Using Stochastic Petri Nets to Predict Quality of Service Attributes of Component-Based Software Architectures

Jens Happe, Viktoria Firus
Software Engineering Group*
University of Oldenburg
[jens.happe|viktoria.firus]@informatik.uni-oldenburg.de

Abstract

The Quality of Service attributes of a software component heavily depend on its environment. For example, if a component uses a highly unreliable service, its own reliability is likely to decrease as well. This relation can be described with parametric contracts, which model the dependence between provided and required services of a component. Until now, parametric contracts can only model single-threaded systems. We plan to extend parametric contracts with Stochastic Petri nets to model multi-threaded systems. This enables the detection of resource conflicts and the consideration of the influence of concurrency on Quality of Service attributes, like performance.

1 Introduction

In industrial software applications, poor performance and low reliability directly translates into financial losses. Despite its importance, Quality of Service (QoS) attributes are only considered during late development stages when at least parts of the system are implemented. This 'fix-it-later' attitude [14] often leads to expensive refactorings of the system, since many QoS issues are based on design decisions made during early development stages of the project. To prevent these types of mistakes, accurate QoS prediction models are required. These models should be able to evaluate the QoS attributes of a software system during early stages of the development. They should allow the evaluation of different design alternatives and guide the software architect.

In many cases, this requires a lot of additional effort from the software architect and, therefore, is unattractive. The advantages of component-based software architectures can ease this task: The specifications of existing components can be reused in a new deployment context. This reduces the overhead of specification for the component deployer. Additionally, predefined components limit the degrees of freedom of the implementation and can provide additional knowledge. The compositional structure of component-based software architectures allows the prediction of QoS attributes for different levels of abstraction [5].

Our approach is based on parametric contracts, which model the dependence of provided and required services of a component. Prediction models based on parametric contracts, which have been proposed so far, use automata-based models to describe the protocol of services [11, 12, 3]. These models lack the expressive power to analyse multi-threaded systems, which results in the following problems for QoS prediction:

Multiple Threads: Automata-based models, like finite state machines, can only be used to describe protocols with a single thread. They neglect the creation of new threads during the execution of a service, which might has a major impact on the QoS attributes of the architecture as discussed in section 3. Since threads are often used in programming, their influence has to be considered in QoS prediction models.

Resources: In multi-threaded systems, resources, like the hard disk or the network connection, are shared among all running threads. The fact that a thread is blocked as log as it cannot access a required resource influences the performance of the system. Thus, the competition for resources has to be considered in a QoS prediction model for multi-threaded systems.

To cope with multi-threaded systems, the current automata-based approach of parametric QoS contracts needs to be extended. Therefore, we propose the usage of Stochastic Petri nets (SPN) to model the externally visible

*This work is supported by the German Research Foundation (DFG), grant GRK 1076/1 (Graduate School Trustsoft)

protocol of a service. This allows the description of multi-threaded systems in a convenient way and enables the recognition of resource conflicts.

The contribution of this paper is the discussion of the extension of parametric contracts with SPNs. Therefore, we illustrate problems of the automata-based approach when modelling multi-threaded systems and show how SPNs can be used to overcome these problems. Furthermore, we discuss the limitations of SPNs as a modelling formalism for concurrent applications.

This paper is structured as follows. Section 2 introduces parametric contracts for the QoS attributes of components. Section 3 discusses the limits of the automata-based illustrated with the example of a web server. Section 4 introduces stochastic Petri nets and illustrates how these can be used for parametric contracts. Section 5 summarises the open issues of our approach. Section 6 concludes this paper.

2 Parametric QoS Contracts

Component contracts lift the Design-by-Contract principle of Meyer ("if a client fulfils the precondition of a supplier, the supplier guarantees the postcondition" [9]) from methods to software components. A component guarantees a set of provided services if all required services are offered by its environment. Design-by-Contract cannot only be applied to functional properties, but also to non-functional properties. Adding Quality of Service (QoS) attributes to the contract, a component service ensures that it provides a certain QoS if its environment offers the required QoS properties.

Static QoS contracts have the drawback, that the component developer cannot foresee all possible reuse contexts of his components in advance. Even if the required quality attributes of a component are not satisfied, the component is likely to work (with an adjusted quality). The authors of [12] come to the conclusion that: "We do not need statically fixed pre- and postconditions, but *parametric contracts* to be evaluated during deployment-time."

