Simulating Taverna workflows using stochastic process algebras

Vasa Curcin Paolo Missier David De Roure

November 21, 2010

Abstract

Scientific workflows provide powerful middleware for scientific computing in that they represent a central abstraction in the research task, by simultaneously acting as an editable action plan, collaboration tool, and executable entity. Taverna workflows in particular have been widely accepted in the bioinformatics community, due to their flexible integration with web service analytical tools that are the essential tools of any bioinformatician. However, the semantics of Taverna has so far only been qualified in terms of the functional composition and data processing. While correct, and useful for reasoning about functional and trace equivalences, this aspect does not help with modelling the throughput and utilisation of individual services in the workflow. In this paper we present a stochastic process model for Taverna, and use it to perform execution simulations in Microsoft's SPIM tool. The model also opens up the possibilities for further static analyses that are explored.

1 Introduction

Workflows are in wide use in an increasing number of application domains of experimental science where computational methods are of key interest. Workflow modelling is a form of high-level scripting, designed to let domain experts specify, using limited programming skills, complex orchestrations of compute- and data-intensive tasks. Modelling workflow semantics has been an ongoing effort in recent years, motivated both by the increase in the number of publicly available workflows, and of the applications depending on them. Research is focused mainly on the analysis of workflow correctness, performance, and reliability. Furthermore, workflow repositories such as myExperiment brought the need to reliably qualify the workflows they offer. Some of the past work included qualifying the functional behaviour of Taverna workflows[Turi et al, 2007], formal reasoning about Discovery Net workflows [Curcin et al, 2009], composition of process semantics of Kepler[Goderis et al, 2007], and reachability analysis of YAWL Petri Nets [Gottschalk et al, 2007] .

This paper is specifically focused on Taverna, a workflow management system that is in wide use in the Life Sciences as well as other scientific application

domains Hull et al [2006]. Two of Taverna's features, that make it potentially suitable for data-intensive processing, are its reliance on remote web services as workflow tasks, and its capability to process data streams. These features make it particularly interesting to focus on resource invocation as the central modelling element. This is done by considering the rate of execution in the model - ie. how frequently a service returns a result and passes it on to the next service - thereby enabling the performance of workflows to be analysed either statically, by checking the model properties, or pseudo-statically using a series of randomised simulations to derive a prediction.

1.1 Problem statement and contributions

Composition of distributed web services used to perform a scientific study, is the standard paradigm for the Taverna user community, as demonstrated by the collection of workflows in the myExperiment repository. The two key properties of interest in such scenarios are:

- **Thread utilisation.** Individual nodes, or processors, in a workflow represent services, each with a specified capacity in terms of the number of threads available. Knowing how the average demand on the processor changes with variations in the performance and capacity of other processors is key to understanding the necessary resources for the workflow to run.
- **Output rate and timing.** During workflow construction, knowing the output rate of a particular node helps in determining the number of threads needed to cope with the output. Furthermore, in long running processes, and in ones where multiple services may be running on the same physical resource, it is useful to know when a particular process will start and end executing.

As the first step towards addressing these challenges, in this paper we present a mechanism to run experiments based on known node past behaviour, and generate simulation graphs displaying the processor activity and outputs produced by each workflow node. Specifically, the paper uses the stochastic π-calculus[Priami, 1995; Phillips et al, 2006] to model the behaviour of Taverna workflow based on the distribution of individual services' execution time. Process algebras are abstract languages that offer a flexible and powerful mechanism for specification and design of concurrent systems. Their loose coupling of individual agents with well-defined communication mechanisms provides a natural and immediate mapping to workflow nodes and links between them. However, the analysis using process algebras is usually based on finite state machines and requires advanced state reduction mechanisms.

1.2 Paper Organisation

Section 2 introduces the basics of Taverna. Section 3 gives an overview of stochastic process algebras and stochastic π in particular. Section 4 explains the

mapping of Taverna workflows to stochastic processes. Section 5 demonstrates the concept by describing the implementation of the model in S-Pi Model viewer [Phillips and Cardelli, 2004; Microsoft, 2007] and showcases simulation of one Taverna workflow from myExperiment repository. Section 6 summarises the work done and gives pointers for further study.

2 Background

2.1 The Taverna dataflow model

A dataflow specification in Taverna is a directed graph $D = (\mathcal{P}, E)$ where nodes $P \in \mathcal{P}$ are workflow processors, which represent either a web service operation, or a script in a common language, e.g. Beanshell or R. Processors have a set of input and output ports, I_P and O_P, respectively. When the processor stands for a service operation, its ports describe the operation's signature. Note that a processor can also be a workflow itself, allowing for structural nesting.

An arc $e \in E$, denoted $e = P : Y \rightarrow P' : X$, connects port Y of processor P to port X of P', and denotes a data dependency from Y to X. In this model, data items have a type T, which is either a simple type τ (string, number, etc.), or a list of values having the same type, and nested to arbitrary levels, according to the syntax $T ::= \tau | [T]$. For example [[“foo”, “bar”], [“red”, “fox”]] has type $[[\tau]]$ where τ denotes the string type.

Conceptually, workflow execution follows a pure dataflow model [Johnston et al, 2004]: a processor P is ready to be activated as soon as all of its input ports that are destinations of an arc are assigned to values (ports with no incoming arcs may be assigned a default value). In the actual implementation, the workflow engine manages a pool of threads of pre-defined size, and one thread from the pool is allocated to a ready processor as soon as it is available, in a greedy fashion. A processor behaves simply as a function that is applied to the value on the input ports, and produces new values on its output ports. Its activation involves the invocation of activity associated to the processor, i.e., invoking a Web Service operation. Values assigned to output ports are immediately transferred through the data links to input ports of downstream processors. Thus, the overall dataflow computation proceeds by pushing data through the data links from one processor to all of its successors, starting with the items that are assigned to the initial workflow inputs, and ending when no more inputs are left to be processed, communicated through the end-of-stream, or stop, token. Note that the values assigned to the ports are always discrete values, as described, but the input to the workflow may consist of a stream of such values, on unbounded length.

2.2 Multiple node activation and parallelism in Taverna

Under certain circumstances, list-valued inputs may produce multiple activations of the same processor, each operating independently, and potentially in

parallel, on a combination of the input list elements. More precisely, this happens when a processor receives a list-valued input that cannot be consumed by one single invocation to the underlying activity. As a typical example, consider a search processor P, which returns a list of hits (data items that satisfy the search) based on some input keyword, followed by a processor P' that can consume one of those hits at a time. This situation will cause P' to iterate on each element of its input list. As each iteration is assumed to be data-independent from each other, the corresponding processor activations can all potentially proceed in parallel, i.e., they can all be allocated to parallel threads.

