Available online at www.sciencedirect.com

ScienceDirect

ELSEVIER

Electronic Notes in Theoretical Computer Science 190 (2007) 167–183

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

A Structural Approach for Modelling Performance of Systems Using Skeletons

Gagarine Yaikhom[1,2] Murray Cole[1,3] Stephen Gilmore[1,4]
Jane Hillston[1,5]

School of Informatics, University of Edinburgh,
Edinburgh - EH9 3JZ, United Kingdom

Abstract

In this paper, we discuss a structural approach to automatic performance modelling of skeleton based applications. This uses a synthesis of performance evaluation process algebra (PEPA) and a pattern-oriented hierarchical expression scheme. Such approaches are important in parallel and distributed systems where the performance models must be updated regularly based on the current state of the resources.

Keywords: Performance evaluation, Patterns, Algorithmic Skeletons.

1 Introduction

Designing a system using high-level constructs has clear advantages. This has long been recognised in structured parallel and distributed programming where (often sequential) sub-tasks are structured for parallel assembly [1][2][3][4]. A more recent example is the BPEL language, which structures a composition of Web Services into an orchestration in which simpler services are aggregated into a composite [5]. The technical agenda behind these styles of description is to have a high-level, concise description of the structure of the computation which can be readily re-shaped in order to find a good mapping of tasks onto computing resources. The programmer is

[1] This paper is supported by the EPSRC project ENHANCE (“Enhancing the Performance Predictability of Grid Applications with Patterns and Process Algebras” (GR/S21717/01)) and the EU FET-IST Global Computing 2 project SENSORIA (”Software Engineering for Service-Oriented Overlay Computers” (IST-3-016004-IP-09)). Jane Hillston is also supported by EPSRC Advanced Research Fellowship EP/c543696/01.

[2] Email: gyaikhom@inf.ed.ac.uk

[3] Email: mic@inf.ed.ac.uk

[4] Email: stg@inf.ed.ac.uk

[5] Email: jeh@inf.ed.ac.uk

1571-0661/$ – see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.entcs.2007.07.010

concerned with achieving a suitable throughput of jobs while satisfying constraints on the utilisation of components such as servers and other execution environments.

However, the composition languages which allow sub-tasks to be composed do not usually provide a mechanism for assessing whether or not the new version is likely to improve on the performance of the previous one. Furthermore, languages suitable for performance modelling such as stochastic process algebras are not usually structured in our sense, and do not have linguistic apparatus to express sub-task composition. Our aim in this paper is to bridge this gap by automatically generating process algebra models from structured application descriptions and thus to allow designers to compile their applications into process algebra models suitable for performance evaluation via steady-state or transient analysis or verification via probabilistic model-checking. In the present paper we focus on the results which can be obtained by Markovian steady-state analysis of the process algebra model. We use algorithmic skeletons [6] as an exemplar of a structured composition language and use Performance Evaluation Process Algebra (PEPA) [7] as our process algebra. Our examples are drawn from the domain of structured parallel and distributed programming.

2 Background

In this section we provide brief descriptions of the stochastic process algebra used and the approach to structured parallel and distributed programming based on algorithmic skeletons. For further details the reader should consult [7] and [6][8].

2.1 Performance Evaluation Process Algebra (PEPA)

PEPA is a Markovian process algebra, in which an exponentially distributed random variable, representing duration, is associated with each activity. As in all process algebras, models are constructed from *components* which interact via *activities*. The syntax of the language is as follows:

$$S ::= (\alpha, r).S \mid S + S \mid C_S \qquad \text{(prefix, choice and component name)}$$

$$P ::= P \underset{L}{\bowtie} P \mid P/L \mid C \qquad \text{(parallel, hiding and component name)}$$

Here S denotes a *sequential component* and P denotes a *model component* which executes in parallel. C stands for a constant which denotes either a sequential component or a model component as introduced by a definition. C_S stands for constants which denote sequential components. The prefix $(\alpha, r).S$ gives a component a designated first activity: it will have action type α and duration governed by the exponential distribution with parameter r. The choice operator $(+)$ enables the activities of its two operands. The first activity to complete distinguishes one of them: the other is discarded. The system will behave as the derivative resulting from the evolution of the chosen component. Structure is introduced into the model using the cooperation combinator ($\underset{L}{\bowtie}$). There are no complementary actions in PEPA and this captures a CSP-style parallel composition: components synchronize on those actions in the cooperation set L which are enabled by both components.

This synchronisation respects the notion of *bounded capacity*, meaning that a component cannot be made to go faster by synchronisation. Thus the duration of the shared activity is governed by the minimum of the random variables associated with each of the contributing activities. In some cases a component is *passive* with respect to a synchronised activity, meaning that it will participate at whatever rate its partner component expects. This is denoted by the distinguished symbol $\top$, i.e. $(\alpha, \top)$. When the cooperation set L is empty we use the shorthand notation $\|$. Finally, there is an abstraction operator, hiding, denoted P/L, which allows the type of an activity whose type is in L to be replaced by the distinguished type τ, which denotes a private or hidden activity.

The language definition is expressed in [7] via a small-step structured operational semantics which maps a PEPA model onto a Continuous-Time Markov Chain (CTMC). The CTMC can then be analysed for both steady state and transient performance measures, using standard techniques.

2.2 The notion of algorithmic skeletons

The skeletal approach to the design of parallel programming systems proposes that the complexity of parallel programming be contained by restricting the mechanisms through which parallelism can be introduced to a small number of architecture independent constructs, originally known as "algorithmic skeletons". Each skeleton specification captures the logical behaviour of a commonly occurring pattern of parallel computation, while packaging and hiding the details of its concrete implementation. Provision of a skeleton-based programming methodology simultaneously raises the level of abstraction at which the programmer operates and provides the scheduling system with crucial information on the temporal and structural interaction patterns exhibited by the application. Responsibility for exploiting this information passes from programmer to compiler and/or run-time system. To obtain such detailed information from an equivalent *ad-hoc* message passing program is impossible in the general case. In this paper, we show how the structural information can be used to construct PEPA performance models of the application.

