A Simple Time Scale Decomposition Technique for Stochastic Process Algebras

JANE HILLSTON[1] and VASSILIS MERTSIOTAKIS[2]

[1] *Department of Computer Science, University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, Scotland.*
email: jeh@dcs.ed.ac.uk

[2] *Universität Erlangen-Nürnberg, IMMD VII, Martensstr. 3, D-91058 Erlangen, Germany.*
email: vsmertsi@informatik.uni-erlangen.de

Many modern computer and communication systems result in large, complex performance models. The compositional approach offered by stochastic process algebra constructs a model from submodels which are smaller and more easily understood. This gives the model a clear component-based structure. In this paper we present cases when this structure may be used to inform the solution of the model, leading to an efficient solution based on a decomposition of the underlying Markov process. The decomposition which we consider is *time scale decomposition*, **based on Courtois's** *near complete decomposability*. **This work has been influenced by related work on stochastic Petri nets: we will discuss the advantages and disadvantages of taking such an approach to the development of techniques for stochastic process algebras. Our technique is illustrated by an example based on a closed network of queues with finite capacity in which blocking may occur.**

1. INTRODUCTION

Like classical process algebras, stochastic process algebras (SPA) model systems as an interaction of autonomous *agents* or *components* who engage in *actions*. Unlike classical process algebras, in general SPA actions have an associated *duration*, which is characterised by an exponentially distributed random variable. Recent papers have shown the benefits of the structure within SPA models for both model construction (Götz *et al.*, 1993) and model simplification (Hillston, 1995). Unfortunately, however, these models suffer from problems of state space explosion. If SPA languages are to fulfil their early promise as performance modelling paradigms, efficient solution techniques must be found.

Given the clear component-based structure of the models it is perhaps natural to consider decomposition techniques. In particular, in this paper we look at the application of *time scale decomposition* to models which satisfy Courtois's *near complete decomposability* property (Courtois, 1977).

The rest of the paper is organised as follows. In Section 2 we introduce the notation and definition of the SPA which we will use in the remainder of the paper. After a brief introduction to the time scale decomposition approach to Markov process solution, Section 3 discusses the application of time scale decomposition to SPA models. This technique is illustrated by the example in Section 4. In Section 5 we explain the limitations of our current technique and how we would like to extend it. In addition, we discuss the advantages and disadvantages of basing SPA work on previous work on stochastic Petri nets. Finally, we present some conclusions and areas for future work in Section 6. Throughout the paper we will work in the notation of TIPP; however, the technique which we describe could be readily adapted for other stochastic process algebras, such as PEPA (Hillston, 1994).

2. NOTATION AND DEFINITIONS

In this section we present the notation which we will use throughout the remainder of the paper. We also introduce some definitions which will be useful in characterising models susceptible to time scale decomposition (TSD).

We assume a fixed set of action names $Act := Com \cup \{\tau\}$, where τ is a distinguished symbol for internal, invisible actions and Com is the set of visible activities (*communication* actions). Each action can either be exponentially distributed or passive. Exponentially distributed actions — denoted by the action's name and the rate (a, λ) — happen instantaneously after a duration that is exponentially distributed with rate λ. Passive actions (usually denoted by $(a, 1)$) describe *receptive* behaviour, i.e. the behaviour of a component which is waiting for a partner before completing the action. Other SPA use different notations for passive actions.

2.1. Syntax

The syntax of TIPP (*timed processes and performability evaluation*) is defined as follows (Hermanns & Rettelbach, 1994):

DEFINITION 2.1. *The set $\mathcal{L}$ of valid system descriptions of TIPP is given by the following grammar, where $a \in Act$, $\lambda \in I\!R^+$; $S \subseteq Com$, $X \in Var$, Var is a set of process variables.*

$$\begin{array}{lll} P & ::= & Stop \mid (a,\lambda).P \mid P+P \mid \\ & & P\|_S P \mid P\backslash S \mid rec\ X : P \mid X \end{array}$$

In addition to the recursion operator *rec*, we will use *defining equations* in order to model recursion. We assume that there is a countable set of *constants*, whose meaning is given by a defining equation such as $A := P$, i.e. the constant A has the behaviour of the agent P.

DEFINITION 2.2. *$Sort(P)$ denotes the set of action names corresponding to the actions which process P may engage in:*

$$\begin{array}{lll} Sort(Stop) & = & \emptyset \\ Sort((a,\lambda).P) & = & \{a\} \cup Sort(P) \\ Sort(P\,\|_S Q) & = & Sort(P) \cup Sort(Q) \\ Sort(rec\ X : P) & = & Sort(P) \\ Sort(P\ +\ Q) & = & Sort(P) \cup Sort(Q) \\ Sort(P\backslash S) & = & Sort(P) \setminus S \end{array}$$

This is termed the action sort *of P.*

Note that the set of actions which P will actually engage in is a subset of $Sort(P)$ because the synchronization may disable the execution of some actions.

2.2. Semantics and equivalences

The semantics of TIPP are given by a set of inference rules: they are presented here in Figure 1 without comment (Hermanns & Rettelbach, 1994). Using these rules each model can be mapped onto a *Markovian Labelled Transition System* (MLTS). This is transformed into the state transition diagram of the underlying CTMC by reducing parallel arcs and neglecting loops (Götz *et al.*, 1993).

We can see from the semantic rule for parallel composition that passive actions may be defined as those which are neutral with respect to this combinator. This explains the use of "1" here and the different notations adopted by other SPA languages, reflecting their different definitions of parallel composition.

DEFINITION 2.3. (REACHABILITY SET) *The* reachability set *$ds(P)$ of a given process P is defined as:*

$$ds(P) = \{P' \in \mathcal{L} \mid P \to^* P'\}$$

where $P \to^* P'$ denotes that the process state P' is derivable by applying a sequence of semantics rules to the process P.

$$\langle . \rangle \qquad \frac{}{(a,\lambda).P \xrightarrow{a,\lambda,\varepsilon} P}$$

$$\langle +_l \rangle \qquad \frac{P \xrightarrow{a,\lambda,w} P'}{P+Q \xrightarrow{a,\lambda,+_l.w} P'}$$

$$\langle +_r \rangle \qquad \frac{Q \xrightarrow{a,\lambda,w} Q'}{P+Q \xrightarrow{a,\lambda,+_r.w} Q'}$$

$$\langle \|_l \rangle \qquad \frac{P \xrightarrow{a,\lambda,w} P'}{P\|_S Q \xrightarrow{a,\lambda,\|_l.w} P'\|_S Q} \qquad (a \notin S)$$

$$\langle \|_r \rangle \qquad \frac{Q \xrightarrow{a,\lambda,w} Q'}{P\|_S Q \xrightarrow{a,\lambda,\|_r.w} P\|_S Q'} \qquad (a \notin S)$$

$$\langle \| \rangle \qquad \frac{P \xrightarrow{a,\lambda,v} P' \quad Q \xrightarrow{a,\mu,w} Q'}{P\|_S Q \xrightarrow{a,\lambda\mu,(v,w)} P'\|_S Q'} \qquad (a \in S)$$

$$\langle \backslash_{yes} \rangle \qquad \frac{P \xrightarrow{a,\lambda,w} P'}{P\backslash a \xrightarrow{\tau,\lambda,w} P'\backslash a}$$

$$\langle \backslash_{no} \rangle \qquad \frac{P \xrightarrow{b,\lambda,w} P'}{P\backslash a \xrightarrow{b,\lambda,w} P'\backslash a} \qquad (a \neq b)$$

$$\langle rec \rangle \qquad \frac{P\{(recX : P)/X\} \xrightarrow{a,\lambda,w} P'}{recX\ :\ P \xrightarrow{a,\lambda,w} P'}$$

