**Software Tools for Technology Transfer manuscript No.**
(will be inserted by the editor)

# Modular State Space Exploration for Timed Petri Nets*

**C. Lakos**[1], **L. Petrucci**[2]

[1] University of Adelaide
Adelaide, SA 5005
AUSTRALIA
e-mail: Charles.Lakos@adelaide.edu.au

[2] LIPN, CNRS UMR 7030, Université Paris XIII
99, avenue Jean-Baptiste Clément
F-93430 Villetaneuse, FRANCE
e-mail: petrucci@lipn.univ-paris13.fr

Received: date / Revised version: date

**Abstract.** This paper extends modular state space construction for concurrent systems to cater for timed systems. It identifies different forms of timed state space and presents algorithms for computing them. These include uniprocessor algorithms inspired by conservative and optimistic approaches to discrete event simulation, and also a distributed algorithm. The paper discusses implementation issues and performance results for a simple case study.

## 1 Introduction

State space exploration is a convenient technique for the analysis of concurrent and distributed systems. Its chief disadvantage is the so-called state space explosion problem where the size of the state space can grow exponentially in the size of the system.

One way to alleviate the state space explosion problem is to use modular analysis, which takes advantage of the modular structure of a system specification. Here, the internal activity of the modules is explored independently rather than in an interleaved fashion. Experiments have indicated [15] that modular analysis can produce a significant reduction in the size of the state space, particularly for systems where the modules exhibit strong cohesion and weak coupling.

This paper extends modular state space exploration [4, 10] to timed systems. The introduction of time raises an interesting challenge since, by its very nature, time is a global entity rather than having local significance. This paper examines whether modular analysis algorithms can be modified to cater for time and still reap the benefits already demonstrated for untimed systems [10,15].

This work applies to Coloured Petri nets as well as to Place/Transition nets. For the sake of readability, we present the algorithms for Place/Transition nets. Their extension to CP-nets is straightforward.

The paper is organised as follows. Section 2 presents an example of a timed system — specifically a sliding window protocol. This serves to introduce the notions associated with a timed system, and also provide an example for which the benefits of modular state space exploration can later be demonstrated. Section 3 presents preliminary definitions of timed and modular Petri nets. Section 4 provides uniprocessor algorithms for modular state space exploration of timed systems, while Section 5 provides a distributed algorithm. Section 6 discusses implementation issues for the timed modular state space techique. In Section 7 we present experimental results obtained by applying this technique to the case study of Section 2. The conclusions are presented in Section 9.

## 2 A Timed Protocol Example

In this section, we describe an example derived from the *Timed Protocol* in [8]. Our model is presented in figure 1.

The protocol in [8] is a stop-and-wait protocol, i.e. a message is sent only if the previous one has been acknowledged. The model in figure 1 includes a *sliding window* and caters for *loss of messages or acknowledgements*.

The model is composed of four modules representing the *sender*, the *message channel* of the network, the *acknowledgement channel* of the network and the *receiver*. They are displayed on the left-hand side, the center and the right-hand side of the figure, respectively. The four modules communicate with their neighbours

* This work is supported by the Australian Research Council Linkage International grant LX04544639, and the French-Australian Science and Technology programme 09872RF.

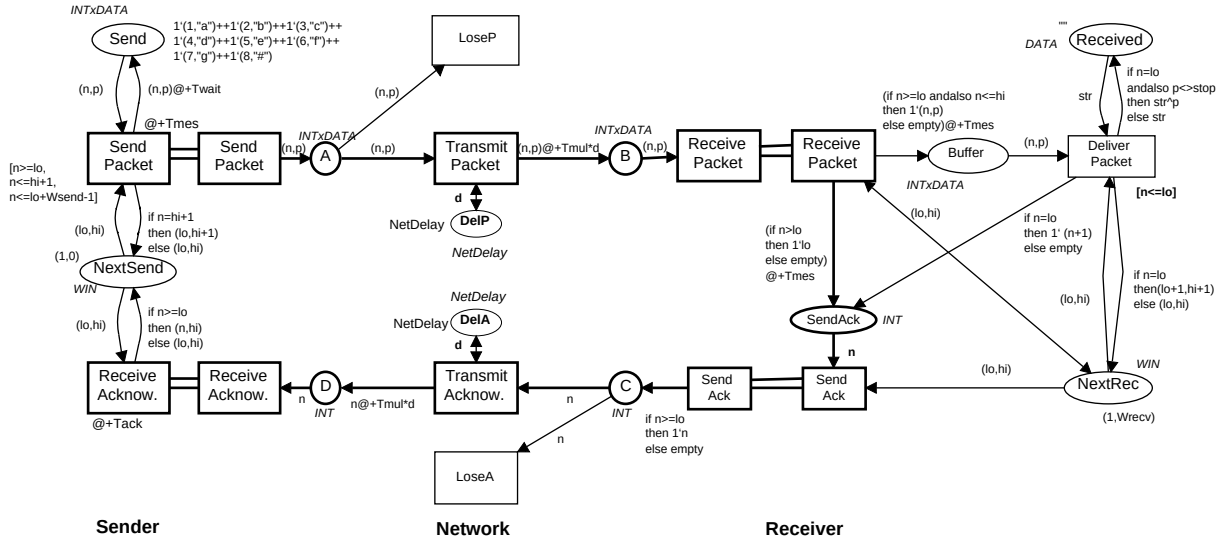


**Fig. 1.** Timed Protocol example

```
color INT = int timed;
color DATA = string;
color INTxDATA = product INT * DATA;
color WIN = product INT * INT;
color NetDelay = int with 3..5 declare ms;

var n, k, lo, hi : INT;
var p, str : DATA;
var d : NetDelay;

val stop = "#";
val Wsend = 2;      val Wrecv = 2;
val Tmes = 8;       val Tack = 4;
val Twait = 72;     val Tmul = 8;
```

**Fig. 2.** Declarations fot the Timed Protocol example

through fused transitions: `Send Packet`, `Receive Packet`, `Send Ack` and `Receive Ack`.

The data manipulated are described in figure 2. Colour sets are defined: `INT` is the set of integers with a timestamp, `DATA` contains strings to represent the messages inside the packets transmitted over the network, `INTxDATA` is pairs of elements of the previous two types, `WIN` is pairs of integers, and finally `NetDelay` allows for a range of different propagation delays over the network. A propagation delay is given by an element of `NetDelay` multiplied by the factor `Tmul`. Then, variables are declared, typed by these colour sets. Some constants are also defined: `stop` is the string terminating the message to send, `Wsend` and `Wrecv` are the sizes of the sending and receiving windows respectively, `Tmes`, `Tack` are the times to process each message and acknowledgement, respectively, while `Twait` is a timeout retransmission delay.

The *sender* module can only send packets or receive acknowledgements. The packets to send are initially stored in place `Send`. They are represented by a pair `(n,p)` where $n$ is the packet number and $p$ the data. We can refer to a packet consisting of the sequence number 1 and data *"a"*, available at time 0 by the notation *(1,"a")@0*. The sending operation takes some time as indicated by the `@+Tmes` label attached to transition `Send Packet`. When a packet is sent, its token remains in place `Send`, but its timestamp is incremented by the timeout retransmission delay `Twait`. With `Twait` set to 80 and `Tmes` set to 8, the token *(1,"a")@0* would be replaced by *(1,"a")*@`Tmes+Twait`*=(1,"a")@88*. The packets that can be sent must have a number comprised between the lowest not yet acknowledged (variable `lo`) and the first non-sent packet (`hi`+1). Moreover, the difference between these two bounds cannot exceed the window size `Wsend`. These conditions are all gathered in the guard of transition `Send Packet`. The lower and upper bounds are stored as a pair `(lo,hi)` in place `NextSend`.
The reception of an acknowledgement (transition `Receive Ack`) takes some time specified by a constant `Tack`. This is expressed by attaching `@+Tack` to the transition. The values of the bounds in place `NextSend` are updated if a message with a number `n` greater than `lo` is acknowledged. (Note that in this simple case study, we do not consider cyclic sequence numbers.)

The *network* stores packets sent in place `A`. Then, it can either lose them (transition `LoseP`) or transmit them (transition `Transmit Packet`). In that case, a delay corresponding to the time spent for transmission is applied to the packet, denoted $(n,p)@+Tmul*d$. The packet is then ready to be received. A similar scheme is applied to the acknowledgements. The main difference is that an acknowledgement is put in the network only if the packet `n` has not yet been acknowledged. This is indicated by the term associated with the arc from transition `Send Ack` to place `C`.

Finally, the module *Receiver* can receive packets numbered from the first expected one (`lo`) up to the size of the reception window (`hi=lo+Wrecv-1`). (Note that this allows for packets to arrive out of sequence.) Such a received packet is stored in place `Buffer` until the processing time `Tmes` has elapsed. Moreover, if the message is not the first one expected (`n>lo`), an acknowledgement numbered `lo` is prepared in place `SendAck`. This informs the sender that the number of the next message expected is `lo`. When a packet is in the receiver buffer, it can effectively be accepted and processed, via transition `Deliver Packet`. If it has a sequence number less than `lo`, then it has already been delivered and this is a duplicate that needs to be discarded. If its sequence number is equal to `lo`, then it is delivered and the receiving window is advanced — the contents `p` of the packet are concatenated with the contents previously received, thus forming a string stored in place `Received`; the receiving window is updated by incrementing both bounds; and an acknowledgement for the new lower bound `(n+1)` is prepared. When an acknowledgement is sitting in place `SendAck`, it can be transmitted to the *sender* via the *network* module.