In general, this *implicit iteration* behaviour is determined by the discrepancy between the *declared depth* d_d of the list expected on an input port (the depth is 0 for an atomic value), and the *actual depth* d_a of an input value assigned to it. As we have seen in our simple example, the two may differ. In particular, d_d is determined by the signature of the actual activity, i.e., a Web Service operation, that the processor represents, while d_a depends on the entire history of the input value, from the initial input down to that node in the graph.

When $d_a > d_d$, the processor "unfolds" the list so that when the activity is executed, its inputs are of the expected list depth.[1] In the simple case of a single input, this behaviour can be described in terms of the higher-order **map** operator, namely the evaluation of P on input list $v = [v_1 \dots v_n]$, denoted $(P\ v)$, unfolds into $(\mathbf{map}\ P\ v) = [(P\ v_1) \dots (P\ v_n)]$. In the more general case of multiple inputs with mismatching list depth, the workflow designer has the option to select a specific *iteration strategy* for the processor. These strategies involve (a) the combination of the list input values into a new, single list-valued structure, and (b) the iterative activation of the processor on that structure. Two operators, namely a cross product and a "dot" product, are available in Taverna to combine the input values. Their semantics, and the general semantics of a processor execution with implicit iteration, is beyond the scope of this paper, and can be found in [Sroka et al, 2009]. What matters in this context is that the exact iteration structure for each processor *can be determined statically*, i.e., by propagating the declared depths of the workflow inputs[2] through the workflow graph, thereby enabling the stochastic model to know which node will have iterative behaviour and which one will not. A description of the propagation algorithm, used in the context of query processing of provenance traces from a workflow run, can be found in [Missier et al, 2010].

2.3 Pipelining in Taverna

The greedy thread allocation strategy described earlier translates into parallel multiple activations of one processor during an implicit iteration loop, subject to a constraint on the max number of threads available in the pool (a configurable

[1] The case $d_a < d_d$, i.e., when the processor expects a list value of depth greater than the one provided, is dealt with simply by wrapping the input into one or more additional list structures, as needed.

[2] The workflow engine enforces the assumption that the user-supplied valued assigned to the initial inputs are indeed of the expected depth.

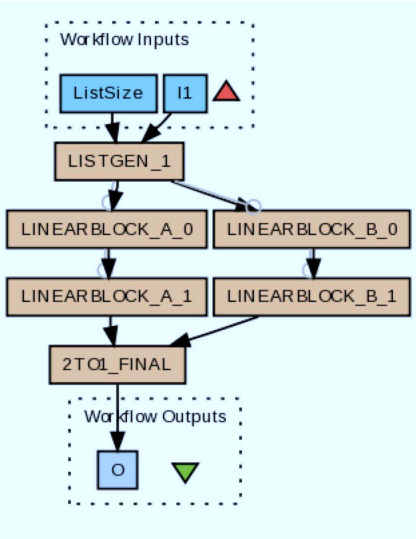


Figure 1: Example workflow for simulation

parameter that can be set at workflow specification time). One consequence of this strategy is that, in some cases, processors that are arranged along a path in the workflow graph can be activated in a pipelined fashion. To appreciate this effect, consider our running example of Fig. 1, characterised by a linear chains of identical processors. While each of these has only one input port, with declared depth 0, the workflow is designed to feed a list of depth 1 at the top of each of the chains. This means that the first processor in the chain creates (at most) one concurrent thread per list element. As soon as one of these threads completes, its result is pushed through the edge and to the next processor's input port. Here there is no need to wait for the entire list to arrive, because each input list element can again be dealt with independently from the others, as part of a new iteration loop similar to the one of the previous processor. This partial input can therefore be immediately allocated to a new thread, again in a greedy fashion. Overall, multiple parallel pipelines are created as the result of applying this strategy through an entire chain of iterating processors (clearly, however, any non-iterating processor in the chain breaks each of the pipelines).

2.4 Stochastic modelling of programs

Stochastic process algebras are characterised by the quantitative measure r that is attached to communication actions, indicating the rate of interaction that corresponds to the temporal delay drawn at random from the exponential distribution defined by r.

The usage of stochastic process algebras for performance analysis of programs is an established research area, that found its application in network

performance modelling and analysis of biochemical systems. PEPA [Gilmore and Hillston, 1994], PRISM [Kwiatkowska et al, 2009], TIPP[Hermanns et al, 1995] and MPA[Bernardo et al, 1994] all rely on Markov chain semantics to perform various types of analysis, mostly focusing on the notion of steady state - a network of states that repeat indefinitely. The system that has reached a steady state is then analysed by varying some set of parameters, e.g. investigating how a load on the server changes over time with respect to its throughput capability.

While, conceptually, the process algebra formalisms the aforementioned tools are based on are capable of handling unbounded states, all of their implementations introduce practical restrictions with the aim of constructing finite Markov chains that can be solved using Ordinarily Differential Equations. These restrictions consist either of preventing the creation of new processes either completely (PEPA) or only allowing replication on the top definition level (PRISM). This type of non-replicating processes are called automata.

Transient analysis is a different approach, based on establishing system properties relative to particular events in the timeline. Continuous Time Logic CSL [Aziz et al, 2000], which attaches probability measures to temporal operators from non-stochastic Continuous Tree Logic CTL [Huth and Ryan, 2000], is commonly used to investigate the probability of an event occurring in a particular time window and is supported in PRISM. Additionally, transient analysis does not necessarily require the system to ever reach a steady state, allowing two possibilities for performing the model checking of terminating systems: through finding the full model, and by performing a sufficient number of simulation runs.

In terms of workflows, steady state analysis can be used with infinite data streams to observe the natural bounds in the system and how the usage and activities fluctuate in the equilibrium. However, transient analysis has more immediate applications, since workflows operating on finite datasets are not guaranteed to have a steady state apart from the terminating one, and, particularly in the case of streaming semantics, the number of states is generally too large to efficiently model and makes some existing state reduction techniques unfeasible - such as the state vector form used for workflows in Curcin et al [2009] and general PEPA models in Hillston [2005]. In this paper, we will employ the most basic form of transient analysis, consisting of simulating the actual runs of the workflow to deduce performance behaviour, with the view of using the theoretical model for further types of transient analysis in the future.

3 Stochastic π-calculus

Process algebras are a family of languages designed for the specification of concurrent behaviour. The system is defined as a set of named processes that evolves through a series of *actions* that represent interactions between processes through incoming and outgoing messages. The basic operators found in most of these languages are sequential actions, deterministic and nondeterministic choice and parallel composition, which are extended further depending on the aim of the algebra. First algebras were introduced in works by Robin

Milner (CCS [Milner, 1989]) and C.A.R. Hoare (CSP [Hoare, 1983]). Important later languages included ACP [Bergstra and Klop, 1989] and especially π-calculus [Milner, 1990]. Furthermore, a number of automatic reasoners over such algebras also exist, such as Edinburgh Concurrency Workbench [Moller and Stevens, 2009], CubeVM [Peschanski and Hym, 2006], Another Bisimilarity Checker [Briais, 2009] and Rocke's tableaux method [Rockl et al, 2001] for Isabelle/HOL.