FIGURE 1. Operational semantics of TIPP

Various notions of bisimulation equivalence have been defined for TIPP. In the context of this paper a pure functional bisimulation equivalence will be of some importance. This relation will be denoted by $\sim_F$ (Hermanns & Rettelbach, 1994). Additionally, a form of weak bisimulation with respect to the functional behaviour will be necessary. We adopt Milner's *weak bisimulation* which is denoted by $\approx$ (Milner, 1989).

2.3. Analysis of the underlying CTMC

Let $Q \in \mathbb{R}^{n\times n}$ denote the infinitesimal generator matrix of the CTMC underlying the MLTS of a given TIPP-process, where n is the number of reachable states. In general, for performance analysis, the following linear equation system has to be solved:

$$\Pi Q = 0 \quad \text{subject to} \quad \sum_{i=1}^{n} \Pi(i) = 1 \qquad (1)$$

where the unknown row vector Π is the steady state probability distribution of the underlying CTMC. Each entry in Π corresponds to a state/process in the MLTS. $\Pi(1)$ is the probability for the initial process P. The values in this steady state probability distribution vector may be used to compute high-level performance or dependability measures.

Unfortunately SPA models suffer from problems of state space explosion and solving the equations (1) may exceed the capabilities of contemporary computers. This suggests the use of decompositional approaches for finding Π. In this case we replace the solution of the single system of equations (1) by a set of solutions of simpler (sub)systems. This set of solutions is then combined using an aggregated version of the original system. This is the so-called *decomposition/aggregation* approach to the solution of CTMC, introduced in (Simon & Ando, 1961).

More formally, decomposition/aggregation (in the sequel referred to as D/A) involves four steps:

Decomposition Partition the state space of the system into k aggregates. Various criteria may be applied to determine which states constitute one partition.

Analyse aggregates separately Consider each aggregate as a separate system: disregard transitions between aggregates and construct the matrix $Q_{[i,i]}$ as the generator of the smaller CTMC, considering only transitions within the aggregate i. Compute its solution Π_i^*. Repeat for each partition $i \in \{1, \ldots, k\}$.

Construct and solve the aggregate CTMC Represent each aggregate as a single state and compute the transition rates between these states using

$$Q_{agg}(i, j) = (\Pi_i^* Q_{[i,j]}) \cdot 1$$

where $Q_{[i,j]}$ are the transitions from partition i to partition j. Then, solve $\Pi_{agg} Q_{agg} = 0$ subject to $\sum_{i=1}^{k} \Pi_{agg}(i) = 1$.

Disaggregate the final result The approximate steady state probability distribution Π is computed as

$$\Pi = (\Pi_{agg}(i) \Pi_i^*)_{i=1}^{k}$$

The quality of this approximation depends upon the structure of the original matrix and the partition chosen. Consequently classes of matrices, and corresponding Markov processes, which give rise to exact or good approximate solutions have been studied extensively. For example, a *completely decomposable* matrix consists of stochastic blocks down the principal diagonal and zeroes everywhere else. The aggregates can be solved separately as in the D/A approach and the solution will be exact*. However, a completely decomposable matrix cannot result from a SPA model with a single initial process.

A *nearly completely decomposable* matrix is one in which the blocks down the leading diagonal have elements which are at least an order of magnitude larger than any element outside these blocks (Courtois, 1977). Again each block represents a separate subsystem, and although they do interact (entries in the off-diagonal blocks) their internal transitions occur much more frequently. Such systems are known to have good approximate, or even exact, solutions when solved using D/A based on the block structure. This is the so-called *time scale decomposition*—aggregates are formed by grouping states which can be reached quickly relative to the transition rates to states in other blocks.

It is not very difficult to apply decomposition of the state space after a complete reachability analysis has been applied. Most of the iterative D/A algorithms require the storage of the complete state space in any case. However, if the decomposition can be applied at the syntactic level or at least during reachability analysis, the storage complexity of the numerical solution will be greatly reduced. The main problem is to find a partitioning scheme which can be applied at the syntactic level, and to generate one representative of each equivalence class.

In order to obtain reasonable results with TSD we will need to identify those models which satisfy the NCD property. In Section 3 we discuss a solution to this problem for a class of TIPP models which exhibit a particular structure. This structure, although restrictive, can be readily checked from the definition of the model. Once such a model has been identified the time scale decomposition can be carried out automatically without the need to ever construct and store the whole state space. This is the major contribution of this paper.

2.4. Static TIPP models

We are interested in those TIPP models which give rise to ergodic Markov processes, and in particular those which have a time scale decomposition which coincides with the component structure of the model. This paper identifies a class of models which satisfy those conditions. Initially we restrict the class of models we consider to ensure ergodicity.

For a model to be ergodic it must be able to repeat any behaviour which it carries out; this means that after every choice it must be possible to return to the agent and make the choice again, possibly with a different outcome. Consequently we consider models which are constructed at the highest level as the synchronisation of agents which are constructed using prefix and choice. For technical reasons in this paper we do not allow hiding to be included in a model. Also, because we want the component structure of the model to be static during evolution we only allow recursion over the lower level components, not over parallel combinations of components.

This leads us to formally define the syntax of *static* TIPP expressions in terms of *sequential components* Q and *model components* P:

$$\begin{array}{lll} P & ::= & Q \mid P \|_S P \\ Q & ::= & (\alpha, r).Q \mid Q + Q \mid rec\ X : Q \mid X \mid Stop \end{array}$$

Definition 2.4. *Let $\mathcal{L}_{seq} \subset \mathcal{L}$ be the set of all sequential processes. Then, the set of static processes*

*There is no aggregate CTMC in this case as there are no transitions between aggregates.

with n components is defined as follows:

$$\mathcal{L}^n_{par} = \{P \in \mathcal{L} \mid \quad P \equiv P_1 \|_{S_1} P_2 \|_{S_2} \dots P_m \\ \wedge P_i \in \mathcal{L}^{k_i}_{par} \;\wedge\; \textstyle\sum_{i=1}^m k_i = n\}$$

for arbitrary $S_i \subseteq Com$ where $\mathcal{L}^1_{par} = \mathcal{L}_{seq}$.

In general the "states" of a TIPP model are the syntactic forms that it will exhibit during its evolution. For static models the number of sequential components, and the general structure of the syntactic terms, will be the same in all states of the system. Thus we can consider an alternative representation of the state, as a tuple of states, representing the current state of each of the sequential components.

