Quantitative analysis of skeleton-structured applicative programs

Anne Benoit, Murray Cole, Stephen Gilmore & Jane Hillston

School of Informatics
University of Edinburgh
Scotland
enhancers@inf.ed.ac.uk

Résumé

We propose in this paper a new technique to analyse quantitatively skeleton-structured programs. We give an overview of high-level structured parallel programming and motivate its usefulness and its relevance to applicative parallel programming. Then we introduce some basics of performance evaluation and focus on the Performance Evaluation Process Algebra PEPA. The presentation of these two concepts leads to PEPA models of two classical skeletons, Pipeline and Deal. Finally we describe a simulator, written in Objective Caml, performing single-step debugging on PEPA models, and we illustrate the use of the simulator on our models of skeletons.

1. Introduction

Applicative programming languages have many virtues. They combine strong, descriptive, static type systems ; precise compile-time analysis of functions ; high-level program structuring facilities ; a carefully-considered semantics for the language ; and a regular, predictable view of functions and data as first-class values which promotes code reuse when coupled with a polymorphic type system.

Programs written in applicative languages can be concise, aiding the process of code scrutiny and inspection which detects programming errors. Together, benefits such as these enable developers to quickly create high-quality applications which perform complex symbolic processing tasks. When compared to similar applications which were developed in programming languages without these rich benefits, applicative programs are often found to have relatively few programming errors. Socially, in the theoretical computer science community, belief in applicative programming languages runs high. Applicative programming languages are often the tool of choice for the implementation of theorem provers and other verification tools where program correctness is very highly valued [29].

For all of their incontrovertible benefits, applicative languages have a disadvantage in that it is challenging to implement run-time support for the pure functional programming style which exceeds—or even equals—that of the imperative programming model. For reasons of pragmatism [29], practical functional languages such as those in the ML family also include imperative data structures such as arrays and (type-safe) updatable references, and loop constructs to allow repetition to be expressed iteratively instead of recursively. The benefits of these features include the direct expression of imperative algorithms and the savings in memory use which accrue from replacing non-tail-recursive functions with iterations which execute in constant stack space. Applicative programmers who are concerned with achieving peak efficiency are encouraged to use the imperative programming style instead of one which overuses functions [26].

Even the imperative programming model can be inadequate for the most demanding computational tasks. For these problems developers turn to the parallel execution of programs. Cognisant that arbitrary parallelism in applicative programming languages could increase the possibility of unwanted program behaviour and subvert the good properties which made applicative languages so valuable in the first place, the designers of parallel applicative languages emphasized *structured* parallel programming [30] based on approaches such as the identification of *algorithmic skeletons* [7] which are the structured parallel programming analogue of the higher-order polymorphic functions (map, fold, and others) which are a cornerstone of sequential applicative programming. One of the design principles of skeletons is that they can be nested arbitrarily [10] leading to a natural programming model which has resonance with the regular, predictable nature of function composition in sequential applicative programming.

Structured parallel functional programs have valuable *qualitative* benefits. For example, it is guaranteed that structured parallel functional programs are free from deadlock [16, 27], and it is even possible to make programs proof for some parallel languages [15]. However, the use of a structured parallel programming approach does not provide similar *quantitative* guarantees about the efficiency of the program. It is not even guaranteed that a parallel version of a program (irrespective of the language used) will run at least as fast as the sequential version, much less surpass it. In the development of successful parallel programs, intuitions about the likely run-time performance of a parallel code can be significantly wrong. It is considerably more reliable to set the analysis of parallel codes on a formal foundation by using mathematically sound methods for modelling and performance analysis of programs.

When using quantitative analysis methods in advance of the implementation of an application exact data about run-times of functions on processors cannot be obtained. Even when a system implementation is available and deployed in active use, then the measured run-time of a function (as opposed to its asymptotic complexity) will vary depending on load on the processor on which it is executed, the size of the input data to be processed and other factors. For these reasons, the appropriate modelling formalism to use is one which uses probabilistic random variables to estimate run-times. Such a model gives rise to a stochastic process which is analysed in order to give insights into the eventual performance of the system. When the only estimate of the run-times of functions which is known is their average duration (as opposed to second or higher moments) then the appropriate random number distribution to use is the exponential distribution. Stochastic processes over a finite state space where all of the transition rates between states are exponentially distributed are an important class of stochastic process known as Continuous-Time Markov Chains (CTMCs).

CTMCs are important because they admit efficient solution methods for finding their stationary probability distribution, making them a practically useful formalism for modelling and analysis. The stationary distribution of a CTMC is a vector of probabilities expressing the likelihood of the system being in each of the states of its finite state space in the long run (that is, at equilibrium), after the necessary "warm-up" period has passed so that the influence of the initial state of the system is no longer measurable. Meaningful measures of the system's performance such as throughput can be simply computed from such a stationary probability distribution. The stationary distribution can also be used to determine hot-spots or bottlenecks in the system, underused or overused components, or other flaws such as livelocks where the system enters a subset of its state space, and can never leave this subset.

Typically, Continuous-Time Markov Chains are not used directly as a formal modelling language. Rather, a higher-level modelling language is used and a CTMC is generated from this by the application of the language semantics. High-level modelling languages which are used for this purpose include stochastic Petri nets [28], stochastic automata networks [32, 14] and stochastic process algebras [21]. A particular instance of such a high-level stochastic modelling language is Hillston's Performance Evaluation Process Algebra (PEPA) [21], where the formal semantics of the language is given via a Plotkin-style small-step operational semantics [33]. This semantic definition describes the labelled multi-transition system (LMTS) which is generated from a parallel composition of sequential PEPA processes (called PEPA components). The labels of the LMTS take the form of (activity, rate) pairs where the activity is an uninterpreted symbolic name for an event (such as an arrival into a pipeline, or a departure from a pipeline) together with the exponentially-distributed rate at which this activity can be performed. A CTMC is derived directly from this LMTS by erasing the activity names.