π-calculus is a special instance of general messagepassing process algebra that introduces mobility, whereby messages passed between processes can be processes themselves, allowing the communication topology to change dynamically, which is not supported in algebras such as CCS. The stochastic extension of the π-calculus, introduced in Priami [1995] and further refined in Phillips and Cardelli [2007]; Phillips et al [2006], associates the input, output, and silent actions of the calculus with rate r. Any activity then takes time δt to complete, where the delay is a random variable taken from an exponential distribution defined by r.

3.1 Syntax

The syntax of the calculus is based on countable sets of names representing communication channels and data, ranged over by lowercase letters *x, y, z, m, n....* Processes are denoted with uppercase letters $A, B, C, ...$, belonging to set $\mathcal{P}$. The operands are defined by the following grammar:

$$\begin{aligned} A ::= &\quad \sum_{i \in I} \alpha_i.A_i \quad | \quad A|A \quad | \quad (\boldsymbol{\nu} x)A \quad | \quad !\alpha.A \\ \alpha ::= &\quad x(y) \quad | \quad \tau_r \quad | \quad \overline{x}(y) \end{aligned}$$

The intuitive meaning of the constructs and prefixes is as follows:

- $\alpha_1.A + \alpha_2.B$ Prefix. Process behaves like either A or B, depending on the action matching the prefix.
- $A|B$. Parallel composition. Process behaves as both A and B.
- $(\boldsymbol{\nu} x)A$. Variable scoping. Variable x is reserved for use only within process A.
- $!\alpha.A$. Replication. Following α, process launches a parallel copy of A, and reverts to its starting state.
- $x^r(y)$. Message input. Process receives name y on channel x with rate r.
- $\overline{x^r}(y)$. Message output. Process outputs name y on channel x with rate r.
- τ^r. Silent action with delay r.

Symbol $\mathbf{0}$ will be used when the size of I in summation is zero. Absence of rate r implies that the channel transition is instantaneous. The rate associated with channel x will sometimes be written as $rate(x)$ to simplify notation. Symbols $\sum_i A_i$, $\prod_i A_i$ and $\bullet_i x_i(y_i)$ are used to denote, respectively, alternative choice, parallel composition of a set of processes, and chain of input message prefixes. The latter represents a series of summations where I is of size 1, and prefix is an input message, informally defined as $\bullet_i x_i(y_i) = x_1(y_1).x_2(y_2).\ ...$

Input $x(y).A$, restriction $\boldsymbol{\nu} y.A$ and output $\overline{x}(y).A$ act as binders, in that all three will bind the variable y in the process A. Therefore if a variable y is considered *bound* in A, denoted $y \in bn(A)$, if A is prefixed by one of these three constructs. Otherwise, y is said to be $free$ in A, denoted $y \in fn(A)$. $A\{z/y\}$ is then the result of replacing free occurrences of y by z in A. A change of bound names, such as replacing $x(y).A$ with $x(z).A\{z/y\}$, where z does not already occur in A, is called α-conversion. Two processes A and B are α-convertible, denoted with $=$, if B can be derived from A through a finite number of α-conversions.

Note that, in order to assist readability, we are using the prefix replication $!\alpha.A$ to represent the replication, in the tradition of standard π-calculus, as opposed to the restricted recursion, used in Phillips and Cardelli [2007]; Phillips et al [2006]. Indeed, the $!\alpha.A$ operator could be expressed in the recursive form as $M = \alpha.(A|M)$.

3.2 Semantics

The semantics of stochastic π-calculus consist of rules that allow a process to evolve into another process with certain delay. Reduction relations define how this is done within a process through static interactions, while transition relations also include evolutions through actions that originate outside the process. Structural congruence of π-calculus allows the transformation of the term-structure that fully preserves its meaning.

Structural congruence is defined in terms of *contexts*. In a π-term, an occurrence of $\mathbf{0}$ is considered degenerate if it is a left or right term in the term $A + B$ and non-degenerate otherwise. For example in the expression $x(y).\mathbf{0} + \mathbf{0}$, the first occurrence is non-degenerate while the second is degenerate. A context is created by replacing a non-degenerate occurrence of $\mathbf{0}$ with the hole [.]. Then, if C is a context, and A a process, $C[A]$ is a process obtained by replacing the hole in C by A.

An equivalence relation R over $\mathcal{P}$ is said to be a congruence if $(A, B) \in R$ implies $(C[A], C[B]) \in R$ for every context C. Structural congruence, denoted

with $\equiv$ is the smallest congruence relation that satisfies the following axioms:

SC-SUM-INACT	$A + \mathbf{0} \equiv A$
SC-SUM-COMM	$A + B \equiv B + A$
SC-SUM-ASSOC	$A + (B + C) \equiv (A + B) + C$
SC-PAR-INACT	$A \mid \mathbf{0} \equiv A$
SC-PAR-COMM	$A \mid B \equiv B \mid A$
SC-PAR-ASSOC	$A \mid (B \mid C) \equiv (A \mid B) \mid C$
SC-RES	$\boldsymbol{\nu} x\ \boldsymbol{\nu} y\ A \equiv \boldsymbol{\nu} y\ \boldsymbol{\nu} x\ A$
SC-RES-INACT	$\boldsymbol{\nu} x\ \mathbf{0} \equiv \mathbf{0}$
SC-RES-PAR	$\boldsymbol{\nu} x(A \mid B) \equiv A \mid \boldsymbol{\nu} x\ B\ x \notin fn(A)$
SC-REP	$!\alpha.A \equiv \alpha.(A \mid !\alpha.A)$

The reduction relation $\longrightarrow$ over $\mathcal{P}$ is the smallest relation satisfying the following rules:

R-COM	$\sum_{i \in I} x_i^{r_i}(y).P_i \mid \overline{x_j^{r_j}}(z) \xrightarrow{r} P_j\{z/y\}$
R-TAU	$\sum_{i \in I} \alpha_i.P_i \xrightarrow{r} P_j\ j \in I\ \alpha_j = \tau^r$
R-PAR	$\dfrac{A \xrightarrow{r} A'}{A \mid B \xrightarrow{r} A' \mid B}$
R-RES	$\dfrac{A \xrightarrow{r} A'}{(\boldsymbol{\nu} x)A \xrightarrow{r} (\boldsymbol{\nu} x)A'}$
R-STRUCT	$\dfrac{A \equiv B\ A \xrightarrow{r} A'\ A' \equiv B'}{B \xrightarrow{r} B'}$

R-COM rule captures the interaction between processes, in which two processes communicate via messages sent along a compatible channel. R-TAU describes the internal transition within a process. R-PAR, R-RES, and R-STRUCT show that the reduction is preserved under parallel composition, variable scoping, and structural congruence, respectively.