DEFINITION 2.5. (STATE VECTOR) *Let P be a static component comprising sequential components $Q_1, Q_2, \dots Q_K$. Then a* state vector *of the model component P as derivative P_i is the vector $(Q_{1_i}, Q_{2_i}, \dots, Q_{K_i})$ where $Q_{k_i}, 1 \leq k \leq K$ is the current derivative of Q_k in P_i.*

DEFINITION 2.6. (REDUNDANCY) *A sequential component Q_k is said to be* redundant *within the state vector representation of a static component P if Q_k is a sequential component of P and for all derivatives P_i of P, given the current derivatives of the other sequential components $Q_{j_i}, j \neq k$, the current derivative of Q_k, Q_{k_i} can be inferred.*

If a sequential component is shown to be redundant within the state vector, a *reduced state vector* may be formed in which the derivatives of this component have been eliminated.

It will sometimes be convenient to be able to define whether a given sequential component occurs within a static agent: the partial order, $\prec$, over components, captures the notion of *being a subcomponent*:

DEFINITION 2.7. (SUBCOMPONENTS)

1. $R \prec P$ *if* $R \in ds(P)$
2. $R \prec P + Q$ *if* $R \prec P \vee R \prec Q$
3. $R \prec P \|_S Q$ *if* $R \prec P \vee R \prec Q$
4. $R \prec A$ *if* $A := P \wedge R \prec P$

The *interface* of a sequential component within a static model is then defined to be the union of all the cooperation sets whose scope includes the component R.

DEFINITION 2.8. (INTERFACE) *For any sequential component R within a model component C (i.e. $R \prec C$) the* interface *of R within C, denoted $\mathcal{I}(C :: R)$, is the set of action types on which R is required to cooperate:*

1. $\mathcal{I}(R :: R) = \emptyset$
2. $$\mathcal{I}(P \|_L Q :: R) = \begin{cases} \mathcal{I}(P :: R) \cup \mathcal{I}(Q :: R) \cup L & \text{if } R \prec P \wedge R \prec Q \\ \mathcal{I}(P :: R) \cup L & \text{if } R \prec P \wedge R \not\prec Q \\ \mathcal{I}(Q :: R) \cup L & \text{if } R \prec Q \wedge R \not\prec P \\ \emptyset & \text{otherwise.} \end{cases}$$

Note that the interface of a component may be larger than the action sort of the component. Sometimes we will be interested in only the subset of the interface over which a component is active; this is termed the *active* interface.

DEFINITION 2.9. (ACTIVE INTERFACE) *The* active interface *of a sequential component R within a model C, denoted $\mathcal{I}_A(C :: R)$, is the set of actions within the interface of R which R can engage in:*

$$\mathcal{I}_A(C :: R) = \mathcal{I}(C :: R) \cap Sort(R)$$

3. TIME SCALE DECOMPOSITION OF SPA MODELS

Time scale decomposition is one of the most widely practised decomposition techniques. As explained earlier, it is based on decomposing a CTMC so that short term equilibrium is reached within single partitions, and partition changes occur only rarely as the process approaches its long term equilibrium (Simon & Ando, 1961).

The practical application of TSD requires a priori knowledge of the structure of the state space, since the complete state space is often too large to permit an efficient decomposition. Instead an approach is needed which allows aggregates to be formed without first constructing the whole state space. High level formalisms, such as Queueing Networks, GSPNs, SANs, or SPAs provide a means to do this systematically and a large body of work has already been published on this problem in the context of Queueing Networks and Stochastic Petri Nets, e.g. (Ammar & Islam, 1989; Blakemore & Tripathi, 1993; Conway & Georganas, 1989; Courtois, 1977; Couvillion *et al.*, 1991). The inspiration for this paper was the TSD algorithm of Blakemore and Tripathi, which we briefly outline below.

3.1. Previous work on TSD algorithms for SPN models

In (Ammar & Islam, 1989) the authors propose a method for applying TSD to SPN models. This involves classifying the SPN transitions as either *slow* or *fast* according to some threshold firing rate. Slow transitions are temporarily removed from the model, and fast subnets are evaluated for various different initial markings. These markings are found via an aggregated SPN which includes one place for each disconnected fast subnet, as well as the slow transitions. Some technical details of the approach are difficult to formalise and therefore restrict the potential for automation.

In a more recent paper (Blakemore & Tripathi, 1993), an alternative approach with the particular aim of automating TSD for SPN is examined. In particular the authors show that classifying the transitions as fast and slow is not sufficient to specify the desired decomposition. They adopt an alternative approach in which a marking dependent integer valued function Φ is used to partition the reachability graph. Φ defines an equivalence relation over the state space—those states which have the same value under Φ fall within the same partition. It is assumed that Φ can be expressed as a linear combination of the number of tokens in each place, with integer coefficients, i.e. it is characterised by a vector $\boldsymbol{A}$. It is the modeller's responsibility to provide Φ or $\boldsymbol{A}$.

From this point the algorithm can proceed automatically. Transitions which change the value of Φ are termed *cross transitions* and these are initially disabled. This means that only the part of the state space corresponding to the *current* value of Φ needs to be stored at any time. Considering each of the cross transitions in turn a list of possible other aggregates is formed before details of the state space are removed. No aggregated SPN is constructed but using the information about the aggregates and the cross transitions the aggregated CTMC is formed and solved.

Both approaches require that the subnets considered separately give rise to irreducible Markov chains, or chains which have a particular form of absorbing states. We will impose similar restrictions on the SPA method.

3.2. TSD algorithm for SPA models

The algorithm for TSD in SPA models which we present was influenced most strongly by that of Blakemore and Tripathi in one important way—the decomposition is carried out *structurally* and not at the CTMC level nor directly in terms of fast and slow actions. As explained above this avoids the need to store the complete state space at once. However, unlike their algorithm, ours does not require any input from the modeller, and can be applied completely automatically. In the SPN case it was necessary to manually define the function Φ, via the vector $\boldsymbol{A}$. In the SPA model we can take advantage of the compositional structure to identify states which are equivalent in the time scale sense.

There are several ways to decompose a process term in the context of TSD. The approach we take is to identify which actions are *slow* and restrict the model so that it can no longer carry out these actions. This is simply achieved by synchronizing the model with the *Stop* process over the set of all such slow actions. However this is not sufficient if we are to automate the algorithm—we still need a method to characterise when a state will belong to a particular aggregate of the model. It is for this purpose that Blakemore and Tripathi use the function Φ. Our solution is more straightforward but relies on a further restriction on the class of models that we consider.

3.2.1. *Fast-Slow processes*

We assume that our process $P \in \mathcal{L}^n_{par}$ is comprised of the sequential components P_i. Moreover we assume that there is some value t, $t \in I\!R$, such that all actions with activity rate r, $r < t$, will be classed as *slow actions*, while actions with activity rate r, $r \geq t$, will be classed as *fast actions*. Let $slow(P) \subseteq Sort(P)$ denote the set of all slow actions in the model P and $\overline{slow}(P) = Sort(P) - slow(P)$. We assume that these sets are well-defined, i.e. that no action exhibits both fast and slow instantiations in the same model. If necessary an action can be renamed to distinguish these different cases. This is shown in an example in Section 5.2.