In the sequel, we describe the application of Hillston's stochastic process algebra PEPA to the modelling of algorithmic skeletons considering the Pipeline and Deal skeletons. Such models can then be used to evaluate the performances of any skeleton-based applications, after a re-parameterisation of the model. Section 3 describes a single-step debugger for PEPA models, and the use of the simulator is illustrated in Section 4 through an example based on the models of skeletons described earlier. Finally, we discuss related work and conclude.

2. Modelling algorithmic skeletons with PEPA

In this section, we give an overview of structured parallel programming and performance models, and then we show how we combine both approaches by making performance models of algorithmic skeletons.

2.1. An introduction to algorithmic skeletons

2.1.1. Structured parallel programming

Simple parallel programming frameworks (Posix threads, core MPI) are universal. They can be used to describe arbitrarily complex and dynamically determined interactions between activities. Programming is by careful selection and combination of operations drawn from a small, simple set.

It is difficult for programmers, examining such a program statically, to understand the overall pattern involved (if such exists), and to radically revise it. It is very difficult for implementation mechanisms (working statically and/or dynamically) to attempt optimisations which work beyond single instances of the primitives.

However, in practice, parallel programs do not actually involve arbitrary, dynamic interaction patterns. In many cases the pattern is entirely pre-determined. In other cases non-determinism is constrained within a wider pattern. The use of an unstructured parallel programming mechanism prevents the programmer from expressing information about the pattern—it remains implicit in the collected uses of the simple primitives.

The structured approach to parallelism proposes that commonly used patterns of computation and interaction should be abstracted as parameterisable library functions, control constructs or similar, so that application programmers can explicitly declare that the application follows one or more such patterns. This matters because it gives developers of parallel codes the right conceptual tools to address the issues which make correct, efficient parallel programming hard.

A fuller introduction to this research agenda can be found in [10, 11].

What we call an algorithmic skeleton [7, 35, 10] is a programming construct which abstracts such a pattern of processes and interactions. The programmer invokes one or more skeletons to describe the structure of a program, specialising each with types and operations from the application domain. Code handling the interaction and invocation of the domain specific operations is inherited implicitly from the chosen skeleton. We briefly recall the concept of Pipeline parallelism, which is of well-proven usefulness in several applications and of the Deal skeleton, which is often used nested inside a Pipeline skeleton.

2.1.2. The Pipeline skeleton

In the simplest form of Pipeline parallelism [8], a sequence of N_s *stages* process a sequence of *inputs* to produce a sequence of *outputs* (Fig. 1). All input passes through each stage in the same order, with processing of a particular input beginning as soon as its predecessor has left the first stage. Notice that parallelism is introduced by overlapping the processing of many input instances. Indeed, it is quite normal for the processing time of each individual input to be increased by pipelining. Performance benefits accrue when many such instances are processed concurrently across the pipeline.

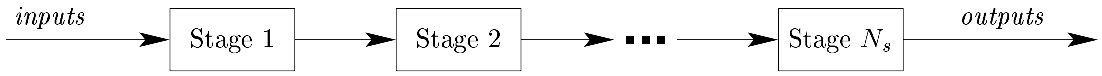

FIG. 1 – The Pipeline application

2.1.3. The Deal skeleton

It is well known that the performance of a pipeline is constrained by the processing time of its slowest stage. This may be improved by exploiting parallelism within, as well as between, stages. A simple approach is to implement the stage with several processors which take turns to accept inputs from the incoming stream, returning results to the output stream in the same order. In this way, the slow processing time of all but the first input can be masked by this intra-stage parallelism (Fig. 2). The Deal skeleton [9] abstracts this pattern. The name is chosen by analogy with the process of dealing out a pack of cards (inputs) in strict rotation to a collection of players (stage internal processors). In our model, we will have Nw_s *workers* for a given Deal skeleton s.

From the application programmer's perspective, it is important to note that this approach is only valid for stateless stage computations. If the stage maintains and updates state from one input to the next, then a more sophisticated internal parallelization strategy must be devised.

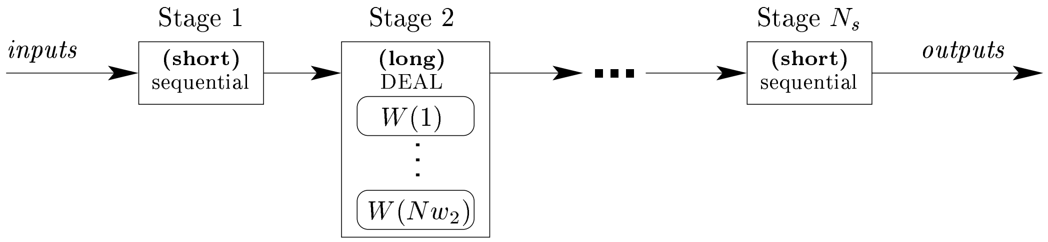

FIG. 2 – Pipeline and Deal

2.1.4. Related works

Several approaches to structured parallel programming have been developed over the past years, and we give an overview of some of them in the following. Many works have been inspired by functional programming, or are even extensions of functional languages, but not all of them. We present different kind of skeletal frameworks.