Service effect specifications are required to describe intra-component dependencies of provided and required services. A service effect specification models external service calls executed by a component service. This can be done by a signature list or, if the execution order of the external services is important, by the set of call sequences invoked by the service. Regular expressions, finite state machines, or Petri nets are possible methods to describe an infinite set of call sequences [10]. Service effect specifications enable the computation of QoS attributes of the provided services (postcondition) in dependence on the QoS attributes of the required services (precondition). For QoS, the computation of the required attributes yields a solution space and, therefore, its benefits are questionable.

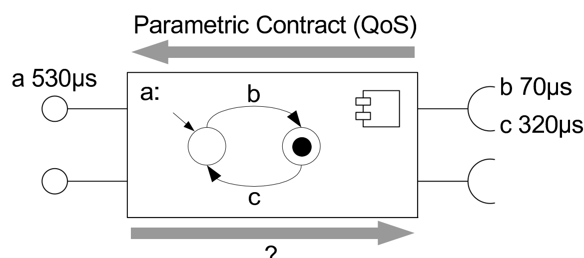

Figure 1: Parametric Component Contract.

Figure 1 illustrates parametric contracts for the QoS attribute execution time. The component shown there provides a single service called `a`, which requires two services `b` and `c`. The execution time of `a` can be calculated from the execution times of `b` and `c`. The computation requires the service effect specification of `a`, which is depicted in the component. As mentioned above, the computation of the parametric contract is only applicable in one direction.

Parametric contracts have already been used to predict quality attributes of component-based software architectures, for example, reliability [12] and performance [3]. In those works Markov models were used for the description of the service effect specification.

3 Limits of the Automata-Based Approach

The models proposed in [11, 12, 3] use an automata-based approach to predict the performance and/or reliability of a component-based software architecture. The service effect specification is given as a Markov model. A Markov model can be seen as a finite state machine, whose transitions are annotated with a probability of taking the transition from its source state. These models can be appropriate when dealing with single-threaded systems. For these cases, the analysis of Markov models is relatively simple and yields realistic estimations. However, as soon as a software system with multiple execution threads has to be analysed, different influences come into play, which can hardly be expressed by finite state machines or the corresponding Markov model.

```
public void Listen()
{
  TcpListener tcpListener = new TcpListener(address, port);
  tcpListener.Start();
  while (IsRunning)
  {
    Socket clientSocket = tcpListener.AcceptSocket();
    IRequestHandler handler = CreateRequestHandler(port, clientSocket);
    thread = new Thread(new ThreadStart(handler.HandleRequest));
    thread.Start();
  }
  tcpListener.Stop();
}
```

Figure 2: Simplified Version of the Method `Listen` of the Web Server.

To illustrate these difficulties, we use the example of a web server developed by our group. The web server consists of several components. One of these, the so-called dispatcher, listens on the network connection and activates the handling of incoming requests. This is done in the method `Listen` of the dispatcher component. Figure 2 shows a simplified version of the method's code. First, a new `TcpListener` object is created. It gets the address and port the web server is listening on as input parameters. After that the `TcpListener` is started. While the web server is running, the `TcpListener` accepts incoming sockets and creates a new handler for each request. Since it is very likely that several requests arrive simultaneously and since requests must be handled as quick as possible, the dispatcher starts a new thread for each incoming request and continues to listen on the network connection.

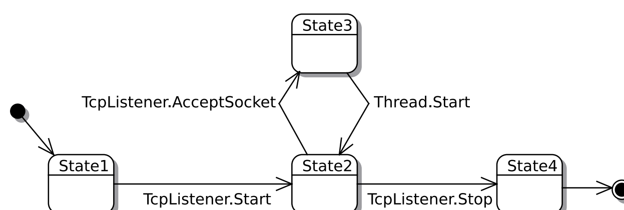

Figure 3: Straight Forward Service Effect Specification of the Method `Listen`.

The creation of a new thread cannot be expressed with finite state machines. The direct translation of the source code shown in figure 2 would ignore the input parameter of the constructor of the class `Thread` and yield a service effect specification that neglects the execution of the method `IRequestHandler.HandleRequest`. The result, depicted in figure 3, does not correspond to the source code shown in figure 2. However, it includes the start of a new thread, but does not consider its impact on the performance and reliability.