3 Expressing structured applications with skeletons

To automate generation of performance models, a given application must be first expressed in a form which captures its essence. In this paper, we adopt a pattern-oriented approach, which is based on the notion of algorithmic skeletons—a system that was designed to enrich, and simplify, structural development of distributed and parallel applications. It must be understood that although the introduced constructs could be used directly by a human performance modeller, they are meant to be used as an internal interface for generating performance models automatically from distributed skeleton-based applications. To facilitate a thorough treatment of the automation, we will focus on the following three basic skeletons. [6]

[6] Extensions to these basic forms are available in the tool: `http://groups.inf.ed.ac.uk/enhance/`.

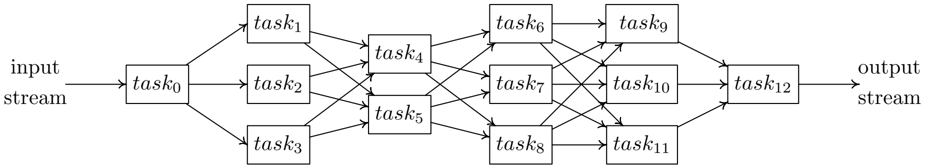


Fig. 1. Dataflow diagram of an application with nested pipelines and task replications.

Pipeline skeleton A pipeline skeleton arranges a set of components sequentially, so that data units entering the pipeline are processed in each of these components in turn (in the order the components are specified) before the final result leaves the pipeline. In our approach we will use the following construct to specify a pipeline:

`pipe`(<number of components>);

The components contained within a skeleton construct could be either skeleton components (which results in hierarchical nesting), or task components (where the data units are processed). In the latter case, a task component is specified with the following construct:

`task`(<component name>, <rate>);

Here <rate> is the rate at which each of the data units entering the task component is processed—used while modelling the task's computational performance.

Deal skeleton A deal skeleton replicates a given task component in parallel, so that data units entering the deal could be processed by one of the replicated components, before the final result leaves the deal. The task component which receives a given data unit is chosen by a round-robin data distribution policy. We use the following construct to specify a deal:

`deal`(<number of replications>, <component name>, <rate>);

Farm skeleton A farm skeleton is similar to the deal: a given task component is replicated in parallel so that data units entering the farm are processed by one of the components. The difference, however, is that the process of choosing the task component that should receive a given data unit is unpredictable, being dynamically demand-driven upon completion of earlier computations. The non-determinism is therefore probabilistic, where the next data unit is sent to one of the succeeding tasks that has completed processing the data units assigned to it previously. We use the following construct to specify a farm:

`farm`(<number of replications>, <component name>, <rate>);

We shall now illustrate the usage of these constructs by expressing a concrete example. Imagine a system similar to the one shown in Fig. 1. Here, we have a six stage[7] pipeline at the highest level. Some of these stages are task components (for example, stages 1 and 6), while some are hierarchical skeleton nestings (say, for example, stage 2 is a Farm, while stage 3 is a Deal).

[7] The components of a pipeline are frequently referred to as "stages".

The skeleton-based expression of the system is shown on the right. This description is hierarchical—each subtree is described depth-first, left-to-right across the same subtree level. We proceed to the next subtree in the same level only after all the previous subtrees have been described completely; i.e. there are no skeleton nestings with insufficient task assignments.

```
pipe(6);
task("task", 1.0);
farm(3, "task", 3.0);
deal(2, "task", 2.0);
farm(3, "task", 3.0);
deal(3, "task", 3.0);
task("task", 1.0);
```

It may be easier to view the whole exercise as a step-wise refinement of the system description, where we conceive the system at the highest level and proceed with refinements until the lowest level descriptions consist of task components only.

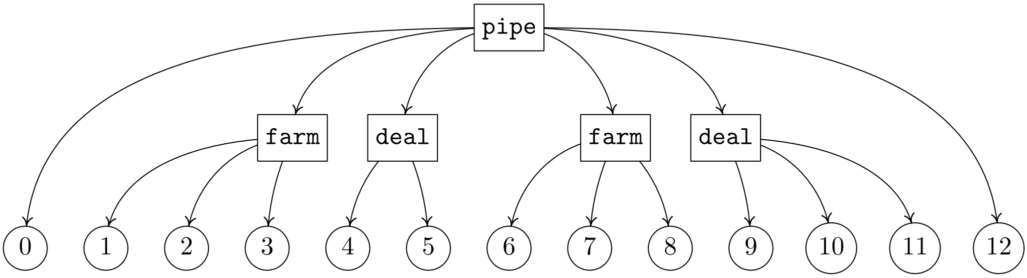


Fig. 2. The skeleton hierarchy tree which corresponds to the example system shown in Fig. 1.

In light of the discussions to follow, it would be prudent to mention here that, for every structured application, a hierarchical tree data structure is maintained by the model generation tool, internally. We shall refer to this data structure as the *skeleton hierarchy tree* (shown in Fig. 2 for our example system).

The skeleton hierarchy tree encapsulates most of the information provided in the description (the overall structural and component details of the system). Additional information is derived from this tree automatically, when needed (for example, the data dependency graph connecting the tasks). We shall now discuss these in detail.

4 Generation of performance models

Generation of PEPA performance models, from a given description of a structured application, can be divided into three phases. In the first phase, the directed acyclic graph (which represents data dependency between task components) is derived from the skeleton hierarchy tree. This graph is then used in the following phases. In the second phase, the process definitions for each of the task components are determined. Finally, in the third phase, the overall system is modelled by combining the task components, and skeletal components, based on their hierarchical organisation. The final phase is important because it completes the performance model by establishing the synchronisation sets, which will be used by the model solver while synchronising task components at different levels of composition.