## 3 Background

We commence with some modified definitions of Petri nets and their state spaces. Adapting the notation of Jensen [7], we write $5@2 + 2@3$ for 5 tokens (or values) available from time 2 and 2 tokens available from time 3. If we remove from this multiset, 3 tokens at time 4, we could end up with $4@2$ or $2@2 + 2@3$ or some other combination. We capture these notions formally as follows:

**Definition 1.** A **time set** $TS$ is a set of numeric values. For much of this paper, the time values will be integral, i.e. $TS = \mathbb{N}$, but in general they could be positive real numbers, i.e. $TS = \mathbb{R}^+$. Markings and arc inscriptions will be given by multisets over $TS$, written as $TS_{MS}$. We also extend operations over multisets to take time into account:

1. Given $m_1, m_2 \in TS_{MS}$, $m_1 \geq_T m_2$ iff $m_2 = \emptyset$ or $\exists m'_1, m'_2 \in TS_{MS}, m_{1i}, m_{2i} \in TS$ such that $m_1 = m'_1 + 1@m_{1i}$ and $m_2 = m'_2 + 1@m_{2i}$ and $m_{1i} \leq m_{2i}$ and $m'_1 \geq_T m'_2$.
2. Given $m_1, m_2, m_3 \in TS_{MS}$, $m_1 -_T m_2 = m_3$ iff $m_2 = \emptyset$ and $m_1 = m_3$ or $\exists m'_1, m'_2 \in TS_{MS}, m_{1i}, m_{2i} \in TS$ such that $m_1 = m'_1 + 1@m_{1i}$ and $m_2 = m'_2 + 1@m_{2i}$ and $m_{1i} \leq m_{2i}$ and $m'_1 -_T m'_2 = m_3$.
3. Given $m_1, m_2, m_3 \in TS_{MS}$, $m_1 +_T m_2 = m_3$ iff $m_1 + m_2 = m_3$.
4. Given $m \in TS_{MS}$ and $k \in TS$, $k +_T m = \sum_{m_i \in m} 1@(k + m_i)$ and $k -_T m = \sum_{m_i \in m} 1@(k - m_i)$.

The comparison operator is interpreted as the multiset $m_1$ having elements which have been accessible for at least as long as the demands specified by $m_2$. In other words, it must be possible to pair elements of $m_2$ with elements of $m_1$ such that the elements of $m_1$ are less than those of $m_2$, i.e. they have been accessible longer than the requirements. This interpretation of comparison is then used to define subtraction (between timed multisets) — $m_1 -_T m_2$ is only defined if $m_1 \geq_T m_2$. In general, $m_1 -_T m_2$ is not uniquely defined unless we insist that the element $m_{1i}$ is always chosen to be the maximum value that is less than the corresponding $m_{2i}$. This is the approach taken by Jensen [7] in order to ensure that $(m_1 -_T m_2) -_T m_3 = (m_1 -_T m_3) -_T m_2$ which is required for the diamond rule to hold. For completeness, we define addition of timed multisets but this is the same as multiset addition. Adding and subtracting multisets to scalars is similar to the scaling function of van der Aalst [1].

**Definition 2.** A **Timed Petri Net** is a quadruple $PN = (P, T, W, M_0)$, where $P$ is a finite set of **places**, $T$ is a finite set of **transitions** such that $T \cap P = \emptyset$, $W$ is the **arc weight function** mapping from $(P \times T) \cup (T \times P)$ into $TS_{MS}$, and $M_0$ is the **initial marking**, namely a function mapping from $P$ into $TS_{MS}$.

For a Timed Petri Net, each token has a time attribute, which indicates the earliest time that it is accessible. The output arcs of a transition indicate the time delays for generated tokens — an output arc with the inscription $5@2$ would indicate that 5 tokens are added to a place and they will become accessible 2 units of time in the future. We allow similar inscriptions on input arcs — an inscription $5@2$ would indicate the consumption of 5 tokens which have been accessible since (at least) 2 units of time in the past.

It would be possible to specify time delays only on the input arcs or only on the output arcs. If time delays are only specified on output arcs, then the time at which a transition can fire depends solely on the accessibility of the tokens. If time delays are only specified on the input arcs, then the time at which a transition can fire depends on transition-specific information. Our approach has the advantage of symmetry and generality. By contrast, the approaches of van der Aalst [1] and Jensen [7] only specify delays on output arcs.

**Definition 3.** A **marking** is a function $M$ mapping from $P$ into $TS_{MS}$. The set of all markings is denoted by $\mathbb{M}$. A transition $t$ is **time-enabled** at time $k$ in a marking $M$, denoted by $M[t\rangle_k$, iff $\forall p \in P : M(p) \geq_T k -_T W(p,t)$. When a transition $t$ is enabled in a marking $M_1$ at time k, it may **occur**, changing the marking $M_1$ to another marking $M_2$, defined by: $\forall p \in P : M_2(p) = (M_1(p) -_T (k -_T W(p,t))) +_T (k +_T W(t,p))$. This is denoted by $M_1[t\rangle_k M_2$. The set of markings **reachable** from a marking $M$, denoted $[M\rangle$, is given by the smallest

set satisfying $M \in [M\rangle$ and $M' \in [M\rangle \wedge M'[t\rangle_k M'' \implies M'' \in [M\rangle$.

**Definition 4.** The **timed state space** for a Timed Petri Net $PN = (P, T, W, M_0)$ is a pair: $TSS = (V, E)$ where $V = [M_0\rangle$ and $E = \bigcup_{m \in V} \{(m, t, m')_k \mid m[t\rangle_k m'\}$.

It is common to require that timed transitions fire at the earliest possible time of enabling. Accordingly, we define an earliest time state space.

**Definition 5.** The **earliest time state space** for a Timed Petri Net $PN = (P, T, W, M_0)$ is a pair: $ESS = (V, E)$ where $V = [M_0\rangle$ and $E = \bigcup_{m \in V} \{(m, t, m')_k \mid m[t\rangle_k m', \nexists k' < k : m[t\rangle_{k'}\}$.

Finally, we define a reduced earliest time state space as a graph where the transitions in conflict at any given marking all have the same time.

**Definition 6.** The **reduced earliest time state space** for a Timed Petri Net $PN = (P, T, W, M_0)$ is a pair: $RSS = (V, E)$ where $V = [M_0\rangle$ and $E = \bigcup_{m \in V} \{(m, t, m')_k \mid m[t\rangle_k m', \nexists t', k' < k : m[t'\rangle_{k'}\}$.

Note that $t'$ could be $t$ firing at an earlier time.

Note that the standard definition of state spaces for timed systems (as in Design/CPN [5] for example) are equivalent to our definition of a reduced earliest time state space. Unlike Jensen, however, we do not attach a time to a marking (to indicate the time of firing of the last transition), and then require that subsequent transitions should have a greater or equal time. This constraint is imposed to ensure that time cannot *go backwards*. Thus, if a marking $M_0$ enables independent transitions $t_1$ at time 2 and $t_2$ at time 5, then our definition of *ESS* would allow $t_1$ to be fired at time 2 *followed by* $t_2$ at time 5, *or vice versa*. With Jensen's constraint, the first alternative would still apply, but the second would have $t_1$ firing at time 5 after $t_2$. Our approach is somewhat anomalous, but we retain it because the anomalies will be eliminated in forming a *RSS* — $t_1$ will not be allowed to fire *after* $t_2$. Furthermore, for a modular system, it will be necessary to consider an *ESS* as a stepping stone to a *RSS*, and the standard unfolding of an earliest time modular state space will produce an *ESS*.

It should also be noted that, for a timed net, the *ESS* is a subgraph of the *TSS* since some edges are removed, and this may also make some nodes unreachable. Similarly, the *RSS* is a subgraph of the *ESS*. A *partially reduced earliest time state space* is also possible. This would be a subgraph of the *ESS* and a supergraph of the *RSS*.

**Definition 7.** A **Timed Modular Petri Net** is a pair $MN = (S, TF)$, where:

1. $S$ is a finite set of **modules** such that:
   - Each module, $s \in S$, is a Timed Petri Net: $s = (P_s, T_s, W_s, M_{0_s})$.
   - The sets of nodes corresponding to different modules are pair-wise disjoint: $\forall s_1, s_2 \in S : [s_1 \neq s_2 \Rightarrow (P_{s_1} \cup T_{s_1}) \cap (P_{s_2} \cup T_{s_2}) = \emptyset]$.
   - $P = \bigcup_{s \in S} P_s$ and $T = \bigcup_{s \in S} T_s$ are the sets of all places and all transitions.
2. $TF \subseteq 2^T \setminus \{\emptyset\}$ is a finite set of non-empty **transition fusion sets**.

The above definition of a Timed Modular Petri Net is identical to existing definitions [4,10] except for the introduction of time. Each module is a Timed Petri Net, and the modules interact via transition fusion — the elements of a fusion set fire as a single transition.

## 4 Modular Timed State Space Exploration

In this section we present two algorithms for modular state space exploration for a uniprocessor. (In section 5 we consider an algorithm suitable for a distributed environment.) The first algorithm is based on a conservative approach, which only explores transitions if their firing is consistent with the *RSS*. The second algorithm is based on an optimistic approach, which explores transitions if their firing is consistent with the *ESS*, with reduction left till later. These algorithms are consistent with the distinction between conservative and optimistic algorithms for distributed discrete event simulation [6]. Before handling the modular cases, we first present the algorithms for a flat timed net.

In this paper, we focus on state spaces built with a predetermined time limit, as computed by some tools such as DESIGN/CPN. To remove this limitation, it would be necessary to construct classes of timed markings, as in e.g. [2]. As we aim to construct a tool which might, in a distributed version, have the underlying structure of the distributed state space construction from [9], this limitation is not yet an issue.

### *4.1 Algorithms for a flat net*

An algorithm to generate an earliest time state space for a timed net is given in Fig. 3. It maintains a set `Waiting` of as-yet unexplored markings. At each iteration of the *repeat* loop, the current elements of `Waiting` are removed and examined for enabled transitions. As usual, function NODE.ADD($M'$) adds a node labelled with $M'$ to the graph and state $M'$ to set `Waiting`, provided it does not already exist. Similarly, ARC.ADD$(M, t, M')_k$ adds an arc to the graph. The algorithm terminates when stability is reached, i.e. when `Waiting` is empty.

The algorithm in Figure 3 includes a time limit (called `system_limit`) beyond which we do not explore transitions, and we always pick the earliest time at which a transition is enabled. It is these two aspects which differentiate this algorithm from the traditional algorithm for reachability analysis of an untimed net.

```
1: TS system_limit ←??
2: set Waiting ← ∅
3: NODE.ADD(M_0)
4: repeat
5:    for all M ∈ Waiting do
6:       Waiting ← Waiting \{M}
7:       for all t ∈ T, ∃k ≤system_limit:
         M[t⟩_k M' and ∄k' < k : M[t⟩_k' do
8:          NODE.ADD(M')
9:          ARC.ADD(M, t, M')_k
10:      end for
11:   end for
12: until stable
```

**Fig. 3.** Algorithm for earliest time state space.

```
1: TS system_limit ←??
2: TS system_time ← 0
3: set Waiting ← ∅
4: NODE.ADD(M_0)
5: while system_time ≤ system_limit do
6:    repeat
7:       for all M ∈ Waiting do
8:          for all t ∈ T, M[t⟩_system_time M' do
9:             NODE.ADD(M')
10:            ARC.ADD(M, t, M')_system_time
11:            Waiting ← Waiting \{M}
12:         end for
13:      end for
14:   until stable
15:   system_time ← system_time + 1
16: end while
```

**Fig. 4.** Algorithm for reduced earliest time state space.

More significant modifications are required to generate the reduced earliest time state space. If we restrict our attention to integral time, then we can maintain a variable (called `system_time`) which is the time for which we are prepared to consider transition enablings. Essentially, the *repeat* loop of lines 4-12 of Fig. 3 can be nested within a loop that increments `system_time` at each iteration, and then transitions are considered for enabling at time `system_time`. This guarantees that in a given marking, we will consider the transitions that can fire earliest. The problem now is that the markings in the set `Waiting` cannot be discarded immediately because, while they may not currently enable a transition, they may enable a transition at some future time.