Another, probably more intuitive way to model the externally visible behaviour of the service `Listen` is shown in figure 4. Here, the creation of a new thread is omitted. Instead, the call to the service `IRequestHandler.HandleRequest` is directly modelled. This maps the multi-threaded web server onto a single-threaded version. The predictions made by this transformed model of the web server are realistic as long as the number of concurrent requests is low. With a higher arrival rate of requests, the predictions become more and more inaccurate.

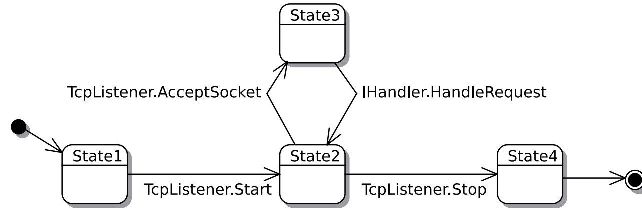

Figure 4: Service Effect Specification of the Method `Listen`.

In the first place, the introduction of threads might increase the performance compared to a single-threaded web server. However, the growth in performance is not unbounded and, if a certain number of threads is exceeded, the performance will decrease again [8, p. 49]. One of the reasons for this is the competition for resources. With a higher number of threads, it is more likely that a resource a single thread is about to access is used by another thread. Furthermore, the overhead of the scheduler increases with a higher number of threads.

For the automaton-based approach, the system load indirectly influences the computation of the service execution times. The execution times of all states and transitions of the service effect specification are estimated or measured. So, the information about the system load is encoded in the execution times. A higher system load simply yields a longer execution time of the states and transitions. This has the drawback that the model cannot be used to predict the performance for different system loads. For example, if one wants to predict the service execution times for a higher arrival rate of requests, new estimations and/or measurements of the execution times are required.

In this paper, we propose the usage of Stochastic Petri nets as a solution to the problems discussed above. This formalism can be used to model multi-threaded software systems, to detect resource conflicts, and to specify the system load at distinct points (*e.g.* as arrival rate of requests). The choice of Petri nets was motivated by the following reasons: (a) Petri nets are a graphical notation with formal semantics, (b) the state of a Petri net can be modelled explicitly, which is not possible for process calculi, and (c) the availability of many analysis techniques for Petri nets [15]. Other description techniques of multi-threaded systems, like process calculi, might be considered in our work as well. However, there is a close relation between Petri nets and process calculi [13].

4 Applying Stochastic Petri Nets on Component-Based Software Architectures

4.1 Modelling Concurrency with Petri Nets

Petri nets allow to model systems with several execution threads. The usage of Petri nets as service effect specifications can be seen as an extension of the automata-based approach, since a finite state machine can be transformed into a Petri net with a single token.

In general, the definition of Petri nets does not consider any timing behaviour or (possible) transition probabilities. Therefore, we use Stochastic Petri nets that associate an exponentially distributed execution time with each transition. The continuous-time Stochastic Petri net $S_N = (N, \Lambda)$ is formed from the Petri net N by adding the set $\Lambda = (\lambda_1, \ldots, \lambda_m)$ to the definition. λ_i is the, possibly marking dependent, rate of transition t_i, by which the exponential distribution is specified [2].

Figure 5 shows the service effect specification of the service `Listen` as a Petri net with its initial marking. Here, each token of the net represents a thread instance. Place p_4 contains five tokens and can be considered a thread pool that stores the idle threads. The Petri net is an extension of the finite state machine shown in figure 3. If the places p_3, p_4 and the transition `IRequestHandler.HandleRequest` are omitted, the reduced net is equivalent to the finite state machine shown in figure 3. The additional part introduces multi-threaded behaviour to the system.