Firstly, the **Pisa Parallel Programming Language** (P3L) [30, 2, 3] was inspired by functional programming, and it was created to help the design of parallel applications with the use of skeletons. In this language, the parallel computations are restricted to a set of parallel constructs, as described earlier. The **OcamlP3l** prototype [13] marries a real functional language (Ocaml) with the p3l skeletons, forming the basis of a powerful parallel programming environment.

Some approaches have been developed on the basis of classical languages, such as the **BSMLlib library** [16] which is a library for Bulk Synchronous Parallel (BSP) programming with the functional language Objective Caml. It is based on an extension of the λ-calculus by parallel operations on a data structure named parallel vector, which is given by intention. An extension of the lazy functional language Haskell [23] has produced the parallel functional language **Eden** [5], which provides a new perspective on parallel programming. It gives programmers enough control to implement their parallel algorithms efficiently (including granularity issues) and at the same time frees them from the low level details of process management. A library of predefined skeletons has been added to the language to allow easy parallel programming.

Finally, some approaches are more detached from functional programming. The Edinburgh Skeleton Library **eSkel** [8, 9] is a C library built on the top of the Message Passing Interface MPI [19]. It allows the C programmer to use generic and flexible skeletons to abstract classical parallelism schemes. In the following, we apply our works to the pipeline and deal skeletons as they are defined in this library, since it is the library we are developing, but we keep in mind the fact that our approach could be extended to any other skeleton framework.

Now that we have given an overview of structured parallel programming and presented some common algorithmic skeletons, we focus in the following on performance evaluation, in order to introduce performance models of skeletons.

2.2. An introduction to performance evaluation

In this section, we first present the Performance Evaluation Process Algebra PEPA [21]. Then we discuss various tools for performance evaluation.

2.2.1. An introduction to PEPA

The PEPA language provides a small set of combinators. These allow language terms to be constructed defining the behaviour of components, via the activities they undertake and the interactions between them. Timing information is associated with each activity. Thus, when enabled, an activity $a = (\alpha, r)$ will delay for a period sampled from the negative exponential distribution which has parameter r. If several activities are enabled concurrently, either in competition or independently, we assume that a *race condition* exists between them. The component combinators used in the skeleton models, together with their names and interpretations, are presented informally below.

Prefix : The basic mechanism for describing the behaviour of a system is to give a component a designated first action using the prefix combinator, denoted by a full stop. For example, the component $(\alpha, r).S$ carries out activity (α, r), which has action type α and an exponentially distributed duration with parameter r, and it subsequently behaves as S.

Choice : The choice combinator captures the possibility of competition between different activities. The component $P + Q$ represents a system which may behave either as P or as Q : the activities of both are enabled. 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.

Constant : It is convenient to be able to assign names to patterns of behaviour associated with components. Constants are components whose meaning is given by a defining equation.

Hiding : The possibility to abstract away some aspects of a component's behaviour is provided by the hiding operator, denoted P/L. Here, the set L of visible action types identifies those activities which are to be considered internal or private to the component and which will appear as the unknown type τ.

Cooperation : In PEPA direct interaction, or *cooperation*, between components is the basis of compositionality. It is denoted by $P \underset{L}{\bowtie} Q$, where P and Q are the two components cooperating, called *co-operands*. The set which is used as the subscript to the cooperation symbol, the *cooperation set* L, determines those activities on which the co-operands are forced to synchronise. For action types not in L, the components proceed independently and concurrently with their enabled activities. However, an activity whose action type is in the cooperation set cannot proceed until both components enable an activity of that type. The two components then proceed together to complete the *shared activity*. The rate of the shared activity may be altered to reflect the work carried out by both components to complete the activity (for details see [21]). We write $P \parallel Q$ as an abbreviation for $P \underset{L}{\bowtie} Q$ when L is empty.

In some cases, when an activity is known to be carried out in cooperation with another component, a component may be *passive* with respect to that activity. This means that the rate of the activity is left unspecified (denoted $\top$) and is determined upon cooperation, by the rate of the activity in the other component. All passive actions must be synchronised in the final model.

The dynamic behaviour of a PEPA model is represented by the evolution of its components, as governed by the operational semantics of PEPA terms (see [21]). Thus, as in classical process algebra, the semantics of each term is given via a labelled multi-transition system (the multiplicities of arcs are significant). In the transition system a state corresponds to each syntactic term of the language, or *derivative*, and an arc represents the activity which causes one derivative to evolve into another. The complete set of reachable states is termed the *derivative set* and these form the nodes of the *derivation graph* which is formed by applying the semantic rules exhaustively.

The derivation graph is the basis of the underlying Continuous Time Markov Chain (CTMC) which is used to derive performance measures from a PEPA model. The graph is systematically reduced to a form where it can be treated as the state transition diagram of the underlying CTMC. Each derivative is then a state in the CTMC. The *transition rate* between two derivatives P and Q in the derivation graph is the rate at which the system changes from behaving as component P to behaving as Q. It is the sum of the activity rates labelling arcs connecting node P to node Q.

2.2.2. Tools for performance models

There are a number of methods and tools available for analysing PEPA models [6] in order to learn more about the systems which they model. An overview of these can be found in Figure 3.