Algorithm 1 *GS*(*node*): Generate source lists from the skeleton hierarchy tree

```
$n := node.nchildren$ // Number of children nodes
$node.slist := parent.slist$ // Inherit source list from parent node
$node.stype := parent.stype$ // Inherit source pattern type
for $i := 0$ to $(n-1)$ do
        $GS(node.child_i)$ // Recursively generate children source lists
if $node.type$ is task then // Node is a task component
        $v := \{x$ : where $x = node.index\}$
        if $parent.type$ is deal or farm then // Parent is a replicable skeleton
                $temp_{node} := v$
        else
                $parent.slist := v$
                $parent.stype :=$ pipe // Update source pattern type
else if $node.type$ is pipe then // Node is a pipeline skeleton
        if $parent.type$ deal or farm then // Pipeline within replicable
                $temp_{node} := node.slist$
        else
                $parent.slist := node.slist$
                $parent.stype := node.type$ // Update source pattern type
else if $node.type$ deal or farm then // Node is a replicable skeleton
        $m := \bigcup_{i=0}^{n-1} temp_i$ // Merge temporary lists on children
        if $parent.type$ deal or farm then // Replicable within replicable
                $temp_{node} := m$
        else
                $parent.slist := m$
                $parent.stype := node.type$ // Update source pattern type
```

4.1 Determination of the directed acyclic graph

Let the directed acyclic graph $\mathcal{G}(\mathcal{T},\mathcal{E})$, where $\mathcal{T}$ is the set of task components and $\mathcal{E}$ is the set of directed edges connecting task components, represent the data dependency graph which corresponds to the skeleton hierarchy tree. To derive such a graph from a given skeleton hierarchy tree, we use recursive preorder tree traversal algorithms, described as follows:

To every task in $\mathcal{T}$, assign a unique index i, where $0 \leq i < |\mathcal{T}|$. We will use the notation t_i to mean: "task component with index i," or sometimes, "task i". To concretely implement the graph $\mathcal{G}$, associate with every task, t_i, two ordered sets of task indices: (1) the source list $\mathcal{S}_i$, which gives the set of tasks in $\mathcal{T}$ from which task i can receive data; (2) the destination list $\mathcal{D}_i$, which gives the set of tasks to which task i can send data. They are formally defined as follows:

$$\mathcal{S}_i = \{j : (t_j, t_i) \in \mathcal{E}, i \neq j\} \text{ and } \mathcal{D}_i = \{j : (t_i, t_j) \in \mathcal{E}, i \neq j\}.$$

When it is clear from the context which task we are referring to, we may choose to drop the subscripts in $\mathcal{S}_i$ and $\mathcal{D}_i$. It is important to note here that these sets only list all the possible predecessors (or successors) of a task—the effective set with which the task eventually communicates is determined from these sets by applying the corresponding source (or destination) *manner of interaction*, which we will be discussing shortly. For example, given a pipeline containing two consecutive deals, the general case is that all the tasks in the first deal will be in the source list for the second deal. However, if the deals are of the same size, then in fact, a task in the second deal will only ever receive data from the task in the first deal which has the same intra-deal sibling rank (the effective set is therefore a *singleton* set).

It can be observed that these two sets, in combination with the task set $\mathcal{T}$, completely define the directed acyclic graph $\mathcal{G}$. We therefore use recursive tree traversal algorithms to generate these sets from the skeleton hierarchy tree. In Algorithm 1, we show the process by which the source lists are derived from the skeleton hierarchy tree. In this algorithm, each node maintains the following data: a pointer to its parent node in the skeleton hierarchy tree, *parent*; the number of its children nodes, *nchildren*; the list of task indices, *slist*, that is available to this node as sources (in fact, this is what the algorithm will determine); the source pattern type, *stype*; the type of node, *type* (this could be a skeleton node or a task node, see Fig. 2). In addition to these, each node also maintains a temporary list, *temp*, which is used by its parent node while finalising the parent's source list. Note here that for some skeleton nodes where replication of tasks are involved, the source list on these nodes cannot be finalised until all of its child subtrees have also finalised their source lists. A similar algorithm is used to derive the destination lists.

4.2 Modelling the task components

In general, a task component in a structured system is a process which repeatedly undergoes the transitions: *receive* → *compute* → *send*. Depending on the higher-level structure containing the task, these three basic activities are specialised accordingly. In some cases, for example, some of these activities are skipped (as in producer tasks, which does not perform receive activities; or consumer tasks, where send activities are never performed).

As noted in Section 4.1, when a task communicates with other tasks, the tasks with which the communications are performed are based on an effective subset of $\mathcal{S}$ (or $\mathcal{D}$), determined by the manner of interaction. This manner of interaction is based on the skeleton preceding (or succeeding) this task. It is easier to define this manner of interaction as a *function* over the source (or destination) list, which chooses task indices from the corresponding list, thus establishing the effective subset for the current communication. In essence, this function therefore outlines for each task how the task should interact with the remaining tasks in the skeleton hierarchy tree: the source function defining how data should be received; the destination function, how data should be sent.