As an example of process interaction, let us take two parallel processes, A and B: $x(y).A \mid \overline{x}(z).B$. After the second process sends out the message on channel x, the first process will behave as $A\{z/y\}|B$. So the name x is shared between the two parallel processes.

While reduction governs the activities within the system of interacting processes, it does not describe how the system can interact with the surrounding environment (e.g. processes not being observed). This is described by the *actions* of the system, which correspond to the prefixes of the calculus.

Actions in the calculus are represented as: $\alpha ::= \quad x^r(y) \mid \tau \mid \overline{x^r}(y)$. $\overline{\alpha}$ denotes the complementary action, such that if $\alpha = x^r(y)$, $\overline{\alpha} = \overline{x^r}(y)$, if $\alpha =$

$\overline{x^r}(y)$, $\overline{\alpha} = x^r(y)$ and if $\alpha = \tau^r$, $\overline{\alpha} = \tau^r$. The transition relation $\xrightarrow{\alpha,r}$ is defined by the following rules:

$$\text{ALPHA} \quad \frac{A \equiv A' \quad A' \xrightarrow{\alpha,r} A''}{A \xrightarrow{\alpha,r} A''}$$

$$\text{PRE} \quad \alpha.A \xrightarrow{\alpha,r} A$$

$$\text{PAR} \quad \frac{A \xrightarrow{\alpha,r} A'}{A|B \xrightarrow{\alpha,r} A'|B}$$

$$\text{RES} \quad \frac{A \xrightarrow{\alpha,r} A'}{(\boldsymbol{\nu} x)A \xrightarrow{\alpha,r} (\boldsymbol{\nu} x)A'}$$

$$\text{COM} \quad \frac{A \xrightarrow{\alpha,r} A' \quad B \xrightarrow{\overline{\alpha,r}, r} B'}{A|B \xrightarrow{\tau,r} \boldsymbol{\nu} x(A'|B')} \alpha \neq \tau, x = bn(\alpha)$$

$$\text{SUM} \quad \frac{A_i \xrightarrow{\alpha,r} A_i', i \in I}{\sum_{i \in I} A_i \xrightarrow{\alpha,r} A_i'} i \in I$$

The reduction relation is therefore a transition where $\alpha = \tau$ and the system evolves into another state with no external messages.

4 Process characterisation of Taverna workflows

The process model of a Taverna workflow needs to accurately reflect the thread usage of each processor, relative ordering of executions in the workflow, and times at which each thread start processing incoming streams of data tokens. Only in this way can an accurate simulation be designed to chart the node activity, display saturation points and predict periods of high and low utilisation. To that goal, we adopt the model in which each node's behaviour is characterised by the type of the input channels (data, control, or both), iterative or non-iterative processor behaviour and the number of inputs.

4.1 Design principles

When translating the Taverna graphs into a process model, the following principles are defined:

1. **Each workflow node is a process.** To allow each node to specify its behaviour, and to maintain loose coupling with other nodes, a separate process will be assigned to it.

2. **Node processes are ignorant of each other.** All knowledge a node possesses about its environment is based on its communication channels.

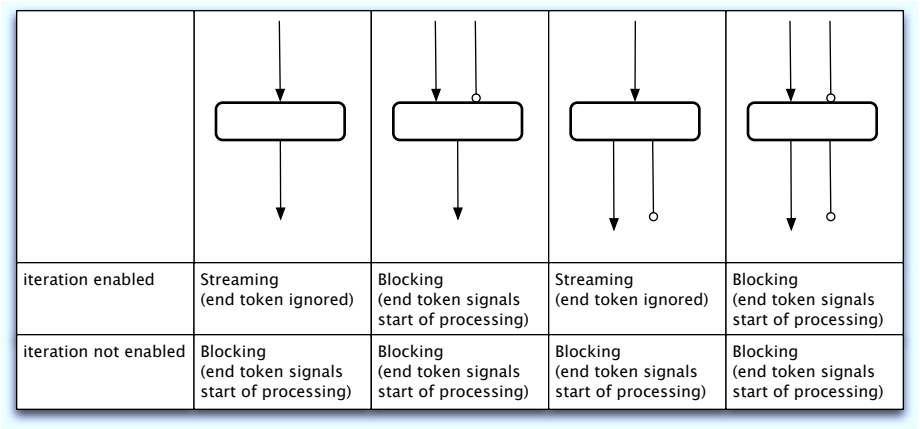
3. **Node execution is a silent action τ.** The node execution, typically a web service or script invocation, is represented by silent action τ. The node execution length is a delay associated with that internal action.

4. **Arcs between the nodes are channels.** The existence of an arc between two nodes implies that there is a reserved communication channel that can be used to pass tokens between those two nodes only.

5. **Compositionality.** The workflow process is the parallel composition of processes representing all the nodes in the workflow, with bound names used for channels. Sequential and other orderings are achieved through nodes waiting on tokens arriving from channels.

These principles are similar to the ones used in Curcin et al [2009], adapted to the Taverna structure and stochastic nature of the calculus.

4.2 Taverna nodes



iteration enabled	Streaming (end token ignored)	Blocking (end token signals start of processing)	Streaming (end token ignored)	Blocking (end token signals start of processing)
iteration not enabled	Blocking (end token signals start of processing)	Blocking (end token signals start of processing)	Blocking (end token signals start of processing)	Blocking (end token signals start of processing)

Figure 2: Single input single output node execution behaviour

The behaviour of a single input, single output Taverna node is defined by the data and control links, and the iterative specification. As described in section 2.2, the nodes with implicit iteration will start processing the data tokens as soon as they start arriving, while the non-iterative nodes will wait for the End-Of-Stream, or stop, token to arrive before commencing with execution. Two behaviours are sufficient to describe these, *Streaming* and *Blocking* reflecting the way in which the data tokens are consumed, the overview of which is shown in Figure 2. Note that the type of output has no impact on the node behaviour, since the results are being sent on the output channel as they become available, followed by the stop token to denote end of execution, leaving all process logic to be handled on the input.