Fig. 4 contains our modified algorithm for producing a reduced earliest time state space. States are only removed from set `Waiting` once we have found an enabled transition. If a state does *not* enable any transition at any time in the future, then it will remain in set `Waiting` forever, which is clearly inefficient.

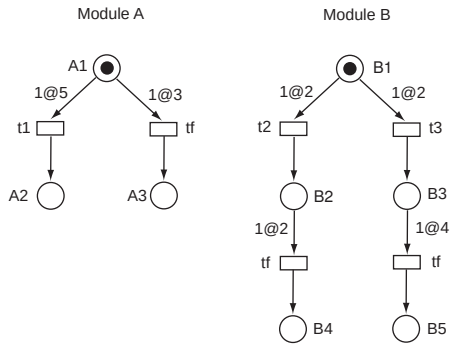


**Fig. 5.** Example of two timed modules

### *4.2 Modular algorithms*

In order to highlight the issues pertinent to the modular analysis of timed systems, we consider the simple example of Fig. 5. The transition input arcs indicate delays on tokens. Transitions $t_1$, $t_2$ and $t_3$ are local transitions, while transition *tf* is fused, with the occurrence in module $A$ needing to synchronise with one of the occurrences in module $B$. The corresponding local state spaces and the synchronisation graph are shown in Fig. 6, where the states indicate the marked places and the token timestamp(s).

In modular analysis, we explore the local state space of each module. The synchronisation graph captures the synchronisation points between the modules. The states of the synchronisation graph are system states, each of which is a tuple of module states. Thus, the state labelled *A1,0 B1,0* corresponds to the system state with module $A$ in state *A1* at time *0*, and with module $B$ in state *B1* at time *0*. The arcs of the synchronisation graph are labelled with the fused transitions together with their time of firing. Since the fused transition will normally fire only after some internal activity of the modules, the arcs are also labelled with the local states which enable the fused transition. Thus, the arc labelled *A1B2,tf,4* indicates that transition *tf* can fire at time *4* when module $A$ is in state *A1* and module $B$ has reached state *B2*. Thus, while module $A$ can fire *tf* at time *3*, it needs to wait till time *4*, when module $B$ is also ready.

The fact that a system state consists of a tuple of module states also means that the state of one module may be combined with states of multiple other modules, and decisions about reduced earliest time state spaces cannot be made merely at the local level. In the example, we cannot tell whether transition $t_1$ is preempted by transitions $t_2$ and $t_3$. Similarly, while we can determine if a local transition preempts a fused transition (in the same module), the converse is not possible. Thus, in module $A$, transitions $t_1$ and *tf* are in conflict. In the local state space, *tf* could preempt $t_1$, but *tf* needs to synchronise with *tf* in module $B$, which delays the firing time. With a short delay, *tf* may preempt $t_1$; with a longer delay *tf* may be preempted by $t_1$.

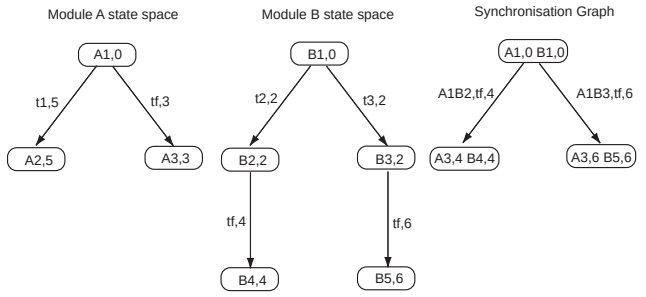


**Fig. 6.** Modular state space for two timed modules

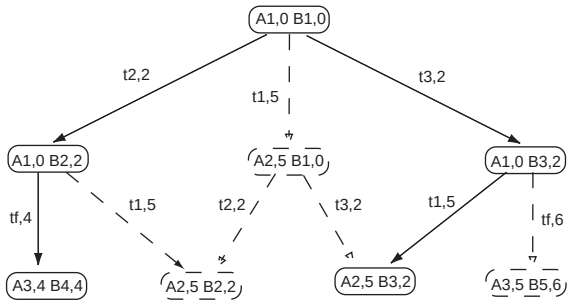


**Fig. 7.** Unfolded state space for two timed modules

In other words, questions about which transition preempts another can only be finalised in an unfolded state space as shown in Fig. 7. The broken arcs indicate transitions that are preempted in the unfolded state space. Thus, the local state space needs to be an earliest time state space rather than a reduced earliest time state space.

### *4.3 Conservative Algorithm*

The algorithm of Fig. 4 is now adapted to cater for modular analysis. The state space we construct is the timed extension of modular state spaces [4]. It consists of one local state space per module, describing only the module's internal behaviour, and a synchronisation graph capturing the interactions between modules.

Fig. 8 presents the algorithm for computing the synchronisation graph, while Fig. 9 presents the algorithm for computing the local state space of a module. As in Fig. 4, both the local state space and the synchronisation graph are computed in lock step — all activity at a time point is explored before time is advanced. This is the conservative approach.

Thus, in Fig. 8, `system_time` is incremented by one each time round the outer loop (up to `system_limit`) and all possible activity is investigated at each time point. Given that we are considering integral time, this means that we will consider transition firings at the earliest possible time. Note that the markings $M$ are immediately removed from `Waiting` even though they may enable a fused transition some time in the future. This

```
1: TS system_limit ←??
2: TS system_time ← 0
3: set Waiting ← ∅
4: NODE.ADD(M_0)
5: while system_time ≤ system_limit do
6:   repeat
7:     ∀i: trysynch_i ←EXPLORE(S_i, Waiting, system_time)
8:     Waiting ← ∅
9:     for all tf ∈ TF do
10:      for all (M, M') s.t. (M, M'_i, tf) ∈ trysynch_i ∨
         tf ∩ T_i = ∅ ∧ M'_i = M_i do
11:      if M'[tf⟩_system_time M'' then
12:        NODE.ADD(M'')
13:        ARC.ADD(M, (M', tf), M''_system_time)
14:      end if
15:    end for
16:  end for
17:  until stable
18:  system_time ← system_time + 1
19: end while
```

**Fig. 8.** Conservative algorithm for synchronisation graph.

is dealt with by having the local state space exploration retain the local markings between calls.

The function EXPLORE (in Fig. 9) returns a set of triples — the first element is a synchronisation node, the second is the local marking reachable from the first, and the third is the fused transition which is locally enabled at this reachable marking. This approach is required because EXPLORE only examines reachable markings in time slices up to the current `system_time`. In other words, EXPLORE will examine the local markings reachable from a synchronisation node over several calls to the function, and it is necessary to relate the locally reachable marking to the synchronisation node from which it was derived.

Thus, each call to EXPLORE returns such a set of triples in variable $\texttt{trysynch}_i$. From the results returned for all the modules, we build global marking pairs $(M, M')$, where the first element of the pair is a node in the synchronisation graph, and the second element corresponds to a marking locally reachable from there. If the second element enables a fused transition at the current time, then we add the appropriate node and arc to the synchronisation graph.

The logic underlying function EXPLORE$(S_i, Current, local_limit_i)$ is based on the one for a flat net, presented in Fig. 4. The variables, such as $\texttt{trysync}_i$, are subscripted to emphasise their local significance, the variable $\texttt{local\_time}_i$ replaces `system_time`, and the marking $M$ is replaced by marking pairs $(M, M'_i)$. Variable $\texttt{Waiting\_future}_i$ holds the marking pairs that might enable a transition in the future, while $\texttt{Waiting\_current}_i$ holds markings to be examined for transition enabling at the current time. Because these variables hold pairs of markings, NODE.ADD (at the local level) needs to have two arguments — it adds the second to the local state

```
1: static TS local_time_i ← 0
2: static set Waiting_future_i ← ∅
3: set Waiting_current_i ← ∅
4: set trysynch_i ← ∅
5: ∀M ∈ Current: NODE.ADD(M, M_i)
6: Waiting_current_i ← Waiting_future_i
7: if local_time_i < local_limit_i then
8:    local_time_i ← local_time_i + 1
9: end if
10: repeat
11:    for all (M, M'_i) ∈ Waiting_current_i do
12:       Waiting_current_i ← Waiting_current_i \ {(M, M'_i)}
13:       for all t_i ∈ T_i \ TF, M'_i[t_i⟩_local_time_i M''_i do
14:          NODE.ADD(M, M''_i)
15:          ARC.ADD(M'_i, t_i, M''_i)_local_time_i
16:          Waiting_future_i ← Waiting_future_i \ {(M, M'_i)}
17:       end for
18:       for all tf ∈ TF ∩ T_i, M'_i[tf⟩_local_time_i do
19:          trysynch_i ← trysynch_i ∪ {(M, M'_i, tf)}
20:       end for
21:    end for
22: until stable
23: return trysynch_i
```

**Fig. 9.** Conservative algorithm for local state space — EXPLORE($S_i, Current, local\_limit_i$).

space, and the pair of markings to both `Waiting_future`$_i$ and `Waiting_current`$_i$. Finally, lines 18-20 consider the enabling of the local component of a fused transition. Where such an enabling is found, the relevant triple is added to variable `trysynch`$_i$, which is then returned as the function result. These possible synchronisations will be repeatedly returned until they are preempted by local transitions.

By exploring local activity at each consecutive time point, EXPLORE caters for transitions preempting others at the local level. As noted in section 4.2, it cannot determine whether a local transition in one module preempts a local transition in another, nor whether a fused transition preempts a local transition. These issues can only be resolved in an unfolded state space.