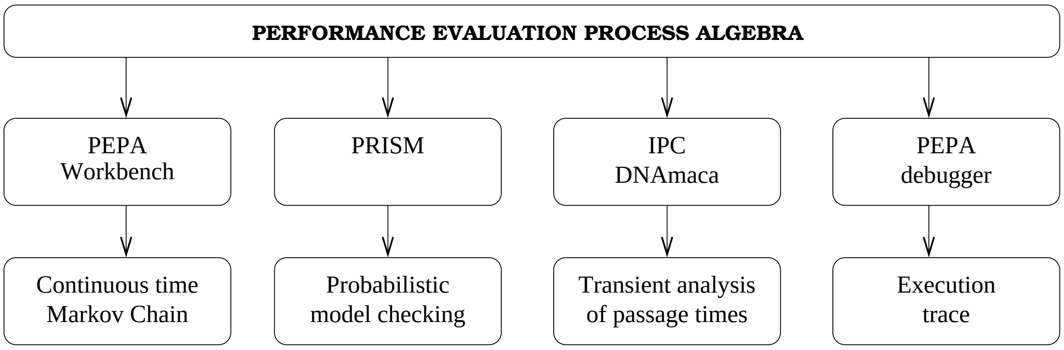

FIG. 3 – Tools for Performance Evaluation Process Algebra

One way in which a PEPA model can be solved directly is to use the PEPA Workbench [17] to generate the state space of the model and the infinitesimal generator matrix of the underlying Markov chain. The PEPA Workbench can solve this for the steady-state probability distribution of the system and performance measures such as throughput and utilisation can be computed from this.

The PEPA Workbench represents the state space of the model as a sparse matrix. A different approach to the representation of the infinitesimal generator matrix of the CTMC is taken by the PRISM probabilistic symbolic model checker [25] which stores matrices using multi-terminal binary decision diagrams (MTBDDs). These are a compact storage format for very large models which have useful symmetric properties. PRISM supports PEPA as one of its input languages and offers a range of numerical solution procedures : Power, Jacobi, forwards and backwards Gauss-Seidel, JOR and forwards and backwards SOR. In addition, PRISM allows formulae of Continuous Stochastic Logic to be model-checked against CTMC models, providing a method for undertaking custom performability verification to find the answer to queries which combine time, probability and paths through the system execution.

Yet other solution procedures are accessed via the Imperial PEPA Compiler (IPC) [4]. IPC is a functional program written in the Haskell lazy functional programming language [23]. Its function is to compile a PEPA model into the input language of Knottenbelt's DNAmaca analyser [24]. The possible solution methods offered by DNAmaca include direct methods (Gaussian Elimination, Grassmann), classical iterative methods (Gauss-Seidel, fixed SOR, dynamic SOR), Krylov subspace techniques (BiCG, CGNR, CGS, BiCGSTAB, BiCGSTAB2, TFQMR) and decomposition-based methods (these include AI (Aggregation-Isolation), AIR). The distinctive feature here is that these can be used to compute passage-time quantiles for PEPA models. Such expressions are often found in service-level agreements (SLAs) which regulate the expected timed behaviour of the system. DNAmaca plots its results as a probability distribution function (PDF) or a cumulative density function (CDF).

Finally, the behaviour of the model can be explored interactively using our newly developed single-step debugger (Section 3). The PEPA Workbench also incorporates a single-step debugger, but its use is limited because it requires the state-space of the model to have already been generated before the debugging session begins. This means that it cannot be used to find missed synchronisations which mean that the state-space of the model is prohibitively large. Our novel O'Caml implementation does not have this restriction, and stores only the current state of the model.

2.3. Models of Pipeline and Deal skeletons

We present models of the Pipeline and Deal algorithmic skeletons described in Sections 2.1.2 and 2.1.3, skeletons which are taken from the *eSkel* library. The models are Performance Evaluation Process Algebra as described in Section 2.2.1.

To model a Pipeline application, we decompose the problem into the stages, the processors and the network. Some of the stages can then be modelled as Deal skeletons.

The stages

The first part of the model is the *application model*, which is independent of the resources on which the application will be computed. The application consists of N_s stages, which are each modelled by a PEPA component $Stage_s$ $(s = 1, \ldots, N_s)$.

When $Stage_s$ is not a Deal, it executes sequentially. As its first activity, it obtains data (activity $move_s$), then processes it (activity $process_{s,1}$), and finally moves the processed data to the next stage (activity $move_{s+1}$). In the $process_{s,1}$ activity, the 1 in the index denotes the first (and only) worker for this stage (i.e. the number of workers for the stage s is $Nw_s = 1$). The second subscript will play a more important role in Deal, when there may be many workers. The definition is in Fig. 4a.

All the rates are unspecified, denoted by the distinguished symbol $\top$, since the processing and move times depend on the resources on which the application is running. These rates will be defined later, in another part of the model.

When $Stage_s$ is a Deal, we consider that we have Nw_s workers which have to process a sequence of inputs. In our model, we enforce cyclic allocation of inputs to workers by introducing, for each Deal, a *Source* component and a *Sink* component which interface between the Deal workers and the *move* actions which link this stage to its pipeline neighbours. Each worker $i \in \{1, \ldots, Nw_s\}$ first gets an input from the source with an $input_{s,i}$ action, processes it ($process_{s,i}$) then transfers its output to the sink ($output_{s,i}$). We obtain the definitions $Source_s$, $Sink_s$ and $Worker_{s,i}$ of Fig. 4b, where the workers are defined for $i = 1, \ldots, Nw_s$. All the workers are independent, and they are synchronised with the source and the sink via the *input* and *output* actions. We define $LI_s = \{input_{s,i}\}_{i \in \{1,\ldots,Nw_s\}}$ and $LO_s = \{output_{s,i}\}_{i \in \{1,\ldots,Nw_s\}}$ in the $Stage_s$ definition (Fig. 4b).