After the acceptance of an incoming connection, one token is placed on p_2. Now transition `Thread.Start` is activated and can fire. If it does so, one token is removed from p_4 and p_2 and a token is placed on p_1 and p_3. The placement of a token on p_1 represents the reiteration of the loop by the main thread. The placement of a token

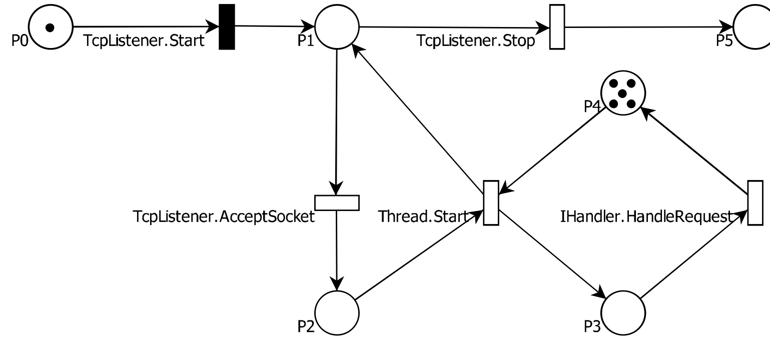

Figure 5: Petri Net for the Service `Listen` with a Maximum of Five Threads.

on p_3 indicates the assignment of an idle thread from the pool (p_4) to the incoming request. Thus, the number of tokens in place p_4 in the initial marking limits the number of parallel threads. The usage of a thread pool will affect the performance of the system if the number of concurrent request is higher than the number of available threads. As in reality, threads are a limited resource.

For the analysis, the reachability graph of a Petri net has to be constructed. In the case of SPNs, it is represented by a continuous-time Markov chain. The state space of the reachability graph of the Petri net in figure 5 is bounded. However, it is relatively simple to construct a net that allows a theoretically unlimited number of threads and whose state space is unbounded. This leads to a state space explosion of the reachability graph. For this reason, simplification techniques are required to be able to perform a formal analysis of the Petri net. Several techniques have been proposed to cope with the state space explosion. These include the decomposition of the net into subnets, the usage of symmetries of the net, the introduction of a hierarchy, and the fixation of upper bounds [1].

4.2 Compositionality and Analysis of Prediction Models

Our approach originates from component-based software architectures and, therefore, provides a compositional structure. For component models, compositionality means that all properties attached to a component should be present for a composite component as well (i.e., there is not difference between a basic and a composite component when neglecting the inner structure) and, in addition, the properties of a composite component should be derivable from the properties of the inner component plus the composition structure. We distinguish between two types of compositionality: The first type concerns the prediction model itself. For example, if the model is compositional, it should be possible to determine the service effect specifications of a composite component from the service effect specifications of its basic components and the composition structure.

The second type of compositionality concerns the analysis of the model. The component acts as a transformer, which gets a set of input parameters and transforms them into a set of output parameters of the same class. Thus, the output parameters of the component can be used as input parameters of another component, which performs the same computation. For example, we can compute the execution times of the services of component `A` in dependence on the execution times of its external services. The resulting execution times can be used as external service execution times of component `B`. The advantage of the computational compositionality is that the results of the computation of one component can be used as inputs for another component, since they are in the same class. This eases the computation and allows the usage of different analytical methods in the same analysis. This is important, since not every component can be analysed in the same way.

The QoS prediction for software components can be performed either by a mathematical analysis or simulation. A mathematical analysis might provide more accurate results, but might not be feasible in some cases. On the other hand, a simulation is always possible, but its results are less exact. Since no real time constraints exist for the application scenario of our approach, simulation would be acceptable. However, it should be possible to handle most parts analytically and, therefore, provide more accurate results. To achieve the highest possible accuracy and to develop a model that can be applied in general, we propose a hybrid approach that analyses the QoS attributes mathematically if possible, and simulates the other parts. This requires a mathematical compositionality of the prediction model.

4.3 Detection of Resource Conflicts with Petri Nets

The detection of global resource conflicts is a challenge for the development of a compositional approach. On the one hand, we do not want to create a composite SPN that describes the behaviour of the whole system. On the other hand, we need enough information about the usage of a global resource to detect conflicts and/or bottlenecks. In [4], we demonstrated that the usage time of a resource can be computed in a compositional way. The only parameters required from the service effect specification are the probability of using the resource and the time the resource is used in this case. However, the computations were only performed for single-threaded applications. It needs to be evaluated if the same (or similar) abstractions are possible in the multi-threaded case.