To distinguish between the regular data tokens, and stop tokens, a very simple sort is defined that, when queried whether the token is a stop or a data token, responds with the correct answer.

$$Stop(token) = token(s, d).\overline{s}.Stop(token)$$
$$Data(token) = token(s, d).\overline{d}.Data(token)$$

4.2.1 Streaming single input node

The streaming single input node consists of a processor that receives the new tokens, a queue of data tokens awaiting execution, and a pool of threads that are assigned to data tokens as they become available, executing them and dispatching the output. The presence of the queue ensures that the data items are sent to the threads in order, even though there is no guarantee that the outputs will be produced in the same order, due to the variability of execution times. Stop token is added to the execution queue like any other, however the $SQNode$ process recognizes it when it gets asked to send the execution request and, instead, notifies the processor with the *terminate* message, which sends the *halt* signal to all threads (ensuring they complete), before sending the stop token on the output. The guaranteed order of items in the queue ensures that once the stop token has been recognized, no other data token is waiting for execution and processors can be safely halted.

$$StreamingNode(a, b) = (proc, k) \prod_{i=1..k} Thread_i(proc) \mid StreamingProcessor(a, b, proc)$$
$$\mid SQueue(append, proc)$$
$$StreamingProcessor(a, b, proc) =$$
$$!a(x).\overline{append}(x) \mid terminate(z).\overline{halt}^k.\overline{b}(z)$$
$$SQueue(append, proc) = (\boldsymbol{\nu} next)\ append(x).SQueueNode(append, next, proc, x)$$
$$\mid \overline{next}$$
$$SQueueNode(append, activate, proc, x) =$$
$$(\boldsymbol{\nu} next)\ append(y).SQueueNode(append, next, proc, y)$$
$$\mid activate.\overline{x}(s, d)(d.(\overline{proc}(x).\overline{next})) + s.\overline{terminate}(x)$$
$$Thread(proc) = proc(x, out).\tau_r.(\boldsymbol{\nu} y)\ \overline{out}(y).Thread(proc) + halt.\mathbf{0}$$

The node definition is in terms of input channel a, output channel b, and internally parameterized on processing request channel *proc* and number of threads k. Note that the queue is ephemeral, in the sense that the item execution destroys that element, allowing the next element to use channel *proc* to communicate with the next available $Thread$.

4.2.2 Blocking single input node

The blocking node behaviour occurs when the node needs to wait for the stop token before starting execution, either due to the presence of an input control

link, or due to non-iterative execution behaviour of the node. Therefore, the arriving input data tokens are stored in a request queue, with the contents being sent to threads only after the stop token has been received. Once this happens, an *activate* message is sent to start execution, and, as with the streaming node, *terminate* denotes that all data tokens have been sent to threads and *halt* requests can be sent.

$$BlockingNode(a,b) = (proc,k,terminate) \prod_{i=1..k} Thread_i \mid BlockingProcessor(a,b,proc)$$
$$\mid BQueue(append,activate,proc)$$
$$BlockingProcessor(a,b,proc) =$$
$$!a(x).\overline{append}(x).\overline{x}(s,d).(d.\mathbf{0} \; + \; s.\overline{activate}) \mid terminate.\overline{halt}^{k}.\overline{b}(x)$$
$$BQueue(append,activate,proc) = (\boldsymbol{\nu} next)\ append(x).BQueueNode(append,next,proc,x)$$
$$\mid activate.\overline{next}$$
$$BQueueNode(append,activate,proc,x) =$$
$$(\boldsymbol{\nu} next)\ append(y).BQueueNode(append,next,proc,y)$$
$$\mid activate.\overline{x}(d.(\overline{proc}(x).\overline{next})) + s.\overline{terminate}$$
$$Thread(proc) = proc(x,out).\tau_r.(\boldsymbol{\nu} y)\ \overline{out}(y).Thread(proc) + halt.\mathbf{0}$$

4.2.3 Dual input node

The dual input streaming node maintains two input queues as data is coming in. The key difference is that the queues will have multiple traversals performed on them, and therefore need to be persistent, unlike the simple single input request queue.

$$DStreamingNode(a,b,c) = (appendA,appendB,proc) \prod_{i=1..k} Thread_i(proc)$$
$$\mid DStreamingProcessor(a,b,c,proc)$$
$$\mid DSQueue(appendA,execA,proc) \mid DSQueue(appendB,execB,proc)$$
$$DStreamingProcessor(a,b,c,proc) = !(a(x).\overline{appendA}(x).\overline{executeB}(x) +$$
$$b(x).\overline{appendB}(x).\overline{executeA}(x))$$
$$\mid terminate.\overline{halt}^{k}.\overline{c}(x)$$
$$DSQueue(append,exec,proc) = (\boldsymbol{\nu} next)\ append(x).DSQNode(append,next,proc,x)$$
$$\mid !exec(y).\overline{next}(y)$$
$$DSQNode(append,exec,proc,x) = (\boldsymbol{\nu} next)\ append(y).DSQNode(append,next,proc,y)$$
$$\mid !exec(y).\overline{y}(s,d)$$
$$(s.\overline{x}(s,d).\ (d.\mathbf{0} + s.\overline{terminate}(x)) + d.\overline{x}(s,d).\ (d.\overline{proc}(x,y).\overline{next}(y) + s.\mathbf{0}))$$
$$Thread(proc) = proc(x,y).\tau_r.(\boldsymbol{\nu} m)\ \overline{out}(m).Thread(proc) + halt.\mathbf{0}$$

The stop tokens are added to the queues in the same manner as normal tokens, however the queue node will not send a request involving a stop token to the thread, and will ignore it. If the request should contain two stop tokens, that indicates that the execution is at the end, and a *terminate* signal is sent.

The blocking variety is identical, except that it adds an additional *start* message that the threads expect before they can start any execution.

$$DBlockingNode(a,b,c) = (appendA, appendB, proc, start) \prod_{i=1..k} start.Thread_i(proc)$$

$$| \; DBlockingProcessor(a,b,c,proc)$$

$$| \; DBQueue(appendA, execA, proc) \mid DBQueue(appendB, execB, proc)$$

$$DBlockingProcessor(a,b,c,proc) = !(a(x).\overline{appendA}(x).$$

$$\overline{x}(s,d)(d.\mathbf{0} + s.\overline{activateA}(executeB)) + b(x).\overline{appendB}(x).\overline{x}(s,d)(d.\mathbf{0} + s.\overline{activateB}(executeA))$$

$$| \; terminate.\overline{halt}^k.\overline{c}(x)$$

$$| activateA.activateB.\overline{start}^k$$

$$DBQueue(append, exec, proc) = (\boldsymbol{\nu} next)\; append(x).DBQNode(append, next, proc, x)$$

$$| \; !exec(y).\overline{next}(y)$$

$$DBQNode(append, exec, proc, x) = (\boldsymbol{\nu} next)\; append(y).DBQNode(append, next, proc, y)$$

$$| \; !exec(y).\overline{y}(s,d).\; (d.\overline{proc}(x,y).\overline{next}(y) + s.\overline{x}(s,d).\; (d.\mathbf{0} + s.\overline{terminate}(x)))$$

$$Thread(proc) = proc(x,y).\tau_r.(\boldsymbol{\nu} m)\; \overline{out}(m).Thread(proc) + halt.\mathbf{0}$$