Once all the stages have been defined, the Pipeline application is then a cooperation of the different stages over the $move_s$ activities, for $s = 2..N_s$. The activities $move_1$ and $move_{N_s+1}$ represent respectively, the arrival of an input into the application, and the transfer of the final output out of the Pipeline. They do not represent any data transfer between stages, so they are not synchronised within the Pipeline application. As above, the rates on the input and output actions are left unspecified. These will be defined elsewhere in the model. The Pipeline definition is in Fig. 4c.

The processors

We consider that the application must be mapped to a set of N_p processors. Each worker is implemented by a given (unique) processor, but a processor may host several workers. In order to keep the model simple, we decide to put information about the processor (such as the load of the processor or the number of stages being processed) directly in the rate $\mu_{s,i}$ of the activities $process_{s,i}$, for $s = 1..N_s$ and $i = 1..N_{w_s}$ (these activities have been defined for the components $Stage_s$). Each processor is then represented by a PEPA component which has a cyclic behaviour, consisting of sequentially processing inputs for a worker. Some examples follow.

– In the case with no Deal, when $N_p = N_s$, we map one worker per processor :

$$Processor_i \stackrel{def}{=} (process_{i,1}, \mu_{i,1}).Processor_i$$

– If several workers are hosted by the same processor, we use a choice composition. In the following example ($N_p = 2$, $N_s = 2$, and the first stage is a Deal with 2 workers), the first processor processes the first worker of both stages, and the second processor processes only the second worker of Stage 1 (so that Stage 1 is distributed across two processors).

$$\begin{aligned} Processor_1 &\stackrel{def}{=} (process_{1,1}, \mu_{1,1}).Processor_1 \\ &+ \; (process_{2,1}, \mu_{2,1}).Processor_1 \\ Processor_2 &\stackrel{def}{=} (process_{1,2}, \mu_{1,2}).Processor_2 \end{aligned}$$

More generally, since all processors are independent, the set of processors is defined as a parallel composition of the processor components (Fig. 4d).

The network

Rather than directly representing the physical structure of the underlying network architecture, our network model is designed to allow us to derive the rates of the logical communication actions ($move, input, output$) of our Pipeline and Deal models from the NWS monitored physical processor to

a. Stage without Deal

$$Stage_s \stackrel{def}{=} (move_s, \top).(process_{s,1}, \top).(move_{s+1}, \top).Stage_s$$

b. Stage with Deal

$$\begin{aligned}
Source_s &\stackrel{def}{=} (move_s, \top).(input_{s,1}, \top).\\
&\quad\ (move_s, \top).(input_{s,2}, \top).\\
&\quad\ \ldots\\
&\quad\ (move_s, \top).(input_{s,N_{w_s}}, \top).Source_s\\
Worker_{s,i} &\stackrel{def}{=} (input_{s,i}, \top).(process_{s,i}, \top).(output_{s,i}, \top).Worker_{s,i}\\
Sink_s &\stackrel{def}{=} (output_{s,1}, \top).(move_{s+1}, \top).\\
&\quad\ (output_{s,2}, \top).(move_{s+1}, \top).\\
&\quad\ \ldots\\
&\quad\ (output_{s,N_{w_s}}, \top).(move_{s+1}, \top).Sink_s\\
Stage_s &\stackrel{def}{=} Source_s \underset{LI_s}{\bowtie} \left(Worker_{s,1} \mid\mid \ldots \mid\mid Worker_{s,N_{w_s}}\right) \underset{LO_s}{\bowtie} Sink_s
\end{aligned}$$

c. The Pipeline application

$$Pipeline \stackrel{def}{=} Stage_1 \underset{\{move_2\}}{\bowtie} Stage_2 \underset{\{move_3\}}{\bowtie} \cdots \underset{\{move_{N_s}\}}{\bowtie} Stage_{N_s}$$

d. The processors

$$Processors \stackrel{def}{=} Processor_1 \mid\mid Processor_2 \mid\mid \ldots \mid\mid Processor_{N_p}$$

e. The network

$$\begin{array}{lllll}
Network \stackrel{def}{=} & & (move_1, \lambda_1).Network & + \ldots + & (move_{N_s+1}, \lambda_{N_s+1}).Network\\
& + & (input_{s,1}, \lambda I_{s,1}).Network & + \ldots + & (input_{s,N_{w_s}}, \lambda I_{s,N_{w_s}}).Network\\
& + & (output_{s,1}, \lambda O_{s,1}).Network & + \ldots + & (output_{s,N_{w_s}}, \lambda O_{s,N_{w_s}}).Network
\end{array}$$

f. Overall model

$$Mapping \stackrel{def}{=} Network \underset{L_n}{\bowtie} Pipeline \underset{L_p}{\bowtie} Processors$$

FIG. 4 – PEPA definitions

processor latency information. Using λ_s for the rate of a $move_s$ and $\lambda I_{s,i}$ and $\lambda O_{s,i}$ for the respective rates of $input_{s,i}$ and $output_{s,i}$ activities, the definition of the network is straightforward.

For example, assuming that only the stage s is a Deal, we obtain the definition of Fig. 4e. If some other stages are also modelled as Deals, we need to add their *input* and *output* activities into the choice.

Pipeline with Deal

Once we have defined the different components of the model, we just have to map the stages onto the processors and the network by using the cooperation combinator. Two cooperation sets are used. L_n synchronises the application and the network (it is the set of all the *move*, *input* and *output* activities), while L_p synchronises the application and the processors (it is the set of all the *process* activities). The definition is in Fig. 4f.

3. Description of the simulator

High-level models of complex systems are not obviously correct just because they have been expressed in a theoretically well-founded formal language. Like programs, they may contain mistakes which need to be repaired.

