Petri nets as Executable Specifications of High-Level Timed Parallel Systems

Franck Pommereau

Petri nets as Executable Specifications of High-Level Timed Parallel Systems

Franck Pommereau

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

Abstract. We propose to use *high-level Petri nets* for the semantics of *high-level parallel systems*. This model is known to be useful for the of verification and we show that it is also executable in a parallel way. Executing a Petri net is easy 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 the internal time with respect to 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 [4], the present one is focused on the practical aspects involved in the execution of so modelled systems.
Keywords. 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, when we consider their *step sequence semantics* in which an execution is represented by a sequence of *steps*, *i.e.*, simultaneous occurrences of transitions. In this paper, we consider high-level Petri nets for modelling high-level parallel systems, with the aim to allow both *verification and execution of the specification*. Petri nets like those used in this paper are already used has a semantical domain for parallel programming languages or process algebra, *e.g.*, [3, 4]. These approaches could be directly applied to massively parallel languages or formalisms.

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.

This work is set in the context of *causal time* which allows to use "ordinary" (untimed) Petri nets by explicitly introducing a *tick transition* which increments *counters* used as clock watches by the rest of the system [2, 5]. It was shown in [1, 4] that the causal time approach is highly relevant since it is simple to put into practice and allows for *efficient verification* through model checking. In this context, the tick transition of a Petri net may causally depend on the other transitions in the net, which results in the so called *deadline paradox* [2]: tick is disabled until the system progresses. In the closed world of verification, this statement is logically equivalent to "the system has to progress before the next tick", which solves the paradox. But, this is not the case in the open world of execution.

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. 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. We will identify favourable situations, very easy to obtain, in which the reaction to a message is ensured within a short delay. An important property of our execution machine is that it preserves the step sequence semantics of the Petri net: this machine can be seen as an implementation of the Petri net execution rule including constraints related to the environment (real time and communication).

This paper is an extended abstract of a technical report which can be found at `http://www.univ-paris12.fr/lacl` where more details (and proofs) are given.

2 Basic definitions about Petri nets

This section briefly introduces the class of Petri nets that will be used in the following. We assume that the reader is familiar with the notion of multisets and we denote by $\mathsf{mult}(X)$ the set of all finite multisets over a set X.

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 $F \subseteq \mathbb{S}$ and $X \subseteq \mathbb{D} \cup \mathbb{V}$, we denote by $F \otimes X$ the set $\{a(x) \mid a \in F, x \in X\}$. 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.

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

- S is a non-empty finite set of *places*;
- T is a non-empty finite set of *transitions*, disjoint from S;
- ℓ defines the *labelling* of places, transitions and *arcs* (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 $\varnothing$ 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.

Petri nets are depicted as usual with several simplifications: the two components of transition labels are depicted separately; true guards and brackets around sets are omitted; arcs may be labelled by expressions as a shorthand, like $n + 1$ in the figure 1 page 4 which could be replaced by $y \in \mathbb{V}$ by adding a guard $y = n + 1$ to the transition t_τ.

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. 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 $M[U\rangle M'$ which naturally extends to sequences of steps. 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.

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 define the (*labelled*) *step sequence semantics* of a Petri net as the set containing all the sequences of (labelled) steps enabled by a net.

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 is very interesting for both theoretical and practical reasons. In particular, 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. As many previous works [5, 1, 4, 3], this paper only considers safe Petri nets.

3 Petri nets with causal time: CT-nets

The class of Petri nets we are actually interested in consists in safe labelled Petri nets, with several restrictions, for which we will define some specific vocabulary