The manner of interaction for a given task corresponds to the location of the task within the skeleton hierarchy tree. As we can see in Algorithm 1, the source pattern type (`stype`), is set with respect to the skeleton components containing the task; the destination pattern type is set similarly. If we respectively represent

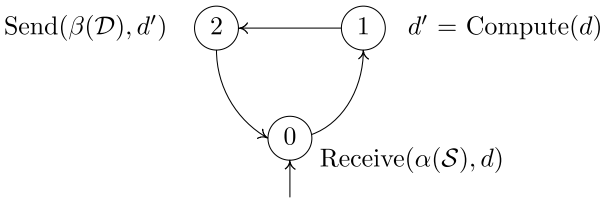

Send($\beta(\mathcal{D}), d'$)
2
1
$d' = \text{Compute}(d)$
0
Receive($\alpha(\mathcal{S}), d$)

Fig. 3. The transition diagram which gives the sequence of generic activities that are performed by any one of the tasks in the system. During model generation, these activities are specialised based on the manner in which the task interacts with its predecessors and successors.

the source and destination interaction functions with α and β, we can summarise a task as shown in Fig. 3. From this abstract representation, it is clear that the PEPA process definition of a task component is determined by the subsets $\alpha(\mathcal{S})$ and $\beta(\mathcal{D})$; and the relationship between α and β, as required by the transition *receive* → *compute* → *send*. Since tasks can have different α and β, we have to determine process definition templates for all the possible pattern combinations. Let us represent, for brevity, such combinations with $\{\alpha(\mathcal{S}) \rightarrow t_i \rightarrow \beta(\mathcal{D})\}$; meaning, "task i receives data based on the source function α; and sends data based on the destination function β". When either of the functions are not defined (as discussed at the beginning of this section), we represent this with a $*$ (as in $\{* \rightarrow t_i \rightarrow \beta(\mathcal{D})\}$ for a producer task).

Furthermore, since enumerating all the cases can be quite involving, we shall condense the case investigations further by making some observations on the relationship between the different interaction functions (based on the definition of the skeleton constructs, see Section 3). These observations are: (a) the *Deal* interaction function is a special case of the *Farm*, where probabilistic non-determinism in the *Farm* is removed by enforcing a round-robin data distribution policy. (b) the *Pipeline* interaction function is a special case of the *Deal*, where the source (or destination) list is a *singleton* set. Based on the later observation, discussion of cases involving the *Pipeline* will be ignored, since it is covered in the cases with *Deal*. We will, however, cover the combinations of *Deal* and *Farm* interaction functions.

Case $\{* \rightarrow t_i \rightarrow \mathit{Deal}(\mathcal{D})\}$ In this case, t_i is a producer task. This task produces data units, which are then sent to one of the tasks in $\mathcal{D}$, chosen according to the round-robin policy. The corresponding PEPA process definition, where $n = |\mathcal{D}|$ and λ_i is the computational activity associated with t_i, is expressed as follows:

$$t_i \overset{\mathit{def}}{=} (\lambda_i, \top).(\mathit{move}_{i0}, \top).(\lambda_i, \top).(\mathit{move}_{i1}, \top). \cdots .(\lambda_i, \top).(\mathit{move}_{i(n-1)}, \top).t_i;$$

Here move_{ij} represents communication of data from task i to the jth task in $\mathcal{D}$. We choose this notation instead of, say send_{ij}, because these activities will be used again later when we define the synchronisation sets. If the latter notation was adopted, we are required to define a system for matching up corresponding send_{ij} and $\mathit{receive}_{ji}$ pairs (which is, in fact, unnecessary).

Case $\{\mathit{Deal}(\mathcal{S}) \rightarrow t_i \rightarrow *\}$ In this case, t_i is a consumer task. The task receives data units from one of the tasks in $\mathcal{S}$, which it then consumes. By following a notation similar to the previous case, we have the following process definition:

$$t_i \overset{def}{=} (move_{0i}, \top).(\lambda_i, \top).(move_{1i}, \top).(\lambda_i, \top).\cdots.(move_{(n-1)i}, \top).(\lambda_i, \top).t_i;$$

Here $n = |\mathcal{S}|$, the number of task indices in the source list. Based on arguments similar to the one used in the previous case, we use $move_{ji}$ to represent communication of data from the jth task in $\mathcal{S}$ to task i.

Case $\{Deal(\mathcal{S}) \to t_i \to Deal(\mathcal{D})\}$ In this case, t_i is an intermediate task: data units received from one of the tasks in $\mathcal{S}$ is processed, and the result is sent to one of the tasks in $\mathcal{D}$. In each instance of the send and receive communications, the effective task is chosen independently based on the round-robin distribution policy.

When the cardinalities of the source and destination lists are the same, $p = |\mathcal{S}| = |\mathcal{D}|$, the process definition is simple, as shown below:

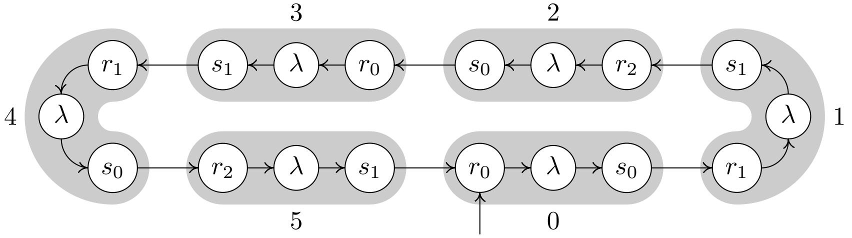
$$\begin{aligned} t_i \overset{def}{=} &(move_{0i}, \top).(\lambda_i, \top).(move_{i0}, \top).\\ &(move_{1i}, \top).(\lambda_i, \top).(move_{i1}, \top).\\ &\ldots\\ &(move_{(p-1)i}, \top).(\lambda_i, \top).(move_{i(p-1)}, \top).t_i; \end{aligned}$$

When $|\mathcal{S}| \neq |\mathcal{D}|$, however, there exists no immediate correspondence between the source and destination tasks. It is therefore necessary to resolve this mismatch until we find a repeatable sequence of activities. If we define *periodicity*, p, as the number of distinct *receive* → *compute* → *send* transitions after which the repetition ensues, it is easy to see that the periodicity is the *least common multiple* of $|\mathcal{S}|$ and $|\mathcal{D}|$. Based on this, we have the following process definition:

$$\begin{aligned} t_i \overset{def}{=} &(move_{xi}, \top).(\lambda_i, \top).(move_{iy}, \top).\\ &\cdots(\text{repeat } p \text{ times, incrementing } k \text{ in every iteration}).t_i; \end{aligned}$$

where $0 \leq k < p$, $x = k \mod |\mathcal{S}|$ and $y = k \mod |\mathcal{D}|$.