4.2.4 Data source node

The data source node is a simple token producer, that generates data items in sequence, sending them out on all of its channels. A single output channel variant, outputting k data items on channel *out* has the form:

$$DataSource(out, k) = \bullet_{i=1..k} (\boldsymbol{\nu} d)\overline{out}(d)$$

The general variety, given n output channels, takes the form:

$$DataSource(out_1, ..., out_n, k) = \prod_{i=1..n} \bullet_{j=1..k} (\boldsymbol{\nu} d)\overline{out_i}(d)$$

4.2.5 Example: Simple workflow

The workflow used in this example is depicted in Figure 1(page 5). It represents a data producer, sending data into two blocking nodes, that are connected to two further blocking nodes that send their outputs into a dual input streaming node which starts processing the data tokens as soon as they arrive. The parameters to the workflow are the size of the input list and the number of items in each list.

The workflow has seven bound channels (p, q, r, s, t, u), and one free channel v (that can communicate with outside entities), is represented using the following formula:

$$W(v) = (\boldsymbol{\nu}\ p, q, r, s, u)(LG(p, q) \mid A0(p, r) \mid A1(r, t) \mid B0(q, s) \mid B1(s, u) \mid F(t, u, v))$$
$$LG := DataSource$$
$$A0 := BlockingNode$$
$$A1 := BlockingNode$$
$$B0 := BlockingNode$$
$$B1 := BlockingNode$$
$$F := DualStreamingNode$$

The workflow is considered terminated after there are no more tokens being passed around the system, and therefore no more events which can cause the change of process states.

5 Use case: simulating a workflow

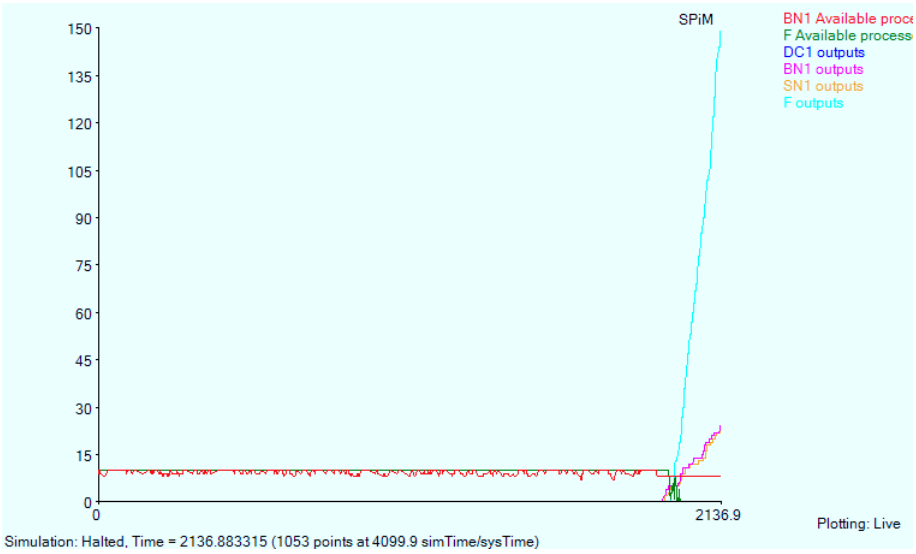


Figure 3: Simulation run of the example workflow

In order to run the workflow simulation, such as shown in Figure 3 there are several steps to be performed. Firstly, the SCUFL document needs to be translated into the process model. Then the execution times need to be entered for each node - either manually or based on performance logs. Finally, the

code is generated for SPiM engine. Since the SCUFL workflow representation already contains the processor information, number of threads for each node and the type of channels between the processors, the preprocessor extract these directly from the XML document saved by Taverna. As a next step, the actual depths of each input, as described in section 2.2 are calculated and compared to declared depths of the processor, specifying the iterative behaviour. The average execution times are not currently stored in the representation, although there are plans to add them as annotations to the document, so presently they are manually added.

The SPiM representation is a direct mapping of the process model, with some simplifications added. Most notably, since SPiM offers a basic set of data structures, individual queues of data tokens are replaced by simple counters. The translation of a streaming node is shown below:

```
let NodeStreaming(inp:chan(token),
          out:chan(token),
          requests:int,
          halted:bool,
          thread_chan:chan(chan)) =
   if (requests>0) then (
(*receive a new item and place it in the queue,
distinguish between stop and data token*)
   do ?inp(t);match(t)
              case("stop")
              NodeStreaming(inp, out, requests, true, thread_chan)
              case("data")
              NodeStreaming(inp, out, requests+1,halted,thread_chan)

(*move the item from the queue into an execution thread*)
   or !thread_chan(out);NodeStreaming(inp, out, requests-1, halted, thread_chan)
)   else if (requests = 0 and -halted) then (
       ?inp(t);match(t)
                case("stop") NodeStreaming(inp, out, requests, true, thread_chan)
                case("data") NodeStreaming(inp, out, requests+1,halted,thread_chan)
)   else !out("stop")

and Thread(proc:chan(chan), r:float) = ?proc(A);delay@r;(!A | Thread(proc,r))
```

5.1 Thread utilization

Tracing the number of threads used in a particular node is accomplished by simply observing the number of instances of *Thread* processes for that node. In the first graph shown in Figure 4, we can see how the number of processors for $B0$ node varies over time and then stays constant as all data is processed. Note that at no point are all ten processors used. However, increasing the execution time of the task leads to the behaviour depicted in the second graph, with the