Based on this classification of actions we classify the time scale behaviour of the subprocesses P_i. P_i is a *slow subprocess* if it enables only slow actions; P_i is a *fast subprocess* if it enables only fast or passive actions; all other subprocesses are *hybrid subprocesses*. For the initial explanation of the algorithm we consider models $P \in \mathcal{L}^n_{par}$ which consist of only fast and slow subprocesses with at least one P_i being a slow subprocess. We will call such models *fast-slow processes*. The case of models which include hybrid subprocesses will be discussed in Section 3.3.

3.2.2. *The algorithm*

We consider models P, comprising of fast sequential processes $F_1, F_2, \ldots, F_k$ and slow sequential processes $S_1, S_2, \ldots, S_\ell$, i.e. in state vector representation $P \equiv (F_1, \ldots, F_k, S_1, \ldots, S_\ell)$. The aggregates we produce are based on sequences of fast activities which may occur between slow activities. This is achieved, as suggested above, by considering P synchronised with the *Stop* process over $slow(P)$. Thus each aggregate will be a set of state vectors which all exhibit the same derivatives for each of the slow sequential processes, although the derivatives exhibited by the fast sequential processes may vary:

$$A_{[S_1,\ldots,S_l]} \equiv \{(F'_1,\ldots,F'_k,S'_1,\ldots,S'_\ell) \mid S'_1 \equiv S_1,\cdots,S'_\ell \equiv S_\ell\}$$

Using the definition of P to establish the first aggregate, we use the standard SOS rules to find all the reachable states of $P \parallel_{slow(P)} Stop$. To find further aggregates, we apply the expansion law to each state in this aggregate in turn, adding it to the list of aggregates if the partial state vector $(S''_1, \ldots, S''_\ell)$ differs from those already in the list. This allows us to find other aggregates and construct the aggregated CTMC. We repeat this process until no new aggregates are added to the list and all aggregates have been expanded. Note that no aggregated SPA model is constructed, only the aggregated CTMC.

We are now ready to present the algorithm applying TSD to SPA models. The structure of the algorithm depicted in Figure 2 closely resembles the structure of the algorithm in (Blakemore & Tripathi, 1993). It is important to note that we must assume that each aggregate contains a single recurrence class. This ensures that a

1) — *PHASE I. analyze each aggregate in isolation*
2) $\mathcal{U} := \{P_{start}\}$ — *set of representatives for unexplored aggregates*
3) $\mathcal{V} := \emptyset$ — *set of explored aggregate indices*
4) **while** $\mathcal{U} \neq \emptyset$ **loop**
5) $P_{rep} := head(\mathcal{U})$
6) $i := index(P_{rep})$ — *index of current aggregate*
7) $\mathcal{U} := \mathcal{U} - \{P_{rep}\}$
8) $\mathcal{V} := \mathcal{V} \cup \{i\}$
9) $R := P_{rep} \parallel_{slow(P)} Stop$ — *disable cross actions*
10) $Q_i :=$ CTMC derived from $ds(R)$
11) compute $\Pi_i Q_i = 0$ subject to $\sum_j \Pi_i(j) = 1$
12)
13) — *for each state in the current aggregate*
14) **foreach** $P \parallel_{slow(P)} Stop \in ds(R)$
15) **foreach** $(a, \lambda, P') \in expand(P)$ — *for all successor states of* P
16) $i' = index(P')$
17) **if** $i' = i$ **then continue** — P' *is in the same aggregate*
18) **end if**
19) **if** $Q_{agg}(i, i') = 0 \wedge i \notin \mathcal{V}$ **then** — *a leads to a new aggregate*
20) $\mathcal{U} := \mathcal{U} \cup \{P'\}$ — *add representative for new aggregate*
21) **end if**
22) $Q_{agg}(i, i') := Q_{agg}(i, i') + \lambda \Pi_i(P)$
23) **end loop**
24) **end loop**
25) free storage for $ds(R), Q_i$
26) **end loop**
27)
28) — *PHASE II. analyze the aggregate CTMC*
29) compute $\Pi_{agg} Q_{agg} = 0$ s.t. $\sum_j \Pi_{agg}(j) = 1$
30)
31) — *PHASE III. compute final result via disaggregation*
32) **foreach** $i \in \mathcal{V}$ **loop** — *for each aggregate* i
33) $\Pi_i := \Pi_i \Pi_{agg}(i)$
34) **end loop**

FIGURE 2. SPA time scale decomposition algorithm

single process representative of each aggregate will be sufficient. This implicit assumption of the algorithm is discussed in more detail in Section 5.

Analogously to the SPN work the algorithm is divided into three phases. In phase I all aggregates are analyzed in isolation and the generator matrix Q_{agg} of the aggregate CTMC is constructed. Phase II analyzes the aggregate CTMC and phase III carries out the disaggregation to compute the steady state probability distribution over the whole model.

The function *index* that is being used in the above algorithm extracts the partial state vector characterizing an aggregate from a given static component. It returns an unambiguous index for the aggregate, that is needed in order to define the aggregate matrix Q_{agg}. The function *expand* applied to a process P returns a list of tuples (a, λ, P') containing the action names, action rates, and successor states of P. Finally, in line 22 we denote with $\Pi_i(P)$ the probability of state P within the aggregate i.

3.3. Decomposition of hybrid subprocesses

So far we have excluded models which include *hybrid* subprocesses. Recall that these are sequential processes which can perform fast and slow activities, or passive and slow activities, or fast, slow and passive activities. First we will discuss why such subprocesses present a problem to the algorithm, before going on to discuss how this problem can be overcome.

Consider a model $P \equiv (F_1, \ldots, F_k, H, S_1, \ldots, S_\ell)$ where H is a hybrid subprocess. If we consider H to be a fast subprocess an aggregate $A_{[S'_1, \ldots, S'_\ell]}$ may become blocked by the synchronisation with *Stop* over slow or passive activities in H. In contrast if we consider H to be a slow subprocess the mechanism of synchronising with *Stop* over slow activities will not necessarily prevent H from changing derivative, thus making it impossible to easily detect when we have changed aggregate, say from $A_{[H,S_1,\ldots,S_\ell]}$ to $A_{[H',S_1,\ldots,S_\ell]}$ by $H \xrightarrow{(fast,r)} H'$. One solution to this problem would be to combine $Sort(H)$ with $slow(P)$ to form the set of activities synchronized with $Stop$. However this has the effect that transitions within the aggregate CTMC are not necessarily larger than the transitions within a single aggregate. Alternatively, in (Herzog & Mertsiotakis, 1994) the special case of hybrid subprocesses in which fast actions do not change the current derivative of H is treated in detail. Below we propose a more general solution.