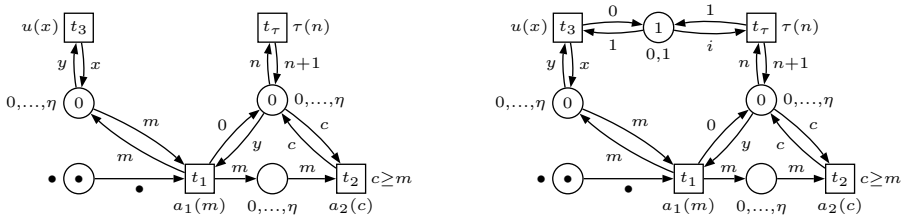

Fig. 1. On the left: an example of a CT-net, where $\eta > 0$, $\{a_1, a_2, u, \tau\} \subseteq \mathbb{S}$, $\{c, n, m, x, y\} \subseteq \mathbb{V}$ and $\{0, \ldots, \eta\} \cup \{\bullet\} \subseteq \mathbb{D}$. On the right: the tick-reactive version of this net, where $i \in \mathbb{V}$.

related to the occurrence of ticks. We assume that there exists $\tau \in \mathbb{S}$.

A *Petri net with causal time* (*CT-net*) is a safe Petri net (S, T, ℓ, M) in which there exists a unique $t_\tau \in T$ such that $\alpha(t_\tau) \in \{\tau\} \otimes (\mathbb{D} \cup \mathbb{V})$ and, for all $t \in T \setminus \{t_\tau\}$, we have $\alpha(t) \notin \{\tau\} \otimes (\mathbb{D} \cup \mathbb{V})$. Moreover, we impose that t_τ has at least one incoming arc labelled by a singleton. This transition t_τ is called the *tick transition* of the net. A *tick-step* is a step U 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.

The left of the figure 1 gives a toy CT-net in which the role of the tick transition 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 executable 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 $\eta > 0$).

Assuming $\eta \geq 5$, a possible execution of this CT-net is $\{\tau(0)\}\{u(2)\}\{a_1(2)\}$ $\{\tau(0), u(1)\}\{\tau(1)\}\{u(5)\}\{\tau(2)\}\{\tau(3)\}\{a_2(3), u(0)\}\{\tau(4)\}$.

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 non-empty 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, a non-tractable 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 not tractable because the transition t_3 can be executed infinitely often between two ticks: in the execution given above, the step $\{u(5)\}$ could be arbitrarily repeated.

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 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. 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.

A CT-net is *reactive* to a set of action symbols $R \subseteq \mathbb{S}_i$ if: (1) either, for all $a \in R$ and all $d \in \mathbb{D}_{io}$, it always allows, but never forces, the execution of a step containing $a(d)$; (2) or the net is in a marking M from which only actions in $\mathbb{S}_i \cup \mathbb{S}_o$ may ever be executed (M is called *final*). Thus, a net which is reactive to some actions will always allow a good responsiveness to the solicitation of the environment using these actions. It turns out that building such a net is very easy in general: for instance, the net given on the left of the figure 1 is reactive to $\{u\}$ (assuming $\mathbb{D}_{io} \subseteq \{0, \ldots, \eta\}$) thanks to the self loop on t_3.

Unfortunately, it can be shown that any CT-net N which is reactive $R \subseteq \mathbb{S}_i$ is not tractable. This negative result shows that the intuitive notion of reactiveness is too strong: the non-tractability actually indicates that a reactive CT-net is expected to be able to respond instantaneously to all the messages that the environment would send on a port in R. But 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. We thus assume that the environment may not produce more than one message on each port between two ticks, which leads to the new notion of tick-reactiveness.

We denote by $U[a]$ the number of occurrences of the action symbol a in a step U, *i.e.*, $U[a] \stackrel{\text{df}}{=} \sum_{a(x) \in U} U(a(x))$. Let N be a CT-net whose marking is M_0 and $R \subseteq \mathbb{S}_i$, consider an execution $M_0[U_1\rangle M_1 \cdots [U_k\rangle M_k$ of N such that only U_k may be a tick-step (if it is not, M_k should enable only the empty step), and define $U_0 \stackrel{\text{df}}{=} \varnothing$. Then, N is *tick-reactive* to R if: (1) for $0 \leq i < k$, the marking M_i is reactive to $R \setminus \cup_{0 \leq j \leq i} \{a \in R \mid U_j[a] > 0\}$; (2) N with the marking M_k is tick-reactive to R. This definition is inductive and holds over the executions of a CT-net. Intuitively, it states that N can react to any message sent on $a \in R$ after what it may miss them until the next tick, being then able to react again. 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 generally easy to transform a reactive CT-net into a tick-reactive one. For instance, the right of the figure 1 shows a modified version of our running example which is tick-reactive to $\{u\}$ and tractable (the step $\{u(5)\}$ could not be duplicated now).