When $|\mathcal{S}| = 3$ and $|\mathcal{D}| = 2$, for example, the steady-state activity sequence is



which gives the following process definition:

$$\begin{aligned} t_i \overset{def}{=} &(move_{0i}, \top).(\lambda_i, \top).(move_{i0}, \top).(move_{1i}, \top).(\lambda_i, \top).(move_{i1}, \top).\\ &(move_{2i}, \top).(\lambda_i, \top).(move_{i0}, \top).(move_{0i}, \top).(\lambda_i, \top).(move_{i1}, \top).\\ &(move_{1i}, \top).(\lambda_i, \top).(move_{i0}, \top).(move_{2i}, \top).(\lambda_i, \top).(move_{i1}, \top).t_i; \end{aligned}$$

The above three cases can be used to completely define task components with any combination of *Pipeline* and *Deal*. We shall now extend this by introducing cases which account for the non-determinism associated with a *Farm* skeleton.

Case $\{* \rightarrow t_i \rightarrow Farm(\mathcal{D})\}$ In this case, the task is a producer. The main difference, however, is the non-determinism, which we capture with the *choice*(+) operator of PEPA. This is shown in the following process definition:

$$t_i \overset{def}{=} (\lambda_i, \top).t_i';$$
$$t_i' \overset{def}{=} (move_{i0}, \top).t_i + (move_{i1}, \top).t_i + \cdots + (move_{i(n-1)}, \top).t_i;$$

where $n = |\mathcal{D}|$. After a data unit has been produced, it is sent to any one of the tasks in $\mathcal{D}$, which brings back the task to the data production state.

Case $\{Farm(\mathcal{S}) \rightarrow t_i \rightarrow *\}$ In this case, the task is a consumer. The process definition is similar to the previous case, as shown in the following definition:

$$t_i \overset{def}{=} (move_{0i}, \top).t_i' + (move_{1i}, \top).t_i' + \cdots + (move_{(n-1)i}, \top).t_i';$$
$$t_i' \overset{def}{=} (\lambda_i, \top).t_i;$$

where $n = |\mathcal{S}|$. After a data unit has been received from any one of the tasks in $\mathcal{S}$, it is consumed; consequently bringing the task back to the receiving state.

Case $\{Farm(\mathcal{S}) \rightarrow t_i \rightarrow Farm(\mathcal{D})\}$ In this case, t_i is an intermediate task. The process definition for such tasks can be achieved by combining the previous two cases, as shown in the following:

$$t_i \overset{def}{=} (move_{0i}, \top).(\lambda_i, \top).t_i' + \cdots + (move_{(x-1)i}, \top).(\lambda_i, \top).t_i';$$
$$t_i' \overset{def}{=} (move_{i0}, \top).t_i + (move_{i1}, \top).t_i + \cdots + (move_{i(y-1)}, \top).t_i;$$

where, $x = |\mathcal{S}|$ and $y = |\mathcal{D}|$. Data units are received from any one of the tasks in $\mathcal{S}$, processed, and the result sent to any one of the tasks in $\mathcal{D}$.

Case $\{Deal(\mathcal{S}) \rightarrow t_i \rightarrow Farm(\mathcal{D})\}$ In this case, t_i is an intermediate task. What is unique about this task is that for every data received in round-robin fashion, the processed result is sent probabilistically to one of the tasks that has completed processing the job assigned to it previously. It is important to note here that since the source function preceding this task is a *Deal*, task i must receive the next data according to round-robin fashion, as it can be inferred that the tasks in $\mathcal{S}$ received their data in round-robin fashion. Once the data has been received and processed, the result is sent to a *Farm*; hence, a choice composition is used while dispatching the results, as shown in the following process definition:

$$t_i \overset{def}{=} (move_{0i}, \top).(\lambda_i, \top).t_i^0;$$
$$t_i^0 \overset{def}{=} (move_{i0}, \top).t_i^1 + (move_{i1}, \top).t_i^1 + \cdots + (move_{i(y-1)}, \top).t_i^1;$$
$$\cdots$$
$$t_i^{x-1} \overset{def}{=} (move_{(x-1)i}, \top).(\lambda_i, \top).t_i^x;$$
$$t_i^x \overset{def}{=} (move_{i0}, \top).t_i + (move_{i1}, \top).t_i + \cdots + (move_{i(y-1)}, \top).t_i;$$

where $x = |\mathcal{S}| - 1$ and $y = |\mathcal{D}|$.

Case $\{Farm(\mathcal{S}) \rightarrow t_i \rightarrow Deal(\mathcal{D})\}$ This case is similar to the previous, except for the reversal in the placement of the choice composition. We therefore have the following process definition:

$$t_i \overset{def}{=} (move_{0i}, \top).t_i^0 + (move_{1i}, \top).t_i^0 + \cdots + (move_{(x-1)i}, \top).t_i^0;$$
$$t_i^0 \overset{def}{=} (\lambda_i, \top).(move_{i0}, \top).t_i^1;$$
$$\ldots$$
$$t_i^{y-1} \overset{def}{=} (move_{0i}, \top).t_i^y + (move_{1i}, \top).t_i^y + \cdots + (move_{(x-1)i}, \top).t_i^y;$$
$$t_i^y \overset{def}{=} (\lambda_i, \top).(move_{iy}, \top).t_i;$$

where $x = |\mathcal{S}|$ and $y = |\mathcal{D}| - 1$.

In Algorithm 2, we incorporate all of the above cases in order to generate the process definition for each of the tasks in the skeleton hierarchy tree. For simplicity of representation, a case number is assigned to each of these cases, as shown in the table on the righthand side. Here, the columns list destination patterns; whereas, the rows enumerate source patterns.