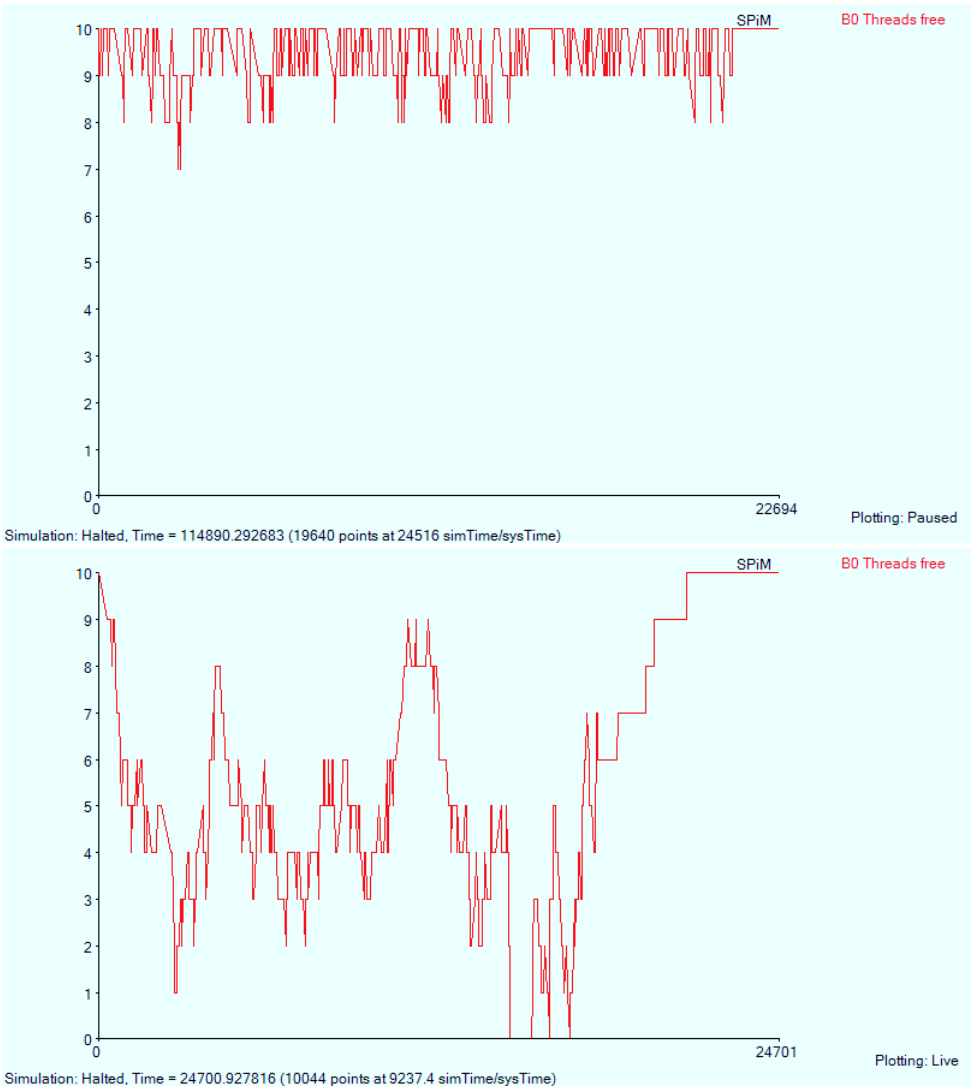


Figure 4: Available threads for B0 node, at varying task lengths.

resources being fully utilized.

5.2 Intermediate node output

Another interesting detail is when and how the output is produced. In the graph shown in Figure 5, the output of node $A1$ is shown. The node, as expected, only starts producing output once $A0$ has finished with execution and sent the

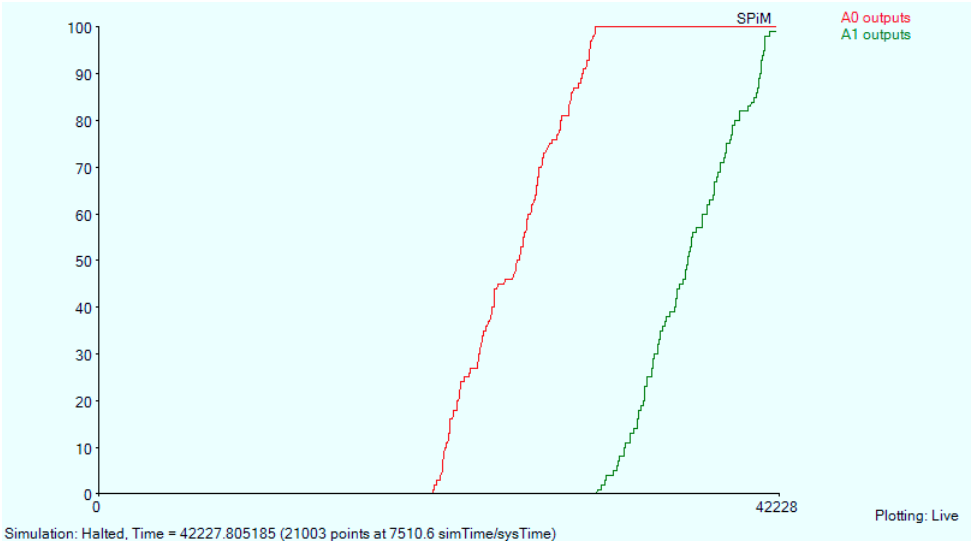


Figure 5: Output production of nodes A0 and A1

stop token onwards.

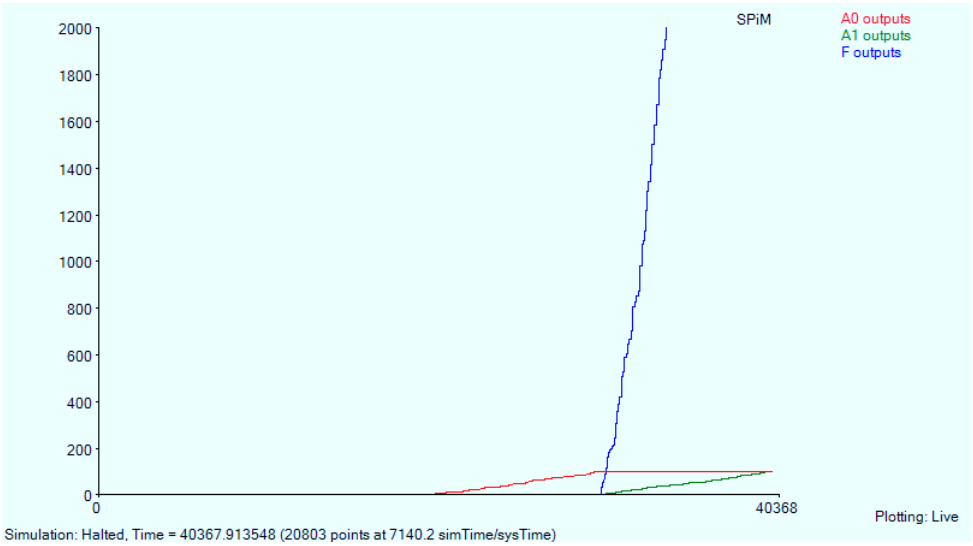


Figure 6: Output production of node F

The diagram in Figure 6 shows the output of terminating node F. As this is

a dual input node that has no block on the inputs, it starts performing the cross-product of inputs received from $A1$ and $B1$ as soon as these become available, reflected by the F outputs being produced in parallel to the $A1$ ones in the graph.