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 none of its reachable markings enables a non-consistent step. Non-consistent steps are those during the execution of which several communications can take place on the same port. Since the transitions executed by a single step occur

simultaneously, this means that several distinct values may be sent or received on the same port at the same time. This is certainly something which is not realistic and should be rejected. The nets given in the figure 1 are both consistent.

4 Compilation and execution

We now show how to transform a tractable and consistent CT-net into a form more suitable to the execution machine. This corresponds to a compilation whose result will be an automaton (non-deterministic in general), called a *CT-automaton*, whose states will be the reachable markings of the net and whose transitions will correspond to the steps allowing to reach one marking from another. It should be remarked that this compilation is not 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.

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.

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 non-empty 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.

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 $a_i? = \circ \notin \mathbb{D}$ 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" (the communication has been correctly handled), "refused" (the communication could not been handled), "erroneous" (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 $a_i \leftarrow$ "mark" when an input port is being marked.

Let $(S_\mathcal{A}, T_\mathcal{A}, s_\mathcal{A})$ a CT-automaton and let Δ be an amount of time (defined below). 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 a set of input ports. The statement "**now**" evaluates to the current time when it is executed.

The behaviour of the machine is given in the figure 2. Several aspects of this algorithm should be commented:

```
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. 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).

- the "**for all**" loops are parallel loops;
- 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;
- one can show that the even occurrence of the ticks is ensured.

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 I. To start with, we assume 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 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. But this discussion is out of the scope of this paper.

The partial order $\prec$ is based on the lexicographic order on the vectors $V_A \in \{\text{"accepted"}, \text{"no mark"}, \text{"refused"}, \text{"erroneous"}\}^{\mathbb{S}_i}$ obtained by simulating the execution of each step A and assuming that the marks are ordered as given above. Again, it is clear that building these vectors and choosing the smallest is feasible in a fixed amount of time since the number of transitions outgoing from a given state is bounded. This is also feasible in parallel: all the V_A's can be computed in parallel and the selection of the smallest one is a 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.

Proposition 1. *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" except from a state which is a final marking of N. Moreover, if $a? = d \neq \circ$ before the execution of the line 7 in the figure 2, then a is marked "*accepted*" after the line 13 has executed.*

5 Conclusion

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 ensure. In order to obtain these results, several restrictions have been adopted: (1) only safe Petri nets are considered; (2) the nets must be *tractable*, *i.e.*, they cannot have unbounded runs between two ticks; (3) the nets must be *consistent*, *i.e.*, they cannot perform several simultaneous communications on the same port; (4) the 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 which can be ensured after measuring physical properties of the environment and choosing an appropriate computer. Future works may consider using non-safe Petri nets, but this class happens to be expressive enough for many interesting problems and there may be no real need to remove the first restriction.

Petri nets like CT-nets have been used for a long time as a semantical domain for high-level parallel programming languages and process algebras (see, *e.g.*, [3, 4]) and these techniques could be directly applied to massively parallel languages or formalisms. Considering the features of the execution machine combined with the fact that CT-nets allow for efficient verification, we obtain a framework in which the analysed model and the executed code are the same object, which saves from the risk of implementation errors.

References

1. C. Bui Thanh, H. Klaudel and F. Pommereau. *Petri nets with causal time for system verification.* MTCS 2002. ENTCS 68(5), Elsevier, 2003.
2. R. Durchholz. *Causality, time, and deadlines.* Data & Knowledge Engineering, 6. North-Holland, 1991.
3. H. Klaudel. *Compositional High-Level Petri nets Semantics of a Parallel Programming Language with Procedures.* SCP 41, Elsevier, 2001.
4. F. Pommereau. *Causal Time Calculus.* FORMATS'03. LNCS, Springer, to appear.
5. G. Richter. *Counting interfaces for discrete time modelling.* Technical report 26, GMD. September 1998.