Errors in computer programs sometimes fall into recognisable classes (divide by zero, array bounds violation, and the like). Sometimes errors in programs are more difficult to detect (returning the wrong integer value, an incorrectly formatted string, and so forth). These errors need to be detected by comparing the incorrectly-returned value and the expected one.

Similarly, errors in high-level models of stochastic processes are sometimes of an obvious kind, and sometimes not. In the former category are deadlocks and synchronisations in which neither partner in the synchronisation determines the rate. Errors such as these can be detected during the generation of the state space, or before. Errors of the second kind include specifying Markov processes which have too few states or transitions (or too many), and protocol errors where activities are being performed in the wrong order by some component. These are much more difficult to detect because nothing in the generation of the state space of the model or the solution for the long-run probability distribution will fail in this case. As with programs which return the wrong value, there is a logical error which could be found by comparing the generated state space with the expected one. Unfortunately though, we do not have an expected state space to compare against. Errors such as these are very difficult to detect, and there is no easy solution here.

To provide partial assistance with this problem we have implemented a debugger for PEPA models which allows the modeller to step through the state space of the model looking for missed synchronisations, unreachable components, or other modelling errors. The debugger is implemented by embedding the PEPA language in Objective Caml and using the O'Caml top level loop to issue commands to compute the derivatives of the current model state, choose activities to follow, and single-step through the model or set breakpoints on components or activities and run the model until they are encountered. This allows a modeller to find unreachable activities or components, pointing to a problem with the model.

This simulator is completely independent from the single-step debugger implemented in the PEPA Workbench. The use of the applicative language O'Caml implies a rigorous type checking and allows us to implement much more functionalities than what was available in the previous debugger. Moreover, there is no need to generate the state space of the model before we can use it.

The code of the new simulator is closely related to the model that we are considering, since we define every activity and rate as a specific type to allow a rigorous type checking within the model.

We detail the code of such simulator in the next section on our models of skeletons. The core of the simulator is of course the same for every model, but it is necessary to include directly in the code the part dependent from the model. We want to point out that one of the objectives of the simulator will be to have its model-dependent code generated automatically from the model we want to debug, allowing a straightforward use. Even if this automatic generation has not been implemented yet, we show in the next section how such simulator would work on a specific example.

4. An example

In this section we illustrate the use of the new O'Caml simulator for the skeleton models presented in Section 2.3. As it has been pointed out previously, a part of the code of this simulator is model-dependent and thus has been written by hand, but it will in future work be generated automatically from a compiler of PEPA models. The goal of this section is to show how the simulator works through an example.

We consider a two stages pipeline, where the first stage is a deal with two workers. The first part of the simulator is specific to the model, it consists of the definition of the identifiers used in the model, as well as the activities and rates. These match the description of the model given in Section 2.3. We need to add the special activity `Tau` to allow hiding in PEPA models. The rate type allows us to define symbolic rates appearing in the model, to use these instead of the real value of the rate for the debugging purpose. The special rate `Top` is added for synchronisation purposes, it corresponds to the $\top$ symbol in PEPA, and the rate `Const` allows us to directly assign a value without using a symbolic name.

```
type identifier = Source | Worker1 | Worker2 | Sink | Stage1 | Stage2
                | Network | Proc1 | Proc2 | Procs;;
type activity   = Tau of activity | Input1 | Input2 | Output1 | Output2
                | Move1 | Move2 | Move3 | Process1 | Process2;;
type rate       = Top | Const of float | Lambda | LambdaI | LambdaO | Mu;;
```

We then define the type component according to the PEPA semantics. The type checking of the various parts of the model is then performed automatically from this definition.

```
type component  = Prefix of ((activity * rate) * component)
                | Choice of component * component
                | Coop   of component * activity list * component
                | Hiding of component * activity list
                | Var    of identifier;;
```

The last part of the simulator relevant for our model is the definition of the model, which consists of a function defining every identifier. We just display a part of it since it is all very similar to the models we are referring to.

```
let definition = function
    Source  -> Prefix ((Move1, Top), Prefix ((Input1, Top),
               Prefix ((Move1, Top), Prefix ((Input2, Top), Var Source))))
  | Worker1 -> Prefix ((Input1,Top), Prefix ((Process1, Top),
                        Prefix ((Output1, Top), Var Worker1)))
  | Stage1  -> Coop (Var Source, [Input1;Input2], Coop ((Coop (Var Worker1, [],
                    Var Worker2)), [Output1;Output2],Var Sink))
  | Stage2  -> Prefix ((Move2, Top),  Prefix ((Process2, Top),
```

```
                                              Prefix ((Move3, Top), Var Stage2)))
  | Network -> Choice (Choice (Choice (Choice (Choice (Choice (
                    Prefix ((Move1,   Lambda),  Var Network),
                    Prefix ((Input1,  LambdaI), Var Network)),
                    Prefix ((Output1, LambdaO), Var Network)), ...))))
  | Proc1   -> Prefix ((Process1,Mu), Var Proc1)
  | Procs   -> Coop (Var Proc1, [], Var Proc2)
  | ...
  ;;
```

The simulator consists then of a series of definitions and functions to allow a pretty print of the components. We can also assign real values to the symbolic rates defined earlier. The core of the simulator is the function `one_step_derivatives`, which returns the list of the derivatives of a given component, together with the activities and rates allowing us to reach these derivatives.

```
val one_step_derivatives: component -> ((activity * rate) * component) list = <fun>
```

The use of this function allows the user to locate by hand the problems, and to ensure that everything is corresponding to his/her expectations. The obvious errors in the definition of the model are detected automatically by O'Caml thanks to the rigorous type checking.