	*	Pipe	Deal	Farm
*	0	1	2	3
Pipe	4	5	6	7
Deal	8	9	10	11
Farm	12	13	14	15

Furthermore, in Algorithm 2, the expression $\mathcal{S}(j)$ gives the mth task index in $\mathcal{S}$, where $m = j \mod |\mathcal{S}|$; the corresponding expression for the destination list, $\mathcal{D}(j)$, is defined similarly. We use the interface `Output:` to emit segments of the generated process definition. For every invocation to this interface, all the characters immediately following this, until the end of line, are emitted as part of the process definition. Also note that the order in which `Output:` is invoked is significant to the validity of the generated process definition.

To generate all the process definitions of all the tasks in the skeleton hierarchy tree, we traverse the hierarchy tree and invoke Algorithm 2 for all the nodes which is a `task` node. Once this is done, we have completed the second phase of performance model generation. We therefore proceed with the final phase where we define the synchronisation sets. Before we proceed, it will be worth recalling that the $move_{ij}$ and $move_{ji}$ activities, which correspond to the communications between tasks, will be used while defining these synchronisation sets.

4.3 Modelling the system

All the process definitions generated at the end of the second phase only model the performance of each task component, independently of the others. Since the structured application is a cooperative manifestation of these tasks, they must be synchronised accordingly with respect to the level of hierarchical composition. This is done in the final phase of model generation, which we shall now discuss.

At every level of the skeleton hierarchy tree, each subtree corresponds to a closed sub-system where only the boundary task components on either side interacts with their adjacent sibling subtrees. The task components which are inside this sub-system (the intermediate components) are synchronised with other task components within the same sub-system—there is no cross-boundary synchronisation. Hence, the final phase of model generation proceeds by defining synchronisation sets between adjacent sub-trees in each level of the hierarchy tree; which are refined

Algorithm 2 $GP(i)$: Generate process definition for task i

```
Output: $t_i \stackrel{def}{=}$
$l$ :=lcm($|\mathcal{S}|, |\mathcal{D}|$) // Least common multiple, or sum if either is zero
if case is 1 or 2 then // Predecessor $*$, successor Pipe or Deal
    for $0 \le j < l$ do Output: $(\lambda_i, \top).(move_{i,\mathcal{D}(j)}, \top).$
else if case is 4 or 8 then // Predecessor Pipe or Deal, successor $*$
    for $0 \le j < l$ do Output: $(move_{\mathcal{S}(j),i}, \top).(\lambda_i, \top).$
else if case is 5, 6, 9 or 10 then // Predecessor and successor Pipe or Deal
    for $0 \le j < l$ do Output: $(move_{\mathcal{S}(j),i}, \top).(\lambda_i, \top).(move_{i,\mathcal{D}(j)}, \top).$
else if case is 3 then // Predecessor $*$, successor Farm
    Output: $(\lambda_i, \top).t'_i; t'_i \stackrel{def}{=} (move_{i,\mathcal{D}(0)}, \top).t_i$
    for $1 \le j < |\mathcal{D}| - 1$ do
        Output: $+(move_{i,\mathcal{D}(j)}, \top).$
        if $(|\mathcal{D}| > 1) \wedge (j < |\mathcal{D}| - 1)$ then Output: $t_i$
else if case is 12 then // Predecessor Farm, successor $*$
    Output: $(move_{\mathcal{S}(0),i}, \top).t'_i$
    for $1 \le j < |\mathcal{S}|$ do
        Output: $+(move_{\mathcal{S}(j),i}, \top).t'_i$
    Output: $; t'_i \stackrel{def}{=} (\lambda_i, \top).$
else if case is 7 or 11 then // Predecessor Pipe or Deal, successor Farm
    for $0 \le j < |\mathcal{S}|$ do
        Output: $(move_{\mathcal{S}(j),i}, \top).(\lambda_i, \top).t^j_i; t^j_i \stackrel{def}{=} (move_{i,\mathcal{D}(0)}, \top).$
        if $j < |\mathcal{S}| - 1$ then Output: $t^{j+1}_i$
        else Output: $t_i$
        for $1 \le k < |\mathcal{D}|$ do
            Output: $+(move_{i,\mathcal{D}(k)}, \top).$
            if $j < |\mathcal{S}| - 1$ then Output: $t^{j+1}_i$
            else if $j < |\mathcal{D}| - 1$ then Output: $t_i$
else if case is 13 or 14 then // Predecessor Farm, successor Pipe or Deal
    for $0 \le j < |\mathcal{D}| - 1$, initialise $x = 0$ and increment by 2 in each step do
        Output: $(move_{\mathcal{S}(0),i}, \top).t^x_i$
        for $1 \le k < |\mathcal{S}|$ do
            Output: $+(move_{\mathcal{S}(k),i}, \top).t^x_i$
        Output: $; t^x_i \stackrel{def}{=} (\lambda_i, \top).(move_{i,\mathcal{D}(j)}, \top).$
        if $j < |\mathcal{D}| - 1$ then Output: $t^{x+1}_i; t^{x+1}_i \stackrel{def}{=}$
else if case is 15 then // Both predecessor and successor Farm
    Output: $(move_{\mathcal{S}(0),i}, \top).(\lambda_i, \top).t'_i$
    for $1 \le j < |\mathcal{S}|$ do
        Output: $+(move_{\mathcal{S}(j),i}, \top).(\lambda_i, \top).t'_i$
    Output: $; t'_i \stackrel{def}{=} (move_{i,\mathcal{D}(0)}, \top).$
    for $1 \le j < |\mathcal{D}|$ do
        Output: $t_i + (move_{i,\mathcal{D}(j)}, \top).$
Output: $t_i$;
```

Algorithm 3 $GM(node, nchild)$: Generate system model

```
$i := node.index$
$r := node.rank$ // Node rank among siblings
if $node.type$ is task then
    Output: $t_i$
    if $r < nchild - 1$ then
        if $parent.type \neq$ (deal or farm) then
            Output $\bowtie_L$ where $L = \{move_{i,j} : j = \mathcal{D}_i(k), 0 \leq k < |\mathcal{D}_i|\}$
        else
            Output: $||$
else
    Output: (
    for $0 \leq j < node.nchild$ do
        $GM(child_j, node.nchild)$// Recursively model subtree
    Output: )
    if $r < nchild - 1$ then
        if $parent.type \neq$ (deal or farm) then
            Output $\bowtie_L$ where
            $L = \{move_{x,y} : y = \mathcal{D}_i(j), x = \mathcal{S}_y(k), 0 \leq j < |\mathcal{D}_i|, 0 \leq k < |\mathcal{S}_y|\}$
        else
            Output: $||$
```

repeatedly until all the task components are synchronised.