Suppose that each hybrid subprocess is replaced by two subprocesses, one fast and one slow. The fast subprocess is formed by making the original subprocess passive with respect to its slow activities; otherwise the structure of this subprocess remains unchanged. The slow subprocess mirrors the behaviour of the original subprocess with respect to its slow activities but disregards any fast or passive activities. When we consider these two subprocesses synchronised over the set of slow activities it will be isomorphic to the original subprocess. Note the new slow subprocess will be redundant in the sense of Definition 2.4. The following algorithmic skeleton implements this approach:

1. Let $H \in \mathcal{L}_{seq}$ be a hybrid subprocess in $P \in \mathcal{L}^n_{par}$.
2. Find a process H_F which is isomorphic to H except that all actions in $slow(H)$ are now passive.
3. Find a process $H_S \approx H\backslash\overline{slow}(H)$ where $\approx$ is Milner's *weak bisimulation*; assign the rates of activities in H to the corresponding activities in H_S.
4. Form P' by removing H and replacing it by $H_F \parallel_{slow(H)} H_S$; increment n.
5. Repeat until there is no remaining hybrid subprocess.

4. MODELLING STUDIES

In order to evaluate the efficiency and the applicability of our implementation of TSD we applied this technique to several models. The latter aspect is discussed in the next section. Here, the first aspect of our studies, the efficiency, is discussed based on experiments we made on a simple model of a LAN that is well known from many textbooks on performance evaluation. We describe a network of workstations and resources connected via a single bus (see Fig. 3).
The specifications of the various model components look as follows:

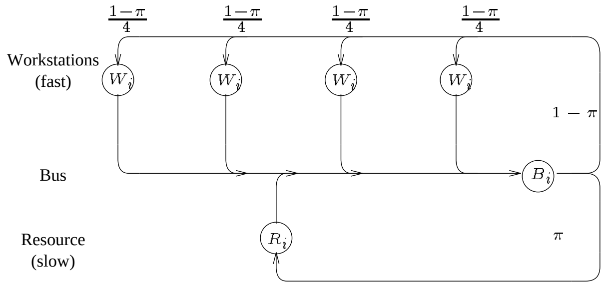

FIGURE 3. Structure of the model

$$
\begin{array}{lcl}
W_0 & := & (w_{in},(1-\pi)/4).W_1 \\
W_i & := & (w_{in},(1-\pi)/4).W_{i+1} \;+\; (w_{out},\lambda).W_{i-1} \\
W_k & := & (w_{out},\lambda).W_{k-1} \\
R_0 & := & (r_{in},\pi).R_1 \\
R_i & := & (r_{in},\pi).R_{i+1} \;+\; (r_{out},\mu).R_{i-1} \\
R_l & := & (r_{out},\mu).R_{l-1} \\
B_0 & := & (w_{out},1).B_1 \;+\; (r_{out},1).B_1 \\
B_i & := & (w_{out},1).B_{i+1} \;+\; (r_{out},1).B_{i+1} \;+ \\
 & & (w_{in},\nu).B_{i-1} \;+\; (r_{in},\nu).B_{i-1} \\
B_m & := & (w_{in},\nu).B_{m-1} \;+\; (r_{in},\nu).B_{m-1}
\end{array}
$$

The complete specification is comprised by

$$(R_0 \parallel W_2 \parallel W_2 \parallel W_2 \parallel W_2) \parallel_{\{w_{in},w_{out},r_{in},r_{out}\}} B_0$$

We analyzed this model with both, exact numerical analysis as well as TSD. Exact analysis was based on complete reachability analysis and subsequent numerical solution of the balance equations using a Gauß-Seidel iteration scheme. Our TSD implementation relies mainly on the algorithm sketched in Fig. 2 using the same Gauß-Seidel method to solve the huge single aggregates. Smaller aggregates as well as the aggregate matrix are solved with LU-factorization. We assume that the resource component is the only slow one.

Since the TSD-algorithm needs additional computational overhead in order to avoid complete reachability analysis, it is obvious that it will be slower for systems with well-ballanced transition rates. To check up to which point this holds, we varied the ratio between fast and slow action rates and analyzed a model with both methods. The required runtimes dependent on the logarithmic ratio between fast and slow transition rates are shown in Fig. 4.

We can see clearly that the runtime of exact analysis increases rapidly if the degree of coupling gets smaller and after a value of 2 (fast action rates are 10^2 times larger) TSD can outperform it already. This is due to the fact that TSD is robust against loosely coupled matrices, since the different aggregates to be solved are within the same time range and therefore the number of iterations can be kept small.

The other diagram in Fig. 4 compares the runtimes for model solution dependent on the state space size.

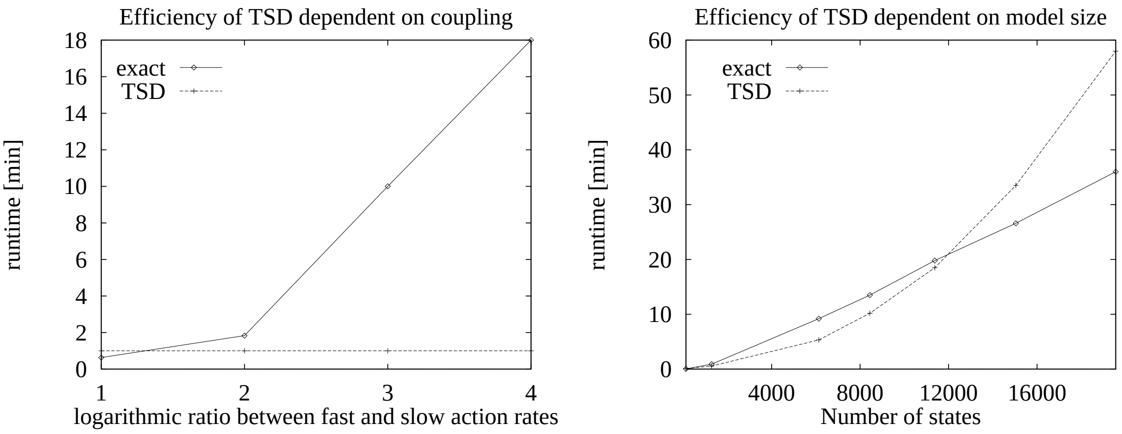

FIGURE 4. Comparison of runtimes

The presented runtimes include also the time needed for state space exploration. We chose a degree of coupling for which the runtime for each method is almost the same and increased the state space size by adding more jobs into the system. Upto a number of 14 jobs (11376 states) the runtime for TSD remains lower than that of exact analysis. However, this changes for larger state spaces. The reason for this bad runtime behaviour may have several sources, but the most important one is probably that the examination for neighbouring aggregates makes it necessary to expand a process state twice. The first time with the disabling, the second time without disabling.