### *4.4 Optimistic Algorithm*

In view of the limitations of the conservative algorithm (noted in section 4.3), we now consider an alternative algorithm, inspired by the optimistic approach to distributed discrete event simulation [6]. Here, we acknowledge that we can only guarantee producing a reduced earliest time state space at the local level or at the global level but not local relative to global.

Consequently, each call to function EXPLORE examines the local state space up to `system_limit` and not just `system_time`. Since we explore all relevant activity from a synchronisation node, and not just some small time slice, we can simply return pairs giving the locally reachable marking and the fused transition which it enables. The modified algorithm is presented in Figs. 10 and 11. Note that if EXPLORE is called multiple times with the same marking, then the locally reachable markings need to be recalculated or else some caching regime is required [12].

```
1: TS system_limit ←??
2: set Waiting ← ∅
3: NODE.ADD(M_0)
4: repeat
5:    for all M ∈ Waiting do
6:       Waiting ← Waiting \{M}
7:       ∀i : trysynch_i ← EXPLORE(S_i, M_i, system_limit)
8:       for all tf ∈ TF do
9:          for all M' s.t. (M'_i, tf) ∈ trysynch_i ∨
            tf ∩ T_i = ∅ ∧ M'_i = M_i do
10:            if M'[tf⟩_k M'' and ∄k' < k : M'[t⟩_k' then
11:               NODE.ADD(M'')
12:               ARC.ADD(M, (M', tf), M'')_k
13:            end if
14:         end for
15:      end for
16:   end for
17: until stable
```

**Fig. 10.** Optimistic algorithm for synchronisation graph.

```
1: set Waiting_i ← ∅
2: set trysynch_i ← ∅
3: NODE.ADD(M_i)
4: repeat
5:    for all M'_i ∈ Waiting_i do
6:       Waiting_i ← Waiting_i \ {M'_i}
7:       for all t_i ∈ T_i \ TF, M'_i[t_i⟩_k M''_i, ∄k' < k : M'_i[t_i⟩_k'
         do
8:          NODE.ADD(M''_i)
9:          ARC.ADD(M'_i, t_i, M''_i)_k
10:      end for
11:      for all tf ∈ TF ∩ T_i, M'_i[tf⟩_k M''_i, ∄k' < k :
         M'_i[tf⟩_k' do
12:         trysynch_i ← trysynch_i ∪ {(M'_i, tf)}
13:      end for
14:   end for
15: until stable
16: return trysynch_i
```

**Fig. 11.** Optimistic algorithm for local state space — EXPLORE($S_i, M_i, system\_limit$).

The optimistic approach simplifies both parts of the algorithm, but the cost is that it may produce larger local state spaces which will require further reduction in an unfolding of the modular state space.

## 5 Distributed Exploration

### *5.1 Basic Algorithm*

The distributed algorithm in this section follows the conservative paradigm of section 4.3 and relies on an archi-

```
1: current_time ← 0
2: repeat
3:    repeat
4:       message ← RECEIVE()
5:       if message == STATE M then
6:          NODE.ADD(M, M_i)
7:       else if message == TIMEINC then
8:          current_time ← current_time +1
9:       end if
10:   until message != STATE M
11:   if message != STOP then
12:      sync ← LOCAL_GENERATION(current_time)
13:      if sync == now then
14:         SEND(WAITING i)
15:      else if sync == future then
16:         SEND(STOPPED)
17:      else
18:         SEND(STUCK)
19:      end if
20:   end if
21: until message == STOP
```

**Fig. 12.** Distributed algorithm for local state space of module $i$.

```
1: Waiting_current ← Waiting_future
2: sync ← none
3: for all (M, M'_i) ∈ Waiting_current do
4:    for all t_i ∈ T_i \ TF, M'_i[t⟩_current_time M''_i do
5:       Waiting_future ← Waiting_future \{(M, M'_i)}
6:       NODE.ADD(M, M''_i)
7:       ARC.ADD(M'_i, t_i, M''_i)_current_time
8:    end for
9:    if ∃t, ENABLED_UNTIMED(M'_i, t) then
10:      sync ← future
11:   else
12:      Waiting_future ← Waiting_future \{(M, M'_i)}
13:   end if
14:   for all tf ∈ T_i ∩ TF, M'_i[tf⟩_current_time do
15:      sync ← now
16:      SEND(SYNC M M'_i tf)
17:   end for
18:   Waiting_current ← Waiting_current \{(M, M'_i)}
19: end for
```

**Fig. 13.** Function LOCAL_GENERATION(current_time)

tecture similar to that of [9]: several processes compute local parts of the state space while a single process handles the synchronisations.

Fig. 12 presents the algorithm to compute the local state space of module $i$. It focusses on the messages exchanged with the synchronisation process. The local generation, per se, is handled by function LOCAL_GENERATION, in Fig. 13.

The synchronisation process is also presented in two parts: the main one (Fig. 14) handles the communications with other processes and ensures the termination of all processes (if necessary), while function SYNCHRONISE, in Fig. 15, computes the synchronisation transitions enabled at the current time.

```
1: ∀i, trysync[i] ← ∅
2: ∀i, status[i] ← RUNNING
3: current_time ← 0
4: system_limit ←??
5: NODE.ADD(M_0)
6: ∀i, SEND(i, STATE M_0)
7: ∀i, SEND(i, SENT)
8: while ∃i, status[i] != STUCK ∧ current_time ≤ system_limit do
9:    for all i do
10:      while status[i] == RUNNING do
11:         message ← RECEIVE(i)
12:         if message == SYNC M M'_i tf then
13:            trysync[i] ← trysync[i] ∪ {(M, M'_i, tf)}
14:         else
15:            status[i] ←message
16:         end if
17:      end while
18:   end for
19:   SYNCHRONISE(∀i trysync[i],current_time)
20: end while
21: ∀i, SEND(i, STOP)
```

**Fig. 14.** Distributed algorithm for the synchronisation process.

```
1: new_sync ← false
2: for all tf ∈ TF do
3:    if ∀i synchronising on tf, ∃(M, M'_i, tf) ∈ trysync[i] then
4:       {a synchronisation is possible}
5:       for all M' such that ∀i, (M, M'_i, tf) ∈ trysync[i] : M'[tf⟩_current_time M'' do
6:          NODE.ADD(M'')
7:          ARC.ADD(M, (M', tf), M'')_current_time
8:          ∀i synchronising on tf, SEND(i, STATE M''_i)
9:       end for
10:      new_sync ← true
11:      ∀i synchronising on tf, status[i] ← RUNNING
12:   end if
13: end for
14: if new_sync then
15:    for all i such that status[i] == RUNNING do
16:       trysync[i] ← 0
17:       SEND(i, SENT)
18:    end for
19: else if current_time < system_limit then
20:    current_time ← current_time +1
21:    ∀ i, SEND(i, TIMEINC)
22: end if
```

**Fig. 15.** Function SYNCHRONISE(∀$i$ trysync[i],current_time)

The different processes communicate by exchanging messages, as in table 1.

Let us now explain the different algorithms. There is one local process per module. Initially, the current time (variable current_time) is set to 0 — it will be incremented when the synchronisation process sends the or-

| Message | Local | Sync. | Meaning |
|---|---|---|---|
| SYNC | ⟶ | | Send a marking |
| WAITING | ⟶ | | Has sent markings, waits for reply |
| STATE | ⟵ | | Synchronisation possible, send a marking |
| SENT | ⟵ | | All new states have been sent |
| TIMEINC | ⟵ | | No new synchronisation, hence increment time |
| STOPPED | ⟶ | | Nothing enabled at current_time |
| STUCK | ⟶ | | No future enabling |
| STOP | ⟵ | | Computation finished |

**Table 1.** Messages exchanged

der to do so. The local process executes a loop until the synchronisation process instructs it to stop. This loop receives markings and calls function LOCAL_GENERATION to explore transitions for current_time. Two sets are used: Waiting_current contains the marking pairs that may enable a transition at the current time, and Waiting_future contains the marking pairs that enable a transition later on. Note that the function creating a node, NODE.ADD adds elements to both of these sets.

After the local generation at current_time is completed, any states enabling a synchronised transition at the current_time are sent to the synchronisation process, followed by a WAITING message. The local process is then ready to receive new states obtained by synchronisation, which it will explore, or a TIMEINC message, indicating that no synchronisation was possible at the current time, and hence the time can be incremented. Otherwise, if the local process can neither fire a synchronised transition nor a local one at current_time, then either there are not enough tokens to enable a transition even in the future, in which case it tells the synchronisation process that it is STUCK, or else it says that it is STOPPED and waits until it is told to increment its time.

The synchronisation process also starts at current_time 0 and assumes that the status of all local processes is RUNNING. It performs a loop until all processes are STUCK or some maximum time system_limit has been reached. When this is the case, it tells all processes to STOP. In the loop, the synchronisation process receives and handles all messages sent by RUNNING local processes. These messages can either be states at which a synchronisation might be possible, or the new status of a local process. Function SYNCHRONISE is the core of the process. For each synchronised transition $tf$, it checks if it can occur, it computes and sends the resulting states to the local processes concerned. The status of these processes is updated to RUNNING (which is really the case when all the necessary operations have taken place). When all synchronisations have been done, a SENT message is sent to all processes that were involved, so that they can pursue their local construction at the same current_time. Their markings are removed from the appropriate set of trysync states. If no synchronisation has occurred, the set of trysync states is reinitialised, the current_time is incremented, and all local processes are told to increment their time.

### *5.2 Correctness*

To prove the correctness of the distributed algorithm, we will explain which markings and firings are handled by each part of each process.

The local processes construct only the local parts of the state space. Function LOCAL_GENERATION uses a set `Waiting_current` of markings which possibly enable a transition at the `current_time`. All markings $M$ of this set are dealt with one by one, and then deleted from `Waiting_current`. If $M$ does not enable any transition, independently of the time, it is also removed from the set `Waiting_future` of markings to be examined later. If $M$ enables a transition at `current_time`, its successors are built and added to both sets of markings. As the transitions enabled from $M$ are dealt with at `current_time`, $M$ can be removed from `Waiting_future` because future transitions have been preempted.

Hence, if no synchronised transition is enabled, the local state space is generated up to `current_time`. The local process then waits for instructions from the synchronisation process, which are either to increment time, if a transition may still be enabled in the future, or stop otherwise.