To use it, we firstly need to define the component that we wish to analyse. In our case, the Pipeline application is defined by

```
let pipeline = Coop(Var Procs, [Process1;Process2],
               Coop(Coop(Var Stage1,[Move2], Var Stage2),
                 [Move1;Move2;Move3;Input1;Input2;Output1;Output2],Var Network));;
```

Then we can print the result obtained by `one_step_derivatives`, which indicates the list of the activities which can occur and the corresponding rate, as well as the derivative component. In this example, the only activity enabled is `Move1`, with a rate `lambda`.

```
# print_string (prettyprint_derivatives (one_step_derivatives pipeline));;
-(move1, lambda)-> (Processors <process1, process2> ((((input1, top).(move1, top).
 (input2,top).Source <input1,input2> ((Worker1 <> Worker2) <output1,output2> Sink))
 <move2> Stage2) <move1, move2, move3, input1, input2, output1, output2> Network));
- : unit = ()
```

It is then possible to evolve through the system by applying the one step debugger to the resulting components.

Notice that as far as our skeletons models are concerned, different parameterisations of the models should not affect the structure of the underlying continuous time Markov Chain. The debugging phase is therefore necessary only once per skeleton models, and not for every applications using such skeletons. However, when considering a new PEPA model, an adequate simulator should be automatically created and the single-step debugger may help finding errors and problems in the model.

5. Related work and Conclusions

Many analysis tools for stochastic process algebras operate by simulating a path through the state space of the model, following the small-step operational semantics of the language. Here, and in an earlier paper [18], our aim was to assist with finding errors in the formulation of the model. In other works the authors have different aims, but they could be used to achieve some of the same ends.

Frequently, a motivation for single-step, transition-following execution of stochastic process models is because it leads to a simulation of the system behaviour. When a very general class of models is being considered, with general distributions for random variable, deterministic delays or behavioural non-determinism, this may be the only analysis method which is available.

A simulation-based approach to the analysis of stochastic process calculus models is found in the work of Philips and Cardelli [31] on an abstract machine model of a variant of Priami's stochastic π-calculus [34]. There, Gillespie's algorithm is implemented to allow the model of the system to be simulated, giving rise to a trace of a time-limited execution path. The simulator is implemented in Objective Caml and produces output files which can be rendered in proprietary third-party tools to visualise the results.

Some other works aim to enhance the quantitative aspects of skeleton-structured applicative programs, but taking another angle of view. As an example, Alt and Gorlatch [1] propose an optimisation of the mechanisms of Java RMI (Remote Method Invocation) in order to improve the performances of grid applications. Another approach is taken by Di Cosmo and Pelagatti [12], motivated by the development of OcamlP3l. They introduce a calculus which describes several strategies to distribute arrays on a set of processors, and a cost is associated to each distribution. Thus, this approach allows to take quantitative decisions to improve the performance of the application. Finally, Hayashi and Cole [20] present work on VEC-BSP, a simple functional language which incorporates concepts of "shape" [22] and parallel skeletons to enable definition of a statically evaluated execution cost calculus, itself parameterised by the cost measures of BSP.

We made in this paper an overview of structured high-level parallel programming with an emphasis on work about skeletons. Then we made a short introduction to performance evaluation through Performance Evaluation Process Algebra (PEPA) and we presented classical tools based on process algebra models. Finally we described how we can mix both subjects by proposing performance models (in PEPA) of algorithmic skeletons (from the *eSkel* library).

We then introduced the idea of a simple one-step debugger for PEPA models. A first version of the debugger has been implemented with the help of the applicative language O'Caml, allowing a rigorous type checking and an easy detection of errors within the models. The debugger has been tested on our PEPA models of skeletons, but it can be used on any other PEPA models.

What we need to do next is to implement a compiler of PEPA models which will automatically generate the single-step debugger for any model. This should be quite straightforward once we have made a detailed analogy between the PEPA definitions and the corresponding O'Caml definitions which need to be included in the debugger. The core of the debugger is common to all models, but the part specific to the model, including the definition of the components and activities, needs to be rewritten to match the PEPA description. Moreover, more functionality could be added to the simulator, for example to allow several derivation steps or derivate with respect to a subset of activities as long as it is possible.

Further work will include a case study involving more skeletons and showing how we can cope with problems on a real application. As well, the simulator will be applied on applications which are not necessarily based on skeletons, and this task will be much easier once the automatic generation of the debugger will be implemented.

Références

[1] M. Alt and S. Gorlatch. A Prototype Grid System Using Java and RMI. In V. Malyshkin, editor, *Seventh International Conference on Parallel Computing Technologies (PaCT 2003)*, number 2763 in LNCS, pages 401–414. Springer-Verlag, 2003.

[2] B. Bacci, B. Cantalupo, M. Danelutto, S. Orlando, D. Pasetto, S. Pelagatti, and M. Vanneschi. An environment for structured parallel programming. In L. Grandinetti, M. Kowalick, and M. Vaitersic, editors, *Advances in High Performance Computing*, pages 219–234. Kluwier, 1997.

[3] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P3L : A structured high level programming language and its structured support. *Concurrency : Practice and Experience*, 7(3) :225–255, May 1995.

[4] J.T. Bradley, N.J. Dingle, S.T. Gilmore, and W.J. Knottenbelt. Derivation of passage-time densities in PEPA models using IPC : The Imperial PEPA Compiler. In G Kotsis, editor, *Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems*, pages 344–351, University of Central Florida, October 2003. IEEE Computer Society Press.