5. DISCUSSION

5.1. Restrictions of the algorithm

In this section we discuss some of the problems which may arise when TSD is applied to TIPP processes. Even processes exhibiting the correct fast-slow structure may reveal problems when the algorithm is applied. These problems are due to an implicit assumption of the algorithm that each aggregate will constitute a *single recurrence class*. In other words, even when considered in isolation, each aggregate remains strongly connected when we consider only fast transitions. This restriction results from the second decomposition/aggregation step, where steady state analysis is applied to each aggregate. If this condition is violated two types of problems can occur:

Reducible Aggregates: The state space of an aggregate may be connected but contain transient states and more than one recurrence class. In this situation steady state analysis will not produce a unique solution. Depending on the representative of the equivalence class which is chosen as the initial state for reachability analysis, different solutions may be obtained.

Undetected States: The state space of an aggregate may be disconnected with separate recurrence classes. In this case some states will remain completely undetected depending on which recurrence class the representative of the equivalence class belongs to.

There are at least two possible solutions to these problems. Firstly, if there is more than one slow process, the decomposition scheme can be changed by regarding one of the slow processes as a fast process. This will have the effect of altering the structure of the aggregates. Secondly, if this is not possible, or if it is ineffectual, the aggregates must be arbitrarily joined, or split into smaller pieces. This latter approach would be difficult to automate. However an even larger problem would be to recognise the problem of reducible aggregates when it occurs: the algorithm will select a single representative of the equivalence class and solve the aggregate on the basis of that representative. It will have no way of recognising that a different representative could have given rise to a different answer. Being able to identify from the form of the fast subprocesses that all aggregates will be strongly connected and limiting the application of the algorithm to these cases would be a more feasible solution.

The following simple example shows a model which exhibits the fast-slow structure but which will have undetected states if the algorithm is applied directly.

Example:

$$\begin{aligned}
Proc &:= (arrival, \lambda).Busy + (fail, 1).Proc' \\
Proc' &:= (repair, 1).Proc \\
Busy &:= (service, \mu).Proc + (fail, 1).Busy' \\
Busy' &:= (repair, 1).Busy
\end{aligned}$$

$$\begin{aligned}
Up &:= (fail, \delta).Down \\
Down &:= (repair, \beta).Up \\
System &:= Proc \,\|_{\{fail, repair\}}\, Up
\end{aligned}$$

Let us suppose that $\delta, \beta << \lambda, \mu$. Consequently, Up is the only slow subprocess and we can decompose according to which state of Up is current. Accordingly, we would get two aggregates since the process Up has got only two states (see Figure 5). However, we can easily see that there are two process states corresponding to the second aggregate ($Down$) but they are not linked together. This means that applying TSD would result in either $Proc' \|_A Down$ or $Busy' \|_A Down$ going undetected.

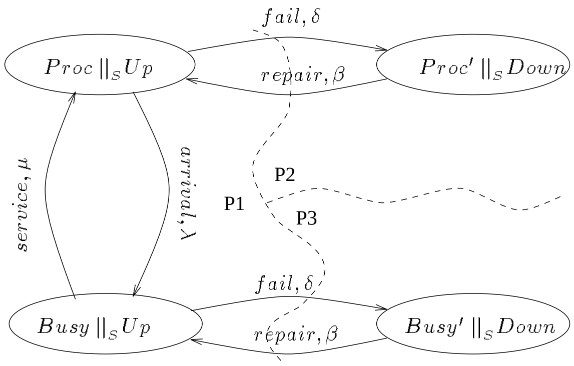

FIGURE 5. Example of a process with undetectable states

We can regard the first component ($Proc$) as a hybrid subprocess and we will now investigate the effect of decomposing by this component, instead of Up. Using the technique for handling hybrid processes by means of dummy processes, as outlined in Section 3.3, we add a redundant component Q in parallel with the original process. Q should be behaviourally equivalent to the original process and additionally it should be able to partition the state space into the three partitions P1, P2, and P3 (see Figure 5). On the first criterion the following is a candidate for Q

$$\begin{array}{rcl} Q_{up} & := & (fail, 1).Q_{down} + (fail, 1).Q'_{down} \\ Q_{down} & := & (repair, 1).Q_{up} \\ Q'_{down} & := & (repair', 1).Q_{up} \end{array}$$

However, this fails on the second criterion since there is no way to obtain three aggregates—it is impossible for Q to recognize into which state it should switch.

For this model the only way to make it suitable for the algorithm is to rename one of the fail actions in $Proc$ and modify Up accordingly:

$$\begin{array}{rcl} Up & := & (fail, \delta).Down + (fail', \delta).Down' \\ Down & := & (repair, \beta).Up \\ Down' & := & (repair', \beta).Up \end{array}$$

With this modification the model can be decomposed successfully. Unfortunately such a solution cannot, in general, be applied automatically.

In the following subsection we consider a class of models in which it is possible to assess automatically whether the problems of reducible aggregates and undetected states are likely to occur.

5.2. Gordon-Newell processes

We consider a class of processes that was motivated by the class of Gordon-Newell queueing networks (GNQN) (Gordon & Newell, 1967). The main characteristics of these networks are: no external arrivals, no departures, exponential services with state dependent service rates, FIFO queueing discipline, state independent routing, and one customer class. In general, GNQNs are assumed to be non-blocking (networks with blocking are referred as GNQN/B) and exhibit a product form distribution in equilibrium. When efficient solution techniques exist for such models decompositional approaches do not need to be considered. Although work is progressing on identifying SPA models with product form solution (Sereno, 1995; Harrison & Hillston, 1995) efficient algorithms to solve such models are yet to be established. Here we aim to demonstate the type of structural reasoning which may be used on models to determine when aggregates will fulfil the single recurrence class assumption.

Before we introduce a class of processes exhibiting a state space structure similar to that of GNQN/B we have to define what is a *Birth-Death* process, since these processes will be the main building blocks of *Gordon-Newell Processes* (GNP).

DEFINITION 5.1. *A process $P \in \mathcal{L}_{seq}$ is called a Birth-Death Process (BDP), iff*

1. $(\exists k, l)(P \backslash Sort(P) \sim_F Q^l_k)$ *where*

$$Q^l_k := \begin{cases} (\tau, 1).Q^l_{k+1} & if\, k = 0 \\ (\tau, 1).Q^l_{k-1} & if\, k = l \\ (\tau, 1).Q^l_{k-1} + (\tau, 1).Q^l_{k+1} & else \end{cases}$$

2. $(\exists \mathcal{R} \subseteq ds(P) \times ds(P))\,(\exists Arv \subset Sort(P))\,(\forall a \in Arv)$ $(P \xrightarrow{a,\cdot} P' \Rightarrow P \mathcal{R} P')$
3. $(\forall d \in Sort(P) \backslash Arv)\,(P \xrightarrow{d,\cdot} P' \Rightarrow P' \mathcal{R} P)$

We remind the reader that the eqivalence relation $\sim_F$ denotes a functional bisimulation equivalence (see Section 2.2). The action set Arv denotes actions that can be interpreted as *arrivals*. Consequently, the relation $\mathcal{R}$ defined on the states of a BDP is nothing else than a successor relation on the states of a BDP. Consider the following specification of a simple buffer as an example:

$$\begin{array}{rcl} Q_0 & := & (p, 1).Q_1 \\ Q_i & := & (p, 1).Q_{i+1} + (q, \lambda).Q_{i-1} \\ Q_n & := & (q, \lambda).Q_{n-1} \end{array}$$