6 Summary

This paper presented a process model for streaming execution semantics in Taverna, using stochastic process algebra to capture the execution behaviour of the workflow nodes, based on the types of link inputs, and their iterative behaviour. The work done has been demonstrated in constructing a simulator for the workflows, using the SPiM tool, and can be developed further for alternative types of performance analysis, adaptation to other models, and for comparison with other tools.

6.1 Future work

The model introduced here can be developed in several directions. Firstly, before it can be incorporated into an analysis tool, a mechanism is needed to generate ensemble averages of the simulation runs, to provide more reliable predictions. Secondly, a more accurate simulation can be obtained by characterising the channels between the processor ports with transfer rates. While an obvious feature in any distributed execution setting, Taverna is still not capturing this information. Finally, developing a model checker based on CSL, or a similar logic, would enable static model checking on workflows, in the style of PRISM. This requires resolving the state problem in such a way to still characterise the data/stop token coordination, but group similar states together, possibly through statistical modelling of aggregated rate transitions of grouped states. This is conceptually similar to the *lumping* technique popular in performance tools.

6.2 Acknowledgments

The authors would like to thank Stian Soiland-Reyes for advice on Taverna processor behaviour and Richard Hayden on useful information about the PEPA tool.

References

Aziz A, Sanwal K, Singhal V, Brayton R (2000) Model-checking continuous-time markov chains. ACM Trans Comput Logic 1(1):162–170, DOI http://doi.acm.org/10.1145/343369.343402

Bergstra JA, Klop JW (1989) Acpτ: a universal axiom system for process specification. In: Wirsing M, Bergstra JA (eds) Algebraic methods: theory, tools

and applications, Springer-Verlag New York, Inc., New York, NY, USA, pp 447–463

Bernardo M, Donatiello L, Gorrieri R (1994) Mpa: a stochastic process algebra. Tech. rep.

Briais S (2009) ABC – Another Bisimulation Checker. Http://lamp.epfl.ch/ sbriais/abc/, Last accessed, May 2009

Curcin V, Ghanem MM, Guo Y (2009) Analysing scientific workflows with computational tree logic. Cluster Computing 12(4):399–419, DOI http://dx.doi.org/10.1007/s10586-009-0099-6

Gilmore S, Hillston J (1994) 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, Springer-Verlag, Vienna, no. 794 in Lecture Notes in Computer Science, pp 353–368

Goderis A, Brooks C, Altintas I, Lee EA, Goble CA (2007) Composing different models of computation in Kepler and Ptolemy II. In: Shi Y, van Albada GD, Dongarra J, Sloot PMA (eds) International Conference on Computational Science (3), Springer, Lecture Notes in Computer Science, vol 4489, pp 182–190

Gottschalk F, van der Aalst WMP, Jansen-Vullers MH, Verbeek HMW (2007) Protos2cpn: using colored Petri Nets for configuring and testing business processes. International Journal on Software Tools for Technology Transfer 10(1):95–110

Hermanns H, Herzog U, Mertsiotakis V, Rettelbach M (1995) Stochastic process algebras – constructive specification techniques integrating functional, performance and dependability aspects. In: Quantitative Methods in Parallel Systems, Springer

Hillston J (2005) Fluid flow approximation of pepa models. In: Proc. 2nd International Conference on Quantitative Evaluation of Systems (QEST'05), IEEE Computer Society Press, pp 33–42

Hoare CAR (1983) Communicating sequential processes. Commun ACM 26(1):100–106

Hull D, Wolstencroft K, Stevens R, Goble CA, Pocock MR, Li P, Oinn T (2006) Taverna: a tool for building and running workflows of services. Nucleic Acids Research 34:729–732

Huth MRA, Ryan MD (2000) Logic in Computer Science: Modelling and reasoning about systems. Cambridge University Press, Cambridge, England, URL `citeseer.ist.psu.edu/huth99logic.html`

Johnston WM, Hanna JRP, Millar RJ (2004) Advances in dataflow programming languages. ACM Comput Surv 36:1–34, DOI http://doi.acm.org/10.1145/1013208.1013209

Kwiatkowska M, Norman G, Parker D (2009) Prism: Probabilistic model checking for performance and reliability analysis. ACM SIGMETRICS Performance Evaluation Review 36(4):40–45

Microsoft (2007) Spim viewer. Http://research.microsoft.com/en-us/projects/spim/. Last accessed, May 2010.

Milner R (1989) Communication and Concurrency. Prentice–Hall

Milner R (1990) Functions as processes. In: Paterson MS (ed) Automata, Languages and Programming: Proc. of the 17th International Colloquium, Springer, New York, pp 167–180

Missier P, Paton N, Belhajjame K (2010) Fine-grained and efficient lineage querying of collection-based workflow provenance. In: Procs. EDBT, Lausanne, Switzerland

Moller F, Stevens P (2009) Edinburgh Concurrency Workbench user manual (version 7.1). Available from http://homepages.inf.ed.ac.uk/perdita/cwb/. Last accessed May 2009

Peschanski F, Hym S (2006) A stackless runtime environment for a pi-calculus. In: VEE '06: Proceedings of the 2nd international conference on Virtual execution environments, ACM, New York, NY, USA, pp 57–67

Phillips A, Cardelli L (2004) A correct abstract machine for the stochastic pi-calculus. In: Concurrent Models in Molecular Biology

Phillips A, Cardelli L (2007) Efficient, correct simulation of biological processes in the stochastic pi-calculus. In: Computational Methods in Systems Biology, Springer, LNCS, vol 4695, pp 184–199

Phillips A, Cardelli L, Castagna G (2006) A graphical representation for biological processes in the stochastic pi-calculus. Transactions in Computational Systems Biology 4230:123–152

Priami C (1995) Stochastic pi-calculus. Comput J 38(7):578–589

Rockl C, Hirschkoff D, Berghofer S (2001) Higher-Order abstract syntax with induction in Isabelle/HOL: Formalizing the pi-calculus and mechanizing the theory of contexts. In: Proceedings of Conference on Foundations of Software Science and Computational Structures, pp 364–378

Sroka J, Hidders J, Missier P, Goble C (2009) Formal semantics for the Taverna 2 Workflow Model. Journal of Computer and System Sciences DOI 10.1016/j.jcss.2009.11.009, URL `http://dx.doi.org/10.1016/j.jcss.2009.11.009`

Turi D, Missier P, Roure DD, Goble C, Oinn T (2007) Taverna Workflows: Syntax and Semantics. In: Proceedings of the 3rd e-Science conference, Bangalore, India, DOI http://dx.doi.org/10.1109/E-SCIENCE.2007.71, URL http://dx.doi.org/10.1109/E-SCIENCE.2007.71