Whenever synchronised transitions are enabled at the `current_time`, the states enabling them are sent to the synchronisation process. Synchronisations may lead to new states which in turn may enable local transitions at `current_time`. Therefore, the local process waits for all states sent by the synchronisation process and then computes again the local parts at `current_time`.

The synchronisation process has the same `current_time` as the other processes. When it receives states, it tries to synchronise shared transitions, and eventually sends back the newly created states. If no synchronisation can occur, a message TIMEINC is sent to all local processes and `current_time` is incremented. The synchronisation process stops either when all processes are stuck or a maximum time is reached.

We conclude that transitions are handled in an *earliest firing time fashion.*

One of the key problems is the termination of the distributed algorithm. When a local process has finished its computation, it sends a message to the synchronisation process. The local process can either have sent markings on which a synchronisation is possible (it is then WAITING for an answer), or is STOPPED (nothing is enabled at the current time) or even STUCK (nothing will ever be enabled). When in state WAITING, if a synchronisation is possible, new markings will be sent to the local process, otherwise, a TIMEINC message will eventually be sent. The same message is sent when the process is STOPPED, so has markings to handle at a future time. Finally, when all local processes are STUCK or the time limit has been reached, all local processes receive a STOP message and end their computation. The synchronisation process stops as well.

## 6 Implementation issues

This section considers the simulation of time in the context of modular state space exploration in the *Maria* tool [12]. *Maria* supports modular state space construction but not time. In subsection 6.1 we consider the capabilities of *Maria* that are used to simulate time. The simulation is achieved by augmenting a *timed system net* with additional net components, which constitute a *generic timing infrastructure*. This infrastructure (which is presented in subsection 6.2) would not be required for a tool that supported timed analysis. The costs of this infrastructure are considered in subsection 7.5.

### *6.1 Maria capabilities used to simulate time*

As in section 3, tokens in the system net are assumed to have a time value which indicates when the token is accessible. Timing constraints on the firing of transitions will compare these time values with the current (local) time, which is stored as a token in a place *LocalTimer*.

In the absence of direct support for time in *Maria* [12], timing capabilities can be simulated using prioritised transitions. A high priority is allocated to any activity of the timed system net (whether local or synchronised), and a low priority is allocated to the infrastructure transitions. Thus, all possible activity of the system net is explored before time is advanced.

In actual fact, the standard distribution of *Maria* has a somewhat irregular implementation of prioritised transitions:

> *There is a simple priority method in the search algorithm of Maria that works as follows. When computing the successors of a marking, Maria investigates the transitions in the order they were defined in the model, from top to bottom. Whenever a transition having a nonzero priority class is found to be enabled, no further transitions of other priority classes will be analyzed in the marking. [13]*

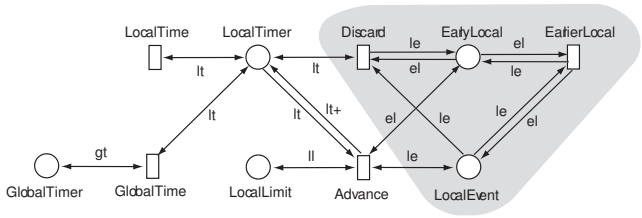


**Fig. 16.** Generic infrastructure for local time management

In other words, the numeric priorities in *Maria* do not imply an ordering of transitions, but simply serve to divide the transitions into a number of transition classes. The order of evaluation of transitions depends on the transition class, and also the order in which transitions appear in the net description. This approach is complicated by the modular structure of the net, and becomes unpredictable when *Maria* analyses a flattened version of a modular net. Therefore, *Maria* was modified so that transitions are sorted in priority order and considered for enabling in this order. Thus the simulation of timing capabilities using prioritised transitions (as described above), can be directly implemented.

*Maria* also provides support for code inclusion (using *#include* directives), and for conditional compilation (using *#ifdef*, *#ifndef*, *#else*, *#endif* directives). The code inclusion is used to incorporate the generic infrastructure wherever required. The conditional compilation is used to choose between the different approaches to time, whether incremental or using a calendar of pending events.

*Maria* also includes the possibility of *hiding* transitions. This can be used to produce results where the time management capabilities are hidden, thus reflecting the state space results which would be achieved if these capabilities were implemented rather than just being simulated.

### *6.2 Infrastructure to support time*

The main part of the generic infrastructure to simulate time at the module level is shown in Fig 16. There are three primary places which hold timer information — *GlobalTimer* holds a single token giving the current global time; *LocalTimer* holds a single token giving the current local time; and *LocalLimit* holds a single token with the limit to which time can be advanced locally.

As noted above, transitions of the system net are allocated a high priority (currently 9). Local transitions of the system net are fused with transition *LocalTime* — this provides access to the current local time, and

has priority 9. Synchronised transitions of the system net are fused with transition *GlobalTime* — it provides access both to the current local time and the current global time, and also has priority 9. (Note that transition *GlobalTime* is only enabled when the two time values coincide, so that different modules cannot synchronise unless their local times agree with the global time.)

Provided that the local limit is not exceeded, time can progress locally by firing the transition *Advance*, as indicated by the arc inscription *lt+*. This transition has priority 5, which is lower than other activity of the system net but higher than the priorities used for global time management. This allows for an optimistic approach to time, where local time can advance ahead of global time. Computation of the next local time (if required) will be at priorities between 5 and 9, while the global time management will be at priority less than 5.

The shaded portion of the generic infrastructure (in Fig 16) is optional and indicates net components used to maintain a calendar of pending events. The place *LocalEvent* holds the time values for known future events. When a transition fires and produces tokens which will become accessible to local transitions at some time in the future, that time value is added to place *LocalEvent*. The place *EarlyLocal* holds a single token giving the earliest pending event time. If this time value is less than that of the token in place *LocalTimer*, then transition *Discard* is enabled, and the token will be replaced by another from place *LocalEvent*. If there is a token in place *LocalEvent* which has a value less than the token in place *EarlyLocal*, then transition *EarlierLocal* can fire and swap the two tokens. Transitions *Discard* and *EarlierLocal* are allocated priorities 7 and 6 respectively, so that the computation of the next local time occurs after any enabled activity of the system net, but before transition *Advance*.

Each module can manage a similar calendar of pending events which may enable synchronised transitions. This involves places *EarlyGlobal* and *GlobalEvent*, and transitions *DiscardGlobal* and *EarlierGlobal*. Instead of transition *Advance* there is a transition *SynchClock* where each module makes its own earliest global event time accessible, and the clock logic chooses the minimum and distributes this to all modules. Transitions *DiscardGlobal* and *EarlierGlobal* are allocated priorities 4 and 3 respectively.

The above infrastructure allows for a number of timing combinations. For a conservative approach to distributed time, the local limit can be kept the same as the local time. This ensures that transition *Advance* is never enabled and the only way to advance the local time is by resynchronising with the global clock (which also serves to reset the local limit). By contrast, an optimistic approach to distributed time can be achieved by setting the local limit to the maximum simulation time. Then, a module's local time can advance up to the local limit, which goes beyond the current global time. The chosen priorities ensure that the local activity is explored before global time is advanced. Note, however, that only the firing of local transitions is explored (ahead of the global time).

Without calendar(s) of pending events, time will be integral and will always advance by steps of 1. With the inclusion of calendars, time can advance to that of the next pending event, in which case dense time can be supported.

Finally, it should be noted that while we can simulate an optimistic approach to distributed time, the possible benefits cannot be realised until the technique is implemented in the tool. This is because the presence of timing values in places *GlobalTimer*, *EarlyGlobal* and *GlobalEvent*, will discriminate states according to the global time at which they were explored, and not just the local time.

## 7 Experimental results

In this section, we apply two variants of the conservative uniprocessor algorithm to the case study of section 2. In subsection 7.1 we make some preliminary observations about the interplay of timing with modular analysis. In subsections 7.2 and 7.3 we present results for long and short retransmission timeout periods, respectively, while in subsection 7.4, we present results for a different modularisation of the system. Finally, in subsection 7.5, we present results illustrating the costs of the generic infrastructure.

### *7.1 Timing Implications for Modular Analysis*

It has already been observed [10] that the benefits of modular state space exploration are most apparent for systems exhibiting strong cohesion and weak coupling. For a weakly coupled system, the modular approach avoids exploring the many possible interleavings of the internal activity of the component modules.

The introduction of time into a system typically changes its state space from a broad structure (where many events can occur at any stage) to a narrow structure (where the number of events that can occur at any stage is significantly constrained by the progress in system time).

The interplay between these two aspects can be illustrated in terms of the case study from section 2. Firstly, we note that the case study is relatively simple, though we have added some complexity (as compared to the original in [8]) by extending it from a stop-and-wait protocol to a sliding window protocol.

Now, without the inclusion of time, the sending of messages, their transmission, reception and acknowledgement could all happen in an interleaved fashion. For example, the reception of one message in the *receiver* could be interleaved with the sending (or even retransmission)

of other messages by the *sender*. Modular state space exploration would avoid enumerating all these possible interleavings, thus providing significant efficiency gains.

On the other hand, if time is included as in figures 1 and 2, then the situation is very different. The sending of messages by the *sender* is separated by at least `Tmes` (here 8) time units. The retransmission of a message can only occur after `Tmes+Twait` (or 88) time units. The packets have a median propagation time of `Tmul*4` (or 32) time units. Then, reception of the message takes `Tmes` (or 8) time units to process. This timing can remove most of the interleaving of module activity — the *sender* sends the message at absolute time 8, the message channel forwards the packet at absolute time 40, the receiver processes the packet and sends an acknowledgement at time 48, the acknowledgement channel forwards the acknowledgement at absolute time 80, the acknowledgement is processed at time 84, which then preempts retransmission of the message. Variations in propagation time will vary the absolute time values but not the strict ordering of events.

Thus, if one is faced with the state space exploration of a timed system, the benefits of a modular approach are only going to be realised if the modules exhibit activity which overlaps at the same point of time. These considerations are to be observed in the results which are presented below.

In general, we benchmark the modular results produced using *Maria* against those obtained for a flat system (i.e. without modules) using Design/CPN [5]. Also, the results are generally computed using a calendar of pending events rather than incrementing time by one, so as to reduce the infrastructure costs. Also, the costs of the generic infrastructure are hidden. A comparison between incremental time and the use of a calendar is presented in subsection 7.5, together with an examination of the infrastructure costs.

Since the main goal of the modular state space technique is to reduce the size of the state space, we have chosen to characterise the experimental results by the number of nodes.