The process Q_i receives custumers via action p and delivers them through action q with rate λ. This is a BDP with

$$Arv = \{p\}, \;\; \mathcal{R} = \{(Q_0, Q_1), (Q_1, Q_2), \ldots, (Q_{n-1}, Q_n)\}$$

In order to describe precisely conditions that have to be fulfilled by a parallel composition of several BDPs to yield a GNP, we need the following definitions that capture the relationship between different sequential components within a parallel process. The set of actions on which two components interact is defined as:

DEFINITION 5.2. (COMMON INTERFACE) *For any sequential components Q, R within a model component C (i.e. $Q \prec C \wedge R \prec C$) the* common interface *of Q and R within C, denoted $\mathcal{I}(C :: Q, R)$, is the set of action types on which Q is required to cooperate with R:*

1. $\mathcal{I}(S :: Q, R) = \emptyset$ *if* $S \in \mathcal{L}_{seq}$
2. $$\mathcal{I}(P_1 \,\|_L\, P_2 :: Q, R) = \begin{cases} \mathcal{I}(P_1 :: Q, R) \\ \qquad \text{if } Q \not\prec P_2 \wedge R \not\prec P_2 \\ \mathcal{I}(P_2 :: Q, R) \\ \qquad \text{if } Q \not\prec P_1 \wedge R \not\prec P_1 \\ \mathcal{I}(P_1 :: Q, R) \cup \\ \mathcal{I}(P_2 :: Q, R) \,\cup\, L \\ \qquad \textit{otherwise.} \end{cases}$$

As with the interface from definition 2.8, the common interface may also contain actions that are bound by other sequential components, as the following simple example demonstrates:
Example:

$$\begin{aligned} P &:= (P_a \,\|\, P_b) \,\|_{\{a\}}\, (P_a \,\|\, P_b) \\ P_a &:= (a, \lambda).Stop \\ P_b &:= (b, \lambda).Stop \end{aligned}$$

According to the definition, the common interface of P_a and P_b is

$$\mathcal{I}(P :: P_a, P_b) = \{a\}$$

even though P_b can never participate on action a. That is why we have to build the intersection with the action sorts of the affected components:

DEFINITION 5.3. (COMMON ACTIVE INTERFACE)

$$\mathcal{I}_A(C :: Q, R) = \mathcal{I}(C :: Q, R) \cap Sort(Q) \cap Sort(R)$$

Now we are ready to define Gordon-Newell Processes. We do not present necessary and sufficient conditions to yield a model class that is equivalent to Gordon-Newell queueing networks. However, it should be possible to translate such queueing networks into SPA-models that fall in this class.

DEFINITION 5.4. *A process $P \in \mathcal{L}^n_{par}$ is a Gordon-Newell Process (GNP), iff*

1. $(\forall i)(1 \leq i \leq n)(P_i$ *is a BDP*$)$
2. $(\forall i)(1 \leq i \leq n)(Sort(P_i) \subseteq \mathcal{I}_{\mathcal{A}}(P :: P_i))$
3. $(\forall i, j, k)(i < j < k \Rightarrow \mathcal{I}_{\mathcal{A}}(P :: P_i, P_j) \cap \mathcal{I}_{\mathcal{A}}(P :: P_j, P_k) \cap \mathcal{I}_{\mathcal{A}}(P :: P_k, P_i) = \emptyset)$

The first condition is obvious. The second condition ensures that no "customers" arrive from outside or leave the represented network, while the latter condition ensures that not more than two processes change their state at once. The following simple example shows a specification of a typical GNP (see Figure 6).

$$\begin{aligned} System &:= P_n \,\|_{\{p_{in}, q_{in}, r_{in}\}}\, (Q_0 \,\|\, R_0) \\ P_0 &:= (p_{in}, 1).P_1 \\ P_i &:= (p_{in}, 1).P_{i+1} \\ &\;\;+\;\; (q_{in}, \lambda).P_{i-1} + (r_{in}, \lambda).P_{i-1} \\ P_n &:= (q_{in}, \lambda).P_{n-1} + (r_{in}, \lambda).P_{n-1} \\ Q_0 &:= (q_{in}, 0.5).Q_1 \\ Q_i &:= (q_{in}, 0.5).Q_{i+1} + (p_{in}, \mu).Q_{i-1} \\ Q_n &:= (p_{in}, \mu).Q_{n-1} \\ R_0 &:= (r_{in}, 0.5).R_1 \\ R_i &:= (r_{in}, 0.5).R_{i+1} + (p_{in}, \nu).R_{i-1} \\ R_n &:= (p_{in}, \nu).R_{n-1} \end{aligned}$$

This example shows a GNP with three subprocesses. It should be emphasized that P delivers customers to Q and R by performing the actions q_{in} and r_{in} respectively, although using only one action would be sufficient, too. This, however, would have as a consequence that the model would not be analyzable with TSD, since decomposing by an arbitrary subprocess would disable the insertion of customers to both, Q **and** R, regardless of on which subprocess we decomposed. The reason for this is that our form of global disabling is not able to deal with the case that one action is both fast and slow. An alternative form of disabling, namely local disabling by replacing Q or R with *Stop* would be a solution. However, for the sake of simplicity we retain the restriction that an action cannot be fast and slow.

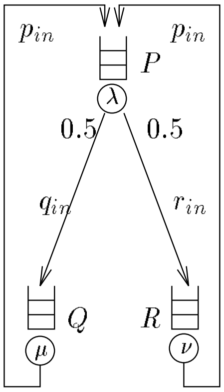

FIGURE 6. GNP with 3 processes

5.3. Non-blocking Gordon-Newell processes

In the case of GNP without blocking it is possible to characterize partitioning schemes that always fulfill the single recurrence class assumption. For this we need a more precise terminology for the structure of a GNP.

DEFINITION 5.5. *The relation $SG(P) \subset \mathcal{L}_{seq} \times \mathcal{L}_{seq}$* (structure graph) *for a given parallel process $P \in \mathcal{L}^n_{par}$ is defined as follows:*

1. $(S_i, S_j) \in SG(P)$ iff $\mathcal{I}_{\mathcal{A}}(P :: S_i, S_j) = A \neq \emptyset \quad \wedge$
2. $(\exists a \in A)(\exists s_i \in ds(S_i))(\exists s_j \in ds(S_j))$
$(s_i \xrightarrow{a,\cdot} s_i' \wedge s_j \xrightarrow{a,\cdot} s_j' \wedge s_i' \mathcal{R} s_i \wedge s_j \mathcal{R} s_j')$

According to the definition, the structure graph for the example of the last subsection would be

$$SG(P) = \{(P_n, Q_0), (P_n, R_0), (Q_0, P_n), (R_0, P_n)\}$$

Now we are able to establish the notion of GNPs which are most amenable to solution using TSD.