Furthermore, it has to be considered that resources can be modelled at different abstraction levels. For example, on a very detailed level a network connection is a blocking resource. Only one single thread can read or write data at a single point in time. On a more abstract level, which measures in larger time intervals, the network connection is not blocking: If several threads access the network connection at the same time, this slows down the execution time of every single thread. We want to apply this distinction of different abstraction levels to ease the computation of the overall performance. For a high level resource, the execution time depends on the number of tokens in a certain place. On the other hand low level resources are modelled as blocking entities.

4.4 Possible Simplifications of the Prediction Model

One basic simplification of SPNs was made by Marsan *et. al.* They introduced Generalized Stochastic Petri nets (GSPN) which omit the execution times of transitions that are significantly smaller than the average. For a formal analysis, SPNs and GSPNs are transformed to continuous time Markov chains. The introduction of immediate transitions which do not consume time allows the reduction of the embedded Markov chain, since only transitions that consume time have to be considered. This reduces the computational effort.

The SPNs used in our model describe the externally visible protocol of a service. We assume that every service terminates. Therefore, every SPN must have a non-empty set of final markings, which is reached after a certain period. This deviates from the common usage of SPNs, since, in general, they model infinite stochastic processes. The consideration of terminating processes might ease the performance prediction of the model. Until now, it is unknown how this affects the computation.

4.5 How Do We Get the Required Data?

Our main aim is the decision support of the system architect. However, our approach requires additional data on the component-based architectures from the component developer and deployer. To ease the task of specifying these data and to make our model more attractive for the system architect, we need (at least) a semi-automated tool support for the specification of the service effect, which requires as little additional effort from the component developer and deployer as possible.

The first idea is the generation of Petri nets from the component source code. This is not possible in every case, but might simplify the specification task in most cases. For the example source code in figure 2, the start of a new thread is directly associated with the execution of the method `thread.Start`. The name of the service executed by the thread is assigned during the construction of the thread, which can be statically analysed. The resulting net is similar to the one shown in figure 5 omitting place p_4, which represents the thread pool in our example.

Limited resources, like a thread pool or a network connection, and their behaviour have to be modelled additionally. Information about resources can be obtained from the deployment diagram of the software architecture. The usage of the knowledge already specified in other design documents (include the source code of already implemented components) eases the specification of the required data. However, additional information, like the execution time and the reliability of external services or characteristics of the middleware, are still required from the component developer and deployer.

5 Open Issues

Above, we discussed the usage of SPNs as service effect specification for parametric contracts. However, several important issues have to be considered when reasoning about QoS predictions for component-based software architectures with SPNs.

Limitation to the Exponential Distribution: SPNs and GSPNs introduced in [2, 6, 7] use the exponential distribution to model the time consumption of a service. Our computations performed in [3, 4] showed that the execution time computed for a service cannot be approximated with an exponential distribution, even if all input execution times were exponentially distributed. For the creation of a compositional model, it is necessary that the output parameters conform to the input parameters to pass them to the next component. Thus, a Petri Net model that is able to deal with arbitrary distribution functions is needed. Possible approaches are discussed in [1]. The SPNs with arbitrary distribution functions listed there are non Markovian and can only be applied in certain cases.

Transition Probabilities: A further useful extension of SPNs is the specification of the transition probabilities. They have already been introduced for the immediate transitions of GSPNs. For timed transitions the probability of taking a transition is computed in dependency of its firing rate. It might be required to model the transition probability independent of the execution time.

Arrival Rates of Requests: As seen in the example above, arrival rates are still implicitly modelled. However, there is an advantage to the formerly introduced model where the arrival rate or system load influenced the execution time of all states and transitions. In the example, the arrival rate only affects one single transition. It would be good to separately specify the arrival rate in the operational profile of the system.

Time Consumption of States: Another question that arises when talking about the time consumption of a service is the time consumption of states. In our model, transitions represent external service calls and states the execution of internal component code. In many cases, the code between two external method calls can take a significant time span and has to be considered in the computation. An ad-hoc solution to this problem would be the introduction of an intermediate transition for states with a significant time consumption.

6 Conclusion

In this paper, we discussed the extension of parametric QoS contracts by SPNs. Parametric contracts require a service effect specification to model the relationship between provided and required services. Until now, the service effect specification is described by a finite state machine. The usage of automatons limits the application of parametric contracts to single-threaded software architectures. We proposed the usage of SPNs as service effect specifications. This would allow to model multi-threaded systems and the detection of resource conflicts. The model can be used to give a more precise estimation of the reliability and performance of a software architecture.