### *7.2 Long retransmission timeout delays*

We first present results for the sliding window protocol with a long retransmission timeout delay — here the constant `Twait` is set to 80. As discussed in subsection 7.1 this means that there will be minimal interleaving of activities between the various modules. However, the amount of activity in each module can be controlled by the variations in propagation times and window sizes.

Given in figures 17-22 are a range of results as the above parameters are varied and as the total simulation time varies from 60 to 200 time units.

The figures show that modular analysis produces a synchronisation graph which is 25-50% the size of the full state space.

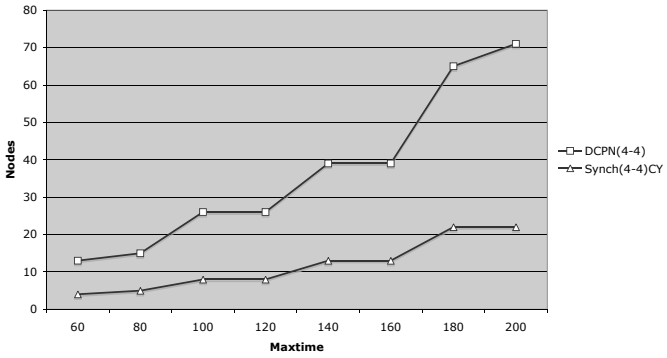


**Fig. 17.** Long timeout, window 1, `Netdelay={4..4}`

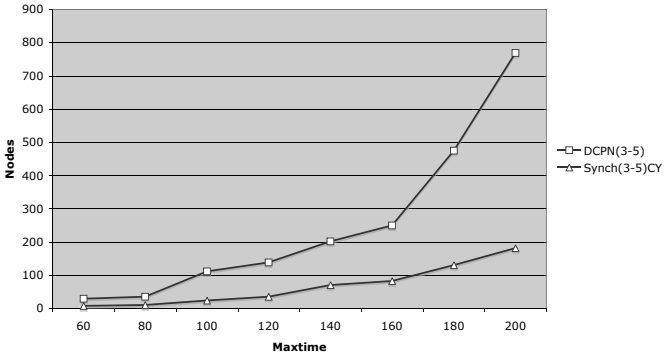


**Fig. 18.** Long timeout, window 1, `Netdelay={3..5}`

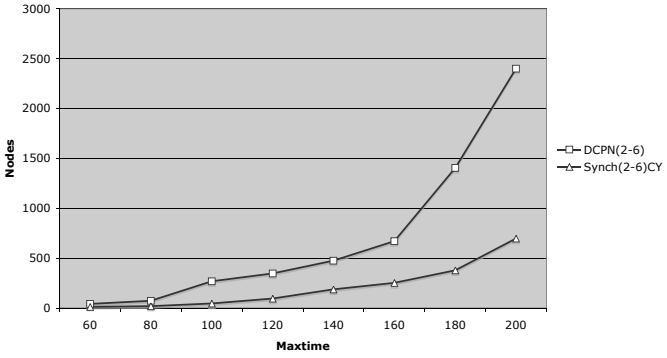


**Fig. 19.** Long timeout, window 1, `Netdelay={2..6}`

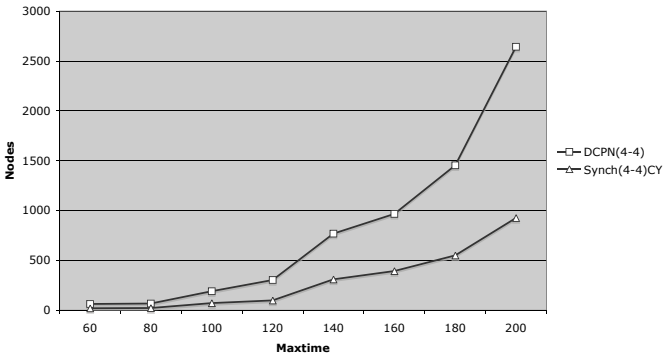


**Fig. 20.** Long timeout, window 2, `Netdelay={4..4}`

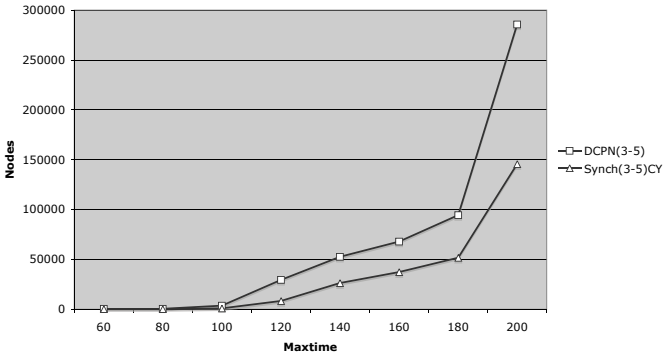


**Fig. 21.** Long timeout, window 2, `Netdelay={3..5}`

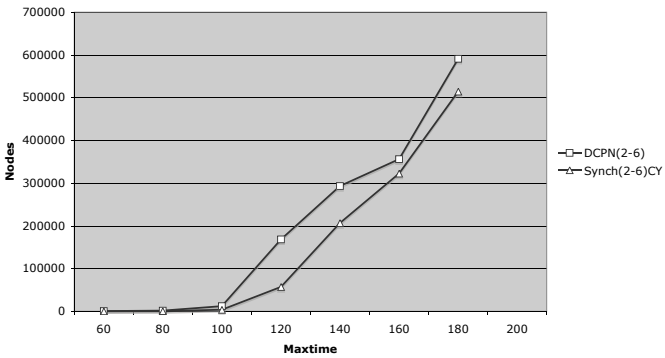


**Fig. 22.** Long timeout, window 2, `Netdelay={2..6}`

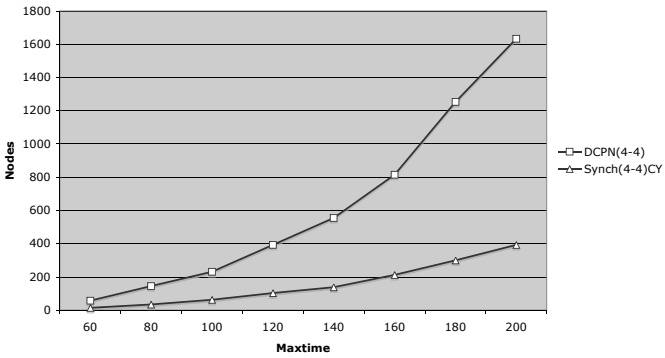


**Fig. 23.** Short timeout, window 1, `Netdelay={4..4}`

### *7.3 Short retransmission timeout delays*

With a long retransmission timeout delay, the behaviour in the various modules is mostly time-ordered rather than being interleaved. As indicated in 7.1, the benefits of modular state space exploration in this case are limited. However, if the retransmission timeout delay is significantly reduced, then the level of concurrent activity is increased. We now set the constant `Twait` to 16, giving a delay of 24 before a message is retransmitted.

Given in figures 23-27 are a range of results as the window size and propagation delay are varied and as the total simulation time varies from 60 to 200 time units.