[5] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. Eden : Language Definition and Operational Semantics. Technical Report 10, Philipps-University of Marburg, 1996. Available at `http://www.mathematik.uni-marburg.de/fb12/bfi/bfi10.ps`.

[6] G. Clark, S. Gilmore, J. Hillston, and N. Thomas. Experiences with the PEPA performance modelling tools. *IEE Proceedings—Software*, 146(1) :11–19, February 1999. Special issue of papers from the Fourteenth UK Performance Engineering Workshop.

[7] M. Cole. *Algorithmic Skeletons : Structured Management of Parallel Computation*. MIT Press & Pitman, ISBN 0-262-53086-4, 1989. Web page available at `http://homepages.inf.ed.ac.uk/mic/Pubs/skeletonbook.ps.gz`.

[8] M. Cole. eSkel : The **e**dinburgh **Ske**leton **l**ibrary. Tutorial Introduction. *Internal Paper, School of Informatics, University of Edinburgh*, 2002. Web page available at `http://homepages.inf.ed.ac.uk/mic/eSkel/`.

[9] M. Cole. eSkel : The **e**dinburgh **Ske**leton **l**ibrary Version 2.0 – Draft API reference manual. *Internal Paper, School of Informatics, University of Edinburgh*, 2003. Web page available at `http://homepages.inf.ed.ac.uk/mic/eSkel/`.

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

[11] M. Cole. Why structured parallel programming matters. Keynote address at EuroPar 2004, September 2004. Pisa, Italy.

[12] R. Di Cosmo and S. Pelagatti. A calculus for dense array distributions. *Parallel Processing Letters*, 13(3) :377–388, 2003.

[13] M. Danelutto, R. D. Cosmo, X. Leroy, and S. Pelagatti. Ocamlp3l a functional parallel programming system. Technical Report LIENS-98-1, E.N.S. Paris, 1998.

[14] P. Fernandes. *Méthodes Numériques pour la solution de systèmes Mar koviens à grand espace d'états*. PhD thesis, Institut National Polytechnique de Grenoble, France, 1998.

[15] F. Gava. Une bibliothèque certifiée de programmes fonctionnels BSP. In V. Ménissier-Morain, editor, *Journées Francophones des Langages Applicatifs (JFLA 2004)*, pages 55–68. INRIA, 2004.

[16] F. Gava and F. Loulergue. A Polymorphic Type System for Bulk Synchronous Parallel ML. In V. Malyshkin, editor, *Seventh International Conference on Parallel Computing Technologies (PaCT 2003)*, number 2763 in LNCS, pages 215–229. Springer Verlag, 2003.

[17] S. Gilmore and J. Hillston. The PEPA Workbench : A Tool to Support a Process Algebra-based Approach to Performance Modelling. In *Proceedings of the Seventh International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, number 794 in Lecture Notes in Computer Science, pages 353–368, Vienna, May 1994. Springer-Verlag.

[18] S. Gilmore and J. Hillston. Performance modelling in PEPA with higher-order functions. In N. Thomas and J. Bradley, editors, *Proceedings of the Sixteenth UK Performance Engineering Workshop*, pages 35–46, Department of Computer Science, The University of Durham, July 2000.

[19] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI, 2nd edition.* MIT Press, ISBN 0-262-57132-3, 1999.

[20] Y. Hayashi and M. Cole. Static Performance Prediction of Skeletal Parallel Programs. *Parallel Algorithms and Applications*, 17(1) :59–84, 2002.

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

[22] C.B. Jay. Costing Parallel Programs as a Function of Shapes. *Science of Computer Programming*, 37(1-3) :207–224, 2000.

[23] S. Peyton Jones. *Haskell 98 Language and Libraries.* Cambridge University Press, ISBN 0521826144, 2003.

[24] W.J. Knottenbelt. Generalised Markovian analysis of timed transition systems. Master's thesis, University of Cape Town, 1996.

[25] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM : A hybrid approach. In J.-P. Katoen and P. Stevens, editors, *Proc. 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 52–66. Springer, 2002.

[26] X. Leroy. Writing efficient numerical code in Objective Caml, July 2002. Web page available at `http://caml.inria.fr/ocaml/numerical.html`.

[27] F. Loulergue, F. Gava, M. Arapinis, and F. Dabrowski. Semantics and Implementation of Minimally Synchronous Parallel ML. *International Journal of Computer and Information Science*, 5(3), 2004.

[28] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets.* John Wiley & Sons, New York, 1995.

[29] R. Milner. How ML evolved. *Polymorphism—The ML/LCF/Hope Newsletter*, 1(1), 1983.

[30] S. Pelagatti. *Structured Development of Parallel Programs.* Taylor & Francis, London, 1998.

[31] A. Phillips and L. Cardelli. A correct abstract machine for the stochastic pi-calculus. In Anna Ingólfsdóttir and Hanne Riis Nielson, editors, *Proceedings of the Second International Workshop on Concurrent Models in Molecular Biology*, pages 11–25, August 2004. To appear in ENTCS.

[32] B. Plateau. *De l'Evaluation du parallélisme et de la synchronisation.* PhD thesis, Université de Paris XII, Orsay (France), 1984.

[33] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[34] C. Priami. Stochastic π-calculus. In S. Gilmore and J. Hillston, editors, *Proceedings of the Third International Workshop on Process Algebras and Performance Modelling*, pages 578–589. Special Issue of *The Computer Journal*, 38(7), December 1995.

[35] F.A. Rabhi and S. Gorlatch. *Patterns and Skeletons for Parallel and Distributed Computing.* Springer Verlag, 2002.