Furthermore, a formal analysis of a component-based software architecture can show up performance bottlenecks of the systems. Resource can be modelled and are included in the evaluation of a design decision. Performance break-ins can be found by analysing the model for different arrival rates. Our approach uses an abstract view on components for the predictions. These can be specified in early stages of the development. So, some performance bottlenecks and performance break-ins or a component whose services do not satisfy the QoS constraints can fixed in early stages of the developments. This saves time and costs in later stages of the development.

Until now, we only discussed the ideas to develop a model for the analysis of multi-threaded systems on the basis of parametric contracts and SPNs. The next steps are the implementation of a concrete prediction model, the development of support mechanisms for the creation of the service effect specification from the component's code and/or design documents like sequence charts, and a detailed evaluation of the prediction model.

Acknowledgements: We would like to thank the members of the Palladio research group for their constructive comments.

References

[1] G. Balbo. Title Stochastic Petri Nets: Accomplishments and Open Problems. In *International Computer Performance and Dependability Symposium*, pages 51 – 60, 1995.

[2] Falko Bause and Pieter S. Kritzinger. *Stochastic Petri Nets - An Introduction to the Theory*. Vieweg, 1996.

[3] Viktoria Firus, Steffen Becker, and Jens Happe. Parametric Performance Contracts for QML-specified Software Components. In *Formal Foundations of Embedded Software and Component-based Software Architectures (FESCA)*, Electronic Notes in Theoretical Computer Science. ETAPS 2005, 2005.

[4] Jens Happe. Reliability Prediction of Component-Based Software Architectures. Master thesis, Department of Computing Science, University of Oldenburg, December 2004.

[5] Scott A. Hissam, Gabriel A. Moreno, Judith A. Stafford, and Kurt C. Wallnau. Packaging predictable assembly. In Judy M. Bishop, editor, *Component Deployment, IFIP/ACM Working Conference, CD 2002, Berlin, Germany, June 20-21, 2002, Proceedings*, volume 2370 of *Lecture Notes in Computer Science*, pages 108–124. Springer, 2002.

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

[7] Marco Ajmone Marsan, Gianni Conte, and Gianfranco Balbo. A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems. *ACM Transactions on Computer Systems*, 2(1):93–122, May 1984.

[8] Daniel A. Menasce, Virgilio A.F. Almeida, and Lawrence W. Dowdy. *Performance by Design - Computer Capacity Planning by Example*. Prentice Hall, 2004.

[9] Bertrand Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, October 1992.

[10] Oscar Nierstrasz. Regular Types for Active Objects. In *Proceedings of the 8th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-93)*, volume 28, 10 of *ACM SIGPLAN Notices*, pages 1–15, October 1993.

[11] Ralf H. Reussner, Viktoria Firus, and Steffen Becker. Parametric Performance Contracts for Software Components and their Compositionality. In Wolfgang Weck, Jan Bosch, and Clemens Szyperski, editors, *Proceedings of the 9th International Workshop on Component-Oriented Programming (WCOP 04)*, June 2004.

[12] Ralf H. Reussner, Iman H. Poernomo, and Heinz W. Schmidt. Reasoning on Software Architectures with Contractually Specified Components. In A. Cechich, M. Piattini, and A. Vallecillo, editors, *Component-Based Software Quality: Methods and Techniques*, number 2693 in LNCS, pages 287–325. Springer-Verlag, Berlin, Germany, 2003.

[13] M. Ribaudo. Stochastic petri net semantics for stochastic process algebras. In *PNPM '95: Proceedings of the Sixth International Workshop on Petri Nets and Performance Models*, page 148, Washington, DC, USA, 1995. IEEE Computer Society.

[14] C. U. Smith and L. G. Williams. *Performance Solutions: a practical guide to creating responsive, scalable software*. Addison-Wesley, 2002.

[15] Wil van der Aalst. Pi Calculus Versus Petri Nets: Let Us Eat Humble Pie Rather Than Further Inflate the Pi Hype. *BPTrends*, 3(5):1–11, May 2005.