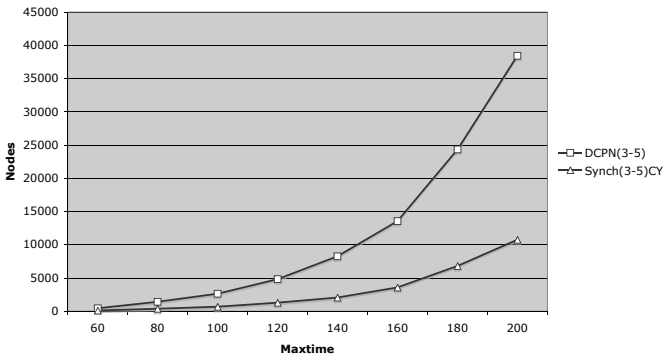


**Fig. 24.** Short timeout, window 1, `Netdelay={3..5}`

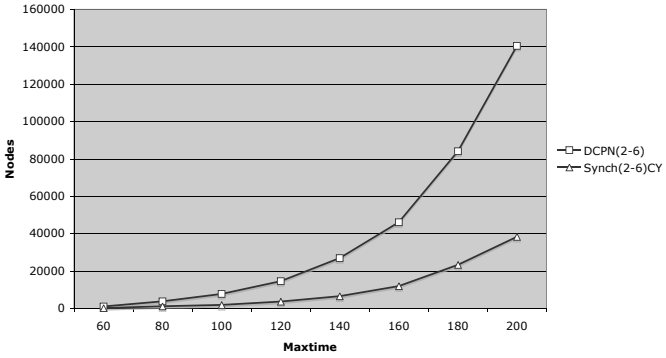


**Fig. 25.** Short timeout, window 1, `Netdelay={2..6}`

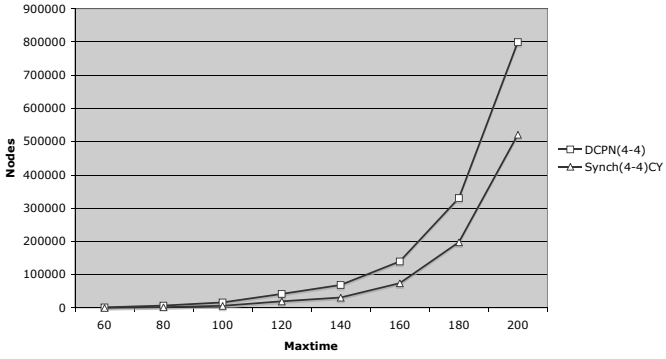


**Fig. 26.** Short timeout, window 2, `Netdelay={4..4}`

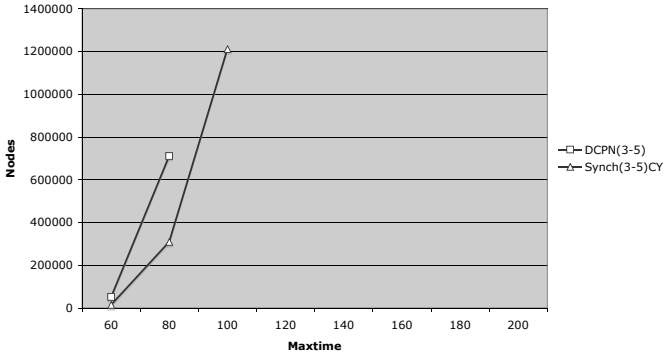


**Fig. 27.** Short timeout, window 2, `Netdelay={3..5}`

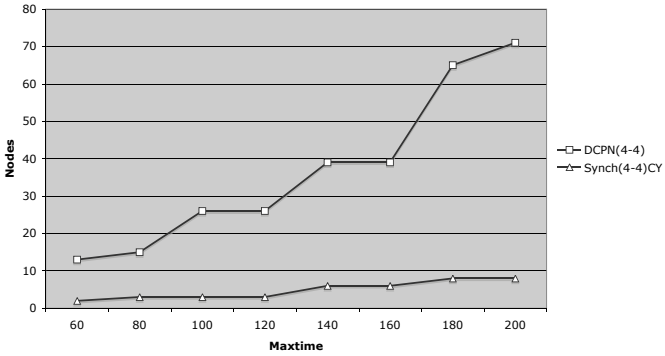


**Fig. 28.** Fewer modules, window 1, `Netdelay={4..4}`

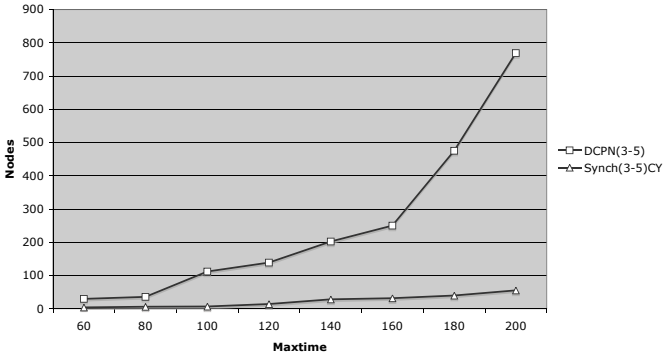


**Fig. 29.** Fewer modules, window 1, `Netdelay={3..5}`

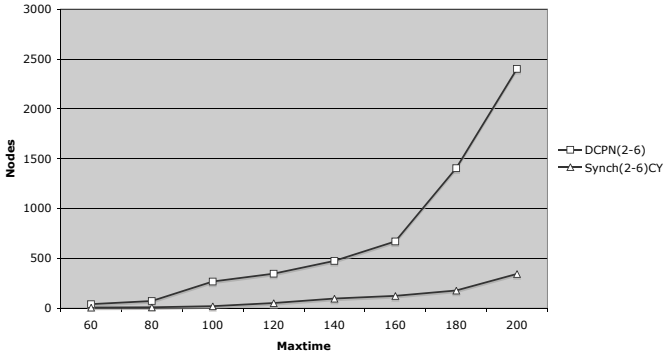


**Fig. 30.** Fewer modules, window 1, `Netdelay={2..6}`

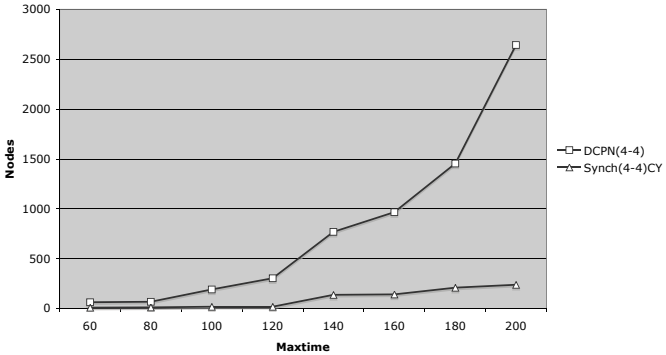


**Fig. 31.** Fewer modules, window 2, `Netdelay={4..4}`

These figures display some improvement in the benefits of modular state space exploration, but they are not extensive. This is because most internal actions result in a corresponding synchronisation action, and hence the level of internal activity is limited. For example, the different propagation delays used in firing transition *Transmit Packet* will result in different occurrences of transition *Receive Packet* (in fig. 1).

### *7.4 Different modularisation*

Another possiblity for increasing the interleaving of activity between modules is to modify the partitioning of the system into modules. Here, we investigate the merging of the modules for the *sender* and the *message channel*, and also for the *receiver* and the *acknowledgement channel*. Intuitively, the effect of this is to combine the variability of the sender and the associated propagation delays (and possible loss) into the one module, thereby hiding some of this from the other modules. We have used the long retransmission timeout delay of section 7.2. The corresponding results are given in figures 28-33.

Here, the benefits of modular state space exploration are much more dramatic. The use of modular techniques can now make the difference between being able to analyse a particular configuration or not being able to do so.

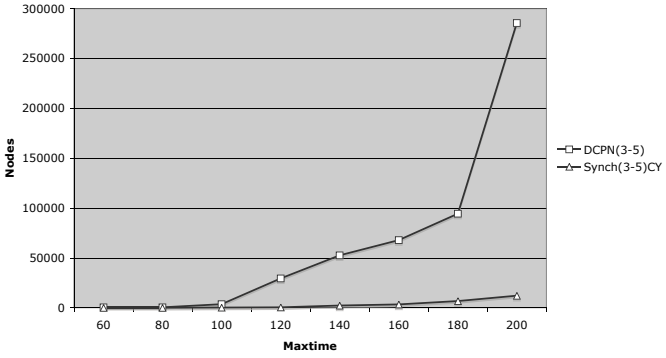


**Fig. 32.** Fewer modules, window 2, `Netdelay={3..5}`

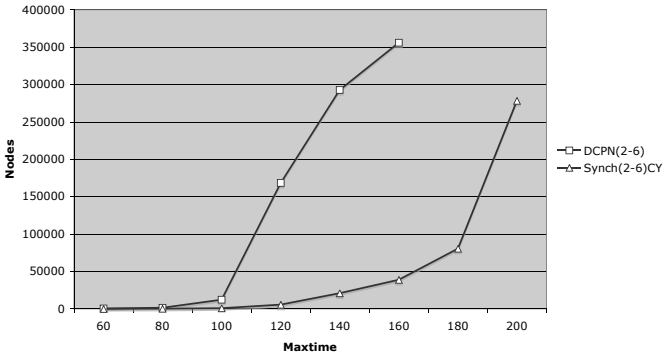


**Fig. 33.** Fewer modules, window 2, `Netdelay={2..6}`

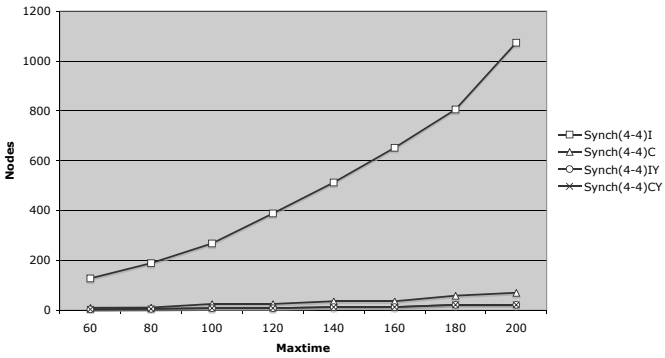


**Fig. 34.** Infrastructure costs, window 1, `Netdelay={4..4}`

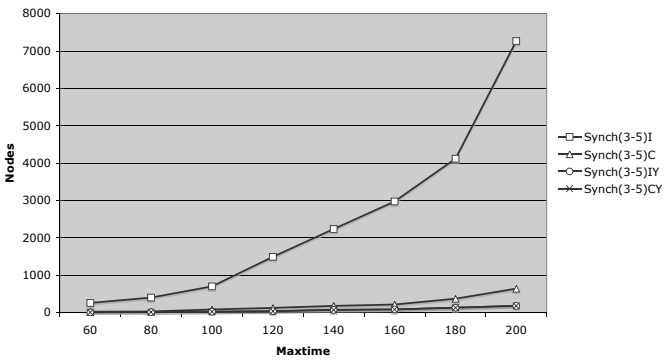


**Fig. 35.** Infrastructure costs, window 1, `Netdelay={3..5}`

### *7.5 Infrastructure costs*

Finally, we present some results which indicate the costs of the generic infrastructure. We use a long timeout delay and, for each combination of window size and propagation delay, we compare the total cost for using incremental time and for using a calendar of pending events, both with and without the infrastructure costs being hidden.

As usual, figures 34-39 present a range of results as the window sizes and propagation delays are varied and as the total simulation time varies from 60 to 200 time units. In the graph legends, the suffixes $I$ and $C$ distinguish between $I$ncremental time and the use of a $C$alendar. The suffix $Y$, if present, indicates that option $Y$ was used in the run of *Maria*, which serves to hide the states and transitions which are solely related to the infrastructure.

It can be observed that the infrastructure costs for incremental time are significantly greater than those for the use of a calendar. By contrast, the infrastructure costs for a calendar of pending events are relatively minor. It is also interesting to note that, with the infrastructure costs hidden, the results for incremental time and a calendar of pending events are largely indistinguishable. This is an interesting result, given that the calendar of pending events adds a significant amount of data to each state.

