Computer Programming 41, Elsevier, 2001.
[11] H. Klaudel and F. Pommereau, *A concurrent semantics of static exceptions in a parallel programming language*, ICATPN'01. LNCS 2075, Springer, 2001.
[12] H. Klaudel and F. Pommereau, *A class of composable and preemptible high-level Petri nets with an application to multi-tasking systems*, Fundamenta Informaticae, 50(1):33–55. IOS Press, 2002.
[13] F. Pommereau, *Modèles composables et concurrents pour le temps-re'el*, PhD. Thesis, University Paris 12, France, 2002.
[14] F. Pommereau, *Causal Time Calculus*, FORMATS'03. LNCS 2791, Springer, 2004.
[15] G. Richter, *Counting interfaces for discrete time modeling*, Technical report 26, GMD. September 1998.

Edited by: Frédéric Loulergue
Received: June 8, 2004
Accepted: June 9, 2005