We use Algorithm 3 to perform this final phase. In this algorithm, we use depth-first preorder tree traversal again. Since the synchronisation set between two subtrees can be expressed with respect to one of these subtrees, we choose a *forward* expression approach where the synchronisation set for a subtree is determined after the sub-system which corresponds to that subtree has been synchronised. We can see this in the algorithm: whenever the node is a task, we emit that task, and generate the synchronisation sets with which this task synchronises with all its successor tasks; when the node is a skeleton component, we generate the synchronisation set by accounting for the tasks on the "send" boundary of this sub-system, which interacts with the tasks on the "receive" boundary of the succeeding sub-tree.

Introducing computation and communication rates

The model which we have generated so far is incomplete in two ways. Although we have the task definitions and the structure of their interactions, both computation and communication rates are passive. Since active rates are necessary while performing synchronisation, we complete the model by introducing the relevant active rates to the model.

Since the model generation method developed here is aimed primarily towards distributed applications, the following discussion will focus on this context. In a distributed system, the principal factor which determines the task rate is the rate of the processing element to which the task is assigned for execution (for example,

```
// Computation rates.
Processor_0 = (comp_0, 1.0).Processor_0;
Processor_1 = (comp_1, 3.0).Processor_1;
Processor_2 = (comp_2, 1.0).Processor_2;
Processor_3 = (comp_3, 2.0).Processor_3;
Processor_4 = (comp_4, 1.0).Processor_4;

// Communication rates.
Network = (move_0_1, 1.0).Network + (move_1_2, 1.0).Network +
          (move_2_3, 1.0).Network + (move_3_4, 1.0).Network;

// Task definitions.
t_0 =   (comp_0, infty).t_01;
t_01 =  (move_0_1, infty).t_0;
t_1 =   (move_0_1, infty).(comp_1, infty).(move_1_2, infty).t_1;
t_2 =   (move_1_2, infty).(comp_2, infty).(move_2_3, infty).t_2;
t_3 =   (move_2_3, infty).(comp_3, infty).(move_3_4, infty).t_3;
t_4 =   (move_3_4, infty).(comp_4, infty).t_4;

// System model.
Network<move_0_1, move_1_2, move_2_3, move_3_4>(t_0 <move_0_1> t_1
<move_1_2> t_2 <move_2_3> t_3 <move_3_4> t_4)<comp_0, comp_1, comp_2,
comp_3, comp_4>(Processor_0 || Processor_1 || Processor_2 ||
Processor_3 || Processor_4)

// Throughput expression.
T1 = 1.0 * {** || (t_01 || t_1 || ** || ** || **) || (** || ** || ** || ** || **)};
```

Fig. 4. Example PEPA performance model generated automatically from a skeleton expression.

the CPU frequency). On the other hand, the inter-task communication rates are determined primarily by the communication latencies of the underlying network connecting these processing elements. Hence, to complete the model, we introduce two further sections. Based on the rate of the processing element, μ, the task rate associated with the computational activity, λ, is introduced as a preamble:

$$Processor_0 \overset{def}{=} (\lambda_0, \mu_0).Processor_0; \quad Processor_1 \overset{def}{=} (\lambda_1, \mu_1).Processor_1; \quad \cdots$$

Note here that the computational activity, λ_i, must be the same as it is used in the definition of task i. Similarly, we introduce the communication rates by adding a section which is determined from an adjacency matrix representing the communication latencies. Again, the $move_{ij}$ activities used here must match the ones in the task definitions. For an example performance model, see Fig. 4.

4.4 Analysis of performance results

We will now discuss a numerical analysis of performance results which demonstrates practical advantages of the generated models as compared to naive systems without automated performance modelling support.

In these analysis, we draw heavily on one practical application of the automatic model generation approach: dynamic scheduling of tasks in parallel and distributed applications [9][10]. Many such applications exhibit a high-level structure in which the outermost skeleton is a *Pipeline*. In Fig. 5, we plot the predicted performance of a pipeline application with five stages[8]. The performance is measured in *throughput*,

[8] To focus our analysis on the task rates, we have set the same communication rates for all the inter-task communications. This is necessary in order to minimise the effect of the communications on the relative throughputs while we contrast the performance that is achieved due to different task rates.

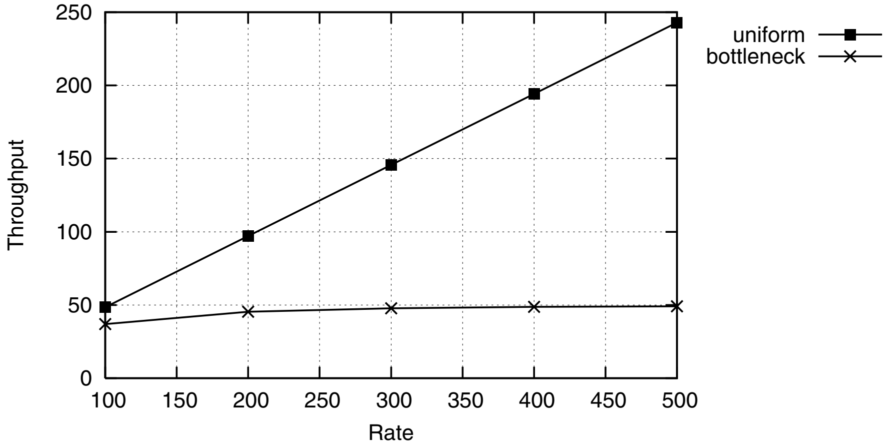


Fig. 5. The effect of bottleneck stages on the performance of a pipeline. Here, we have a pipeline with three stages. The job processing rates of all the stages, except for the middle stage is increased uniformly. The job processing rate of the middle stage is maintained at a constant value of 50 jobs per unit time.