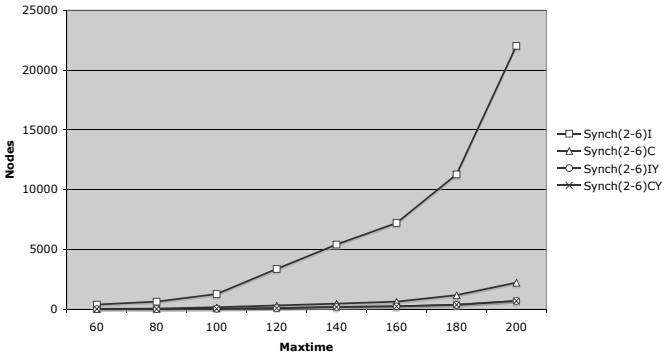


**Fig. 36.** Infrastructure costs, window 1, `Netdelay={2..6}`

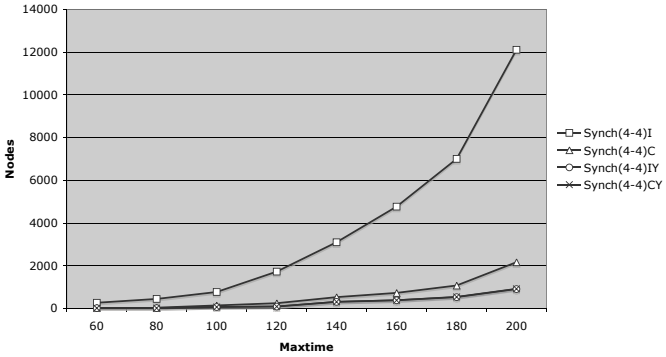


**Fig. 37.** Infrastructure costs, window 2, `Netdelay={4..4}`

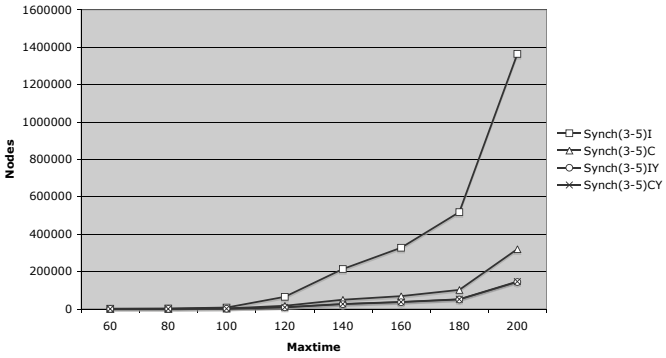


**Fig. 38.** Infrastructure costs, window 2, `Netdelay={3..5}`

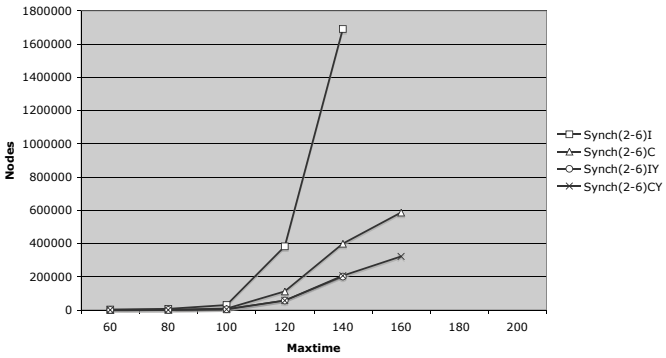


**Fig. 39.** Infrastructure costs, window 2, `Netdelay={2..6}`

## 8 Related work

There has been extensive work on the model-checking of timed systems, both *Time Petri nets* (TPNs) and *Timed Automata* (TAs). For a recent survey see [14]. For TPNs, transitions are associated with a time interval, during which the transition may fire (relative to the time when the transition became enabled). The state space is structured by pairs of markings and clocks — there is one clock per transition indicating the time since the transition became enabled. The state space is partitioned into equivalence classes or convex regions, the equivalence capturing the similar time response of the system. Clocks are reset to zero when the transition becomes enabled, but otherwise advance at the same rate. Region boundaries are determined by the firing intervals (associated with the transitions) and the time constraints in the temporal property being checked.

There has also been work on model-checking of timed systems with partial order reduction [16]. Here, the transitions explored at each point in time are not just the fireable transitions, but a subset of these — those which are visible (with respect to the formula being model-checked) and which affect the enabling of each other. The reduced state graph is *time stuttering equivalent* to the original state graph.

Another, different, approach is that of Berthelot and Boucheneb [2,3]. Here, state classes are identified by the timing properties since the last transition fired, together with a history of the relevant intervals associated with tokens which still belong to the state.

The work presented in this paper has focussed on modular state space exploration rather than the more general issue of model-checking of timed systems. Unlike the model-checking approaches considered above, we do not restrict attention to 1-safe nets. This means that it is not possible to attach a single clock to each transition. Rather, timestamps are attached to tokens, and if multiple tokens are resident in a place, a choice of the token(s) to be consumed may need to be made. Individual delays (rather than delay intervals) are attached to input and output arcs so as to limit the accessibility to the tokens by the passage of time. Transitions then fire at the earliest possible time (unless precluded by conflicting transitions). In the above-mentioned model-checking approaches, there are a number of clocks, one per transition. However, they advance at the same rate (unless they are reset when a transition becomes enabled). The modular approach also has a number of clocks, but these are associated with modules, and these clocks can advance at different rates (under the optimisitic view of time).

Both partial order reduction and modular state space exploration take advantage of some notion of independence between transitions. Partial order reduction restricts attention at each state to a mutually dependent subset of the enabled transitions, the dependency being determined by access to the global state. Modular state space exploration builds on the fact that the local transitions of one module are largely independent of the local transitions of another module. The precise relationship between these two forms of dependence is an open question, though there has been some work on using partial order reduction to make modular state space exploration more efficient [11].

In view of the above, we consider the work on modular state space exploration as largely orthogonal to the above-mentioned work on model-checking of timed systems and partial order reduction. Future work may consider extending the modular techniques to investigate their applicability to state classes, and even to temporal model checking.

## 9 Conclusions and Further Work

This paper has extended the definition of state spaces to cater for timed systems. It has identified an earliest time state space, where all transitions enabled in a marking occur at the earliest time possible. It has also identified a reduced earliest time state space, where transitions enabled at a state are preempted by others which can occur earlier. The latter is the more traditional approach to timed state spaces. However, for modular systems, the independent analysis of the modules means that a reduced earliest time state space cannot be determined except in the unfolded state space. This being the case, it is appropriate to generate an earliest time state space as part of modular exploration.

The above raises the challenging question whether, as for earlier results [10], timing properties can be determined from the timed modular state space *without* unfolding. This is an important issue for future work.

This paper has presented two algorithms for a uniprocessor and one for a distributed environment which have generated the timed modular state space. The uniprocessor algorithms were inspired by the conservative and optimistic approaches to distributed discrete event simulation. The optimistic approach is based on generating the earliest time state space.

The formal definitions allow for dense time but the algorithms are defined for integral time. The algorithms can be modified to handle dense time by maintaining a schedule of pending events, e.g. in a priority queue. While this does not constitute a significant change, it would unnecessarily clutter the presentation of the algorithms. A more challenging issue for further work consists in adapting the algorithms to the generation of state class graphs, as in [2]. This would alleviate the maximum system time constraint.

Experimental results have been presented for the conservative uniprocessor algorithm. These demonstrate the value of modular analysis for timed systems, provided that the activity of different modules do overlap in time.

We have also compared results for the use of incremental time as opposed to the use of a calendar of pending events. In the former case, the overheads introduced can be quite significant, while in the latter case they are relatively minimal. Further, the addition of the calendar of pending events, which extends the data stored with each state, seems to have little impact on the size of the synchronisation graph.

We have also derived preliminary results for the optimistic uniprocessor algorithm, and they indicate that this approach does reduce the amount of synchronisation between modules without necessarily resulting in a lot of superfluous exploration of the local state space. However, the optimistic algorithm does need to be paired with the use of a schedule of pending events, or else each module from each synchronisation node will increment time up to the time limit looking for possible enabled transitions. It also needs to be implemented rather than just simulated or else the local states are differentiated not just by the local time, but also by the global time of the preceding synchronisation node. It will be important to perform further experiments to see whether the preliminary results for the optimistic algorithm carry over to more realistic case studies. It will also be important to experiment with a number of generalisations and optimisations that we have identified, in order to see whether they are of value in fine-tuning the algorithms.

## References


1. W. van der Aalst. Interval Timed Coloured Petri Nets and their Analysis. In M.A. Marsan, editor, *Internation Conference on the Application and Theory of Petri Nets*, volume 961 of *LNCS*, pages 453–472, Chicago, 1993. Springer.
2. G. Berthelot and H. Boucheneb. Occurrence graphs for Interval Timed Coloured Nets. In *Proc. 15th Int. Conf. Application and Theory of Petri Nets (ICATPN'1994), Zaragoza, Spain, June 1994*, volume 815 of *LNCS*, pages 79–98. Springer, June 1994.
3. H. Boucheneb and G. Bertholet. Towards a Simplified Building of Time Petri Nets Reachability Graph. In *5th International Workshop on Petri Nets and Performance Models*, pages 46–55. Springer, 1993.
4. S. Christensen and L. Petrucci. Modular analysis of Petri nets. *The Computer Journal*, 43(3):224–242, 2000.
5. DESIGN/CPN *online*. `http://www.daimi.au.dk/designCPN`.
6. R.M. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):30–53, 1990.
7. K. Jensen. *Coloured Petri Nets: Basic concepts, analysis methods and practical use. Volume 1: Basic Concepts.* Monographs in Theoretical Computer Science. Springer, 1992.
8. K. Jensen. *Coloured Petri Nets: Basic concepts, analysis methods and practical use. Volume 2: Analysis Methods.* Monographs in Theoretical Computer Science. Springer, 1994.
9. L. Kristensen and L. Petrucci. An approach to distributed state space exploration for coloured Petri nets. In *Proc. 25th Int. Conf. Application and Theory of Petri Nets (ICATPN'2004), Bologna, Italy, June 2004*, volume 3099 of *LNCS*, pages 474–483. Springer, June 2004.
10. C. Lakos and L. Petrucci. Modular analysis of systems composed of semiautonomous subsystems. In *Proc. 4th Int. Conf. on Application of Concurrency to System Design (ACSD'04), Hamilton, Canada, June 2004*, pages 185–194. IEEE Comp. Soc. Press, June 2004.
11. G.A. Lewis. *Incremental Specification and Analysis in the Context of Coloured Petri Nets.* PhD, Department of Computing, University of Tasmania, 2002.
12. M. Mäkelä. Model Checking Safety Properties in Modular High-Level Nets. In W. van der Aalst and E. Best, editors, *24th International Conference on the Application and Theory of Petri Nets*, volume 2679 of *LNCS*, pages 201–220, Eindhoven, The Netherlands, 2002. Springer.
13. M. Mäkelä. Maria - Modular Reachability Analyzer for Algebraic System Nets (Version 1.3.4). Technical report, Helsinki University of Technology, Laboratory for Theoretical Computer Science, June 2003 2003.
14. W. Penczek and A. Polrola. Specification and Model Checking of Temporal Properties in Time Petri Nets and Timed Automata. In J. Cortadella and W. Reisig, editors, *International Conference on the Application and Theory of Petri Nets*, volume 3099 of *Lecture Notes in Computer Science*, pages 37–76, Florida, 2004. Springer.
15. L. Petrucci. Cover picture story: Experiments with modular state spaces. *Petri Net Newsletter*, 68:Cover page and 5–10, April 2005.
16. I. Virbitskaite and E. Pokozy. A Partial Order Method for the Verification of Time Petri Nets. In G. Ciobanu and G. Paun, editors, *Fundamentals of Computation Theory*, volume 1684 of *Lecture Notes in Computer Science*, pages 547–558. Springer, 1999.