Definition 5.6. *A Gordon-Newell process P is* S-decomposable, *iff $\exists P_k \prec P$ where P_k is a slow subprocess within P that fulfills the condition*

$$\{(S_i, S_j) \in SG(P) \mid S_i \neq P_k \neq S_j\}$$

is strongly connected.

From the above definitions we can deduce the following proposition:

Proposition 5.1. *Be P an S-decomposable GNP without blocking. Then, P may be analyzed with TSD guaranteeing that the single recurrence class assumption is met.*

Assuming that we decompose by a subprocess P_i that fulfills the above condition, for every aggregate the "flow of customers" between the remaining subprocesses cannot be prohibited by P_i since there are connections to bypass P_i.

If we take a look at our simple example again, we can see that it is indeed an S-decomposable process, provided either μ or ν is clearly smaller than the other rates. Both Q and R satisfy the condition defined in Def. 5.6.

Unfortunately, the blocking case is much more difficult to handle. Although in many cases TSD works fine also for GNP/B, it is not easy to generalize the above proposition.

5.4. Adapting SPN techniques to SPA

The algorithm presented in Figure 2 is closely related to that presented for SPN in (Blakemore & Tripathi, 1993). Indeed the work presented in this paper arose from an attempt to adapt Blakemore and Tripathi's TSD technique for SPN to SPA. In this section we will consider the merits of taking such an approach to the development of techniques for SPA.

As pointed out in (Donatelli *et al.*, 1995) there are many similarities between SPN and SPA models, particularly with respect to the generation and solution of the underlying CTMC. However, since SPN and GSPN form a more mature paradigm there are many more efficient algorithms available and it seems natural to follow these established techniques, rather than develop new ones for SPA models. While the current work endorses this view to some extent, the authors would like to stress that the benefits were derived from following the SPN work *in spirit* rather than *in detail.*

The algorithm of Blakemore and Tripathi was reliant on the function Φ defined in terms of the vector $\boldsymbol{A}$. As explained in Section 3 this function is a linear combination of the number of tokens in each place in the net. In the SPA model there is no corresponding quantifiable information readily accessible in each state. Attempts were made to use the derivatives of the sequential subprocesses in a similar manner but this seemed to imply that the modeller must have intimate knowledge of the complete state space of the system.

The current solution, the identification of fast and slow subprocesses, is attractive as it shows that the time scale structure coincides with the physical structure of the model. However it is quite removed from Blakemore and Tripathi's Φ. It would not be possible for a similar approach to be taken in SPN models, as in that case no physical structure is apparent within the model. Thus we conclude that researchers should not try too hard to identify similarities between the formalisms but rather to capitalise upon their individual features.

6. CONCLUSION

We have presented an approach to Time Scale Decomposition for a simple class of SPA models which can be fully automated. Moreover, in Section 3.3 we have suggested how this class can be extended to include all models in the wider class $\mathcal{L}^n_{par}$. However, full development of this procedure will rely on establishing formal and efficient methods for finding the slow version, H_S, of a hybrid subprocess H. This is one area for future work. Another is to extend the considered class of models to include those which involve the hiding operator—this will introduce the problem of decomposing over τ actions. Our decomposition algorithm currently relies on synchronisation with the *Stop* process to disable slow actions. Since τ actions cannot be synchronised this approach would not be possible with such actions.

SPA models are prone to problems of state space explosion. However we have demonstrated that the compositional structure inherent in these models may be successfully exploited to develop efficient solution techniques which avoid these problems. Further work is needed to develop a suite of such techniques and syntactic characterisation of the SPA models susceptible to them.

Acknowledgements

The authors Jane Hillston and Vassilis Mertsiotakis are supported by the EPSRC and the SFB182 respectively. This cooperation was made possible by the BC/DAAD. Furthermore, the authors would like to thank Alex Blakemore and Satish Tripathi for stimulating this work and Stephen Gilmore and Matteo Sereno for their valuable help in finalising the paper.

References

Ammar, H. H., & Islam, S. M. Rezaul. 1989. Time Scale Decomposition of a Class of Generalized Stochastic Petri Net Models. *IEEE Transactions on Software Engineering*, **15**(6), 809–820.

Blakemore, A., & Tripathi, S. 1993. Automated Time Scale Decomposition of SPNs. *In: Proc. of 5th Int. Workshop on Petri Nets and Performance Models (PNPM '93).*

Conway, A.E., & Georganas, N.D. 1989. *Queueing Networks – Exact Computational Algorithms.* MIT Press.

Courtois, P.J. 1977. *Decomposability: Queueing and Computer System Applications.* Academic Press, New York.

Couvillion, J., Freire, R., Johnson, R., Obal, W. D., Qureshi, A., Rai, M., Sanders, W. H., & Tvedt, J. E. 1991. Performability Modeling with UltraSAN. *IEEE Software*, **8**(5), 69–80.

Donatelli, S., Hillston, J., & Ribaudo, M. 1995. A Comparison of Performance Evaluation Process Algebra and Generalized Stochastic Petri Nets. *In: Proc. of 6th Int. Workshop on Petri Nets and Performance Models (PNPM '95).* IEEE CS-Press.

Gordon, W.J., & Newell, G.F. 1967. Closed Queueing Systems with Exponential Servers. *Operations Research*, **15**, 254–265.

Götz, N., Herzog, U., & Rettelbach, M. 1993. Multiprocessor and Distributed System Design: The Integration of Functional Specification and Performance Analysis Using Stochastic Process Algebras. *In: Proc. of the 16th Int. Symposium on Computer Performance Modelling, Measurement and Evaluation, PERFORMANCE '93.* Springer.

Harrison, P., & Hillston, J. 1995. Exploiting Quasi-reversible Structures to Find Product From Solutions in Markovian Process Algebra Models. *The Computer Journal*, **this issue**.

Hermanns, H., & Rettelbach, M. 1994 (July). Syntax, Semantics, Equivalences, and Axioms for MTIPP. *Pages 71–88 of:* Herzog, U., & Rettelbach, M. (eds), *Proc. of the 2nd Workshop on Process Algebras and Performance Modelling.*

Herzog, U., & Mertsiotakis, V. 1994 (July). Applying Stochastic Process Algebras to Failure Modelling. *Pages 107–126 of:* Herzog, U., & Rettelbach, M. (eds), *Proc. of the 2nd Workshop on Process Algebras and Performance Modelling.*

Hillston, J. 1994. *A Compositional Approach to Performance Modelling.* Ph.D. thesis, University of Edinburgh.

Hillston, J. 1995. Compositional Markovian Modelling Using a Process Algebra. *In:* Steward, W.J. (ed), *Numerical Solution of Markov Chains.* Kluwer.

Milner, R. 1989. *Communication and Concurrency.* London: Prentice Hall.

Sereno, M. 1995. Towards a Product Form Solution for Stochastic Process Algebras. *The Computer Journal*, **this issue**.

Simon, H.A., & Ando, A. 1961. Aggregation of Variables in Dynamic Systems. *Econometrica*, **29**, 111–138.