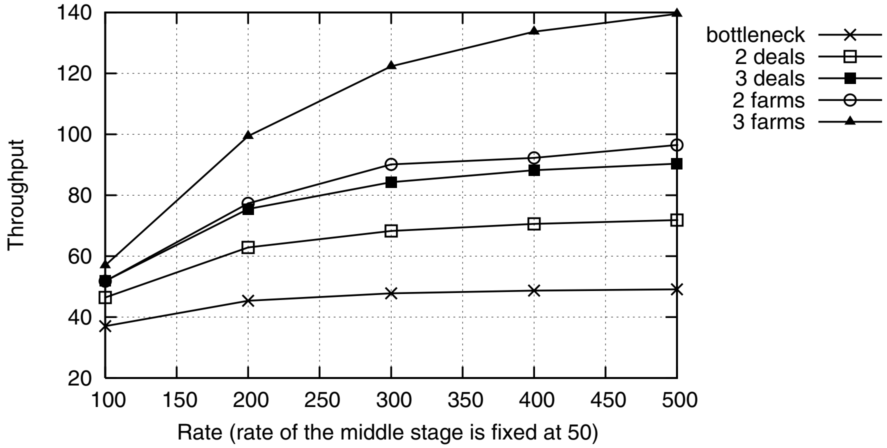


Fig. 6. The effect of the bottleneck stage in Fig. 5 is reduced by replicating the bottleneck task. The replication is done in two ways by using a *Deal* and a *Farm*. The number of replicated tasks are also varied (2 and 3 replications).

where throughput at steady state is the expected number of completed jobs per unit time.

As we can see, the throughput of the pipeline increases linearly as long as the rates at which data units are processed by each of the stages increases uniformly. However, when some stages of the pipeline become a bottleneck (in Fig. 5, we have made the middle stage a bottleneck, with its task rate kept at a constant value of 50) it is often the case that the overall performance of the pipeline degrades, staying almost at the same level (since, the throughput is determined by the worst performing task) even when the rates of the other stages are increased. This shows that in order to improve the overall performance of *Pipeline* application, we must ensure uniform task rates.

One way of ensuring uniformity of task rates is the replication of the worst performing task so that multiple tasks of the same kind can share the load. As we have discussed in Section 3, this could be done in two ways. First, we use a *Deal* where the bottleneck stage is replicated in a manner so that data is processed in round-robin fashion. Second, we use a *Farm* where the data distribution is not fixed, but probabilistic. In Fig. 6, we show the throughputs for five variations of the replication: (1) the *Pipeline* application with a middle stage bottleneck (same as shown in Fig. 5); (2) the case when the middle task is replicated twice as a *Deal*, (3) thrice as a *Deal*; (4) the case when the middle task is replicated twice as a *Farm*, (5) thrice as a *Farm*.

As we can see, the performance of the pipeline improves when the bottleneck stage is replicated. We also see that the throughput depends on the number of replications in relation to the degree of deviation of the rate of the bottleneck stage from the rate of the others; i.e., the throughput increases more sharply when the combined rate of the replicated tasks is closer to the rate of the others, than it does when the uniform rate is higher than the combined rate. This can be seen in the saturation curve as we proceed towards higher uniform rates in the other stages. We also notice that the throughputs of the *Farm* based replications are higher than those of the *Deal* replications. This, we believe, is a consequence of the strict round-robin policy that is imposed on the *Deal* replications; whereas, the policy for the *Farm* replications is determined responsively based on the given rates.

5 Conclusion

In this paper, we have discussed an automated approach which generates PEPA performance models from skeleton-based applications. Such automatic approaches are important in systems where the model must be updated regularly, and dynamically, depending on the current state of the resources. We have demonstrated in a practical setting the advantages of the generated models by contrasting the performances achieved through various task replication schemes.

References

[1] M. Aldinucci and M. Danelutto. Algorithmic Skeletons Meeting Grids. *Parallel Computing*, 32(7-8):449–462, 2006.

[2] F.A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing.* Springer Verlag, ISBN 1-85233-506-8, 2003.

[3] A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Flexible Skeletal Programming with eSkel. In *Euro-Par 2005*, LNCS 3848, pages 761–770, 2005.

[4] C. Gomes, J. Cunha, and O. Rana. Pattern Operators for Grid Environments. *Scientific Programming*, 11:237–261, 2003.

[5] OASIS WSBPL Technical Committee. Web Services Business Process Execution Language Version 2.0. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html, January 2007.

[6] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, Cambridge, Massachusetts, 1989.

[7] J. Hillston. *A Compositional Approach to Performance Modelling.* Cambridge University Press, 1996.

[8] M. Cole. Bringing Skeletons Out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, March 2004.

[9] G. Yaikhom, M. Cole, and S. Gilmore. Combining Measurement and Stochastic Modelling to Enhance Scheduling Decisions for a Parallel Mean Value Analysis Algorithm. In V. Alexandrov, D. van Albada, P. Sloot, and J. Dongarra, editors, *Intl. Conf. on Computational Science (ICCS 2006)*, volume 3992 of *LNCS*, pages 929–936. Springer, April 2006.

[10] A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Scheduling skeleton-based grid applications using PEPA and NWS. *The Computer Journal*, 48(3):369–378, 2005.