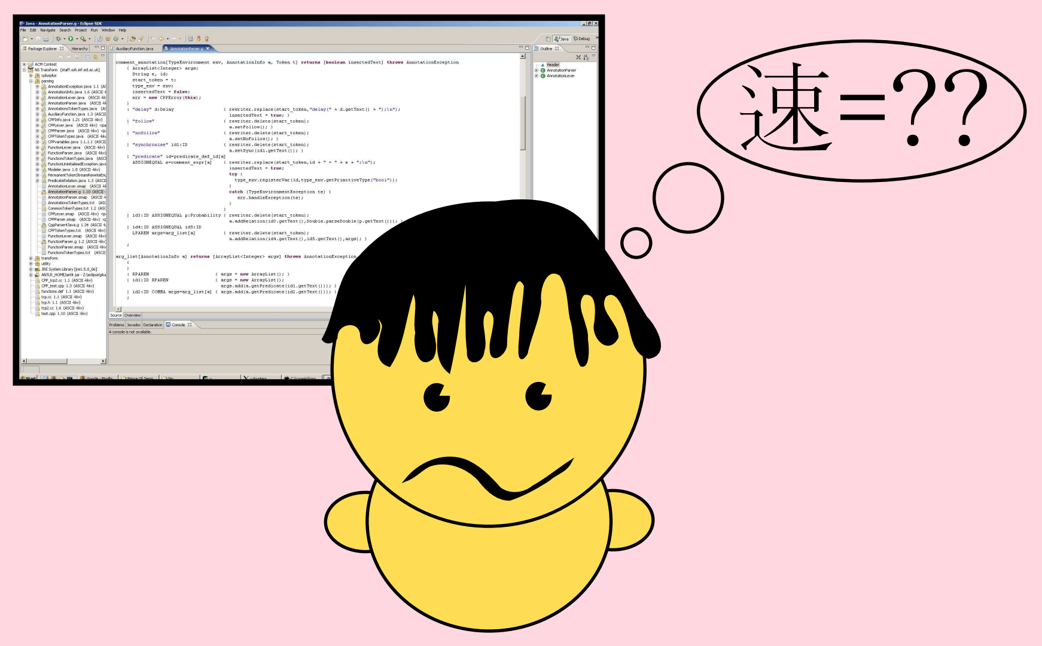
Performance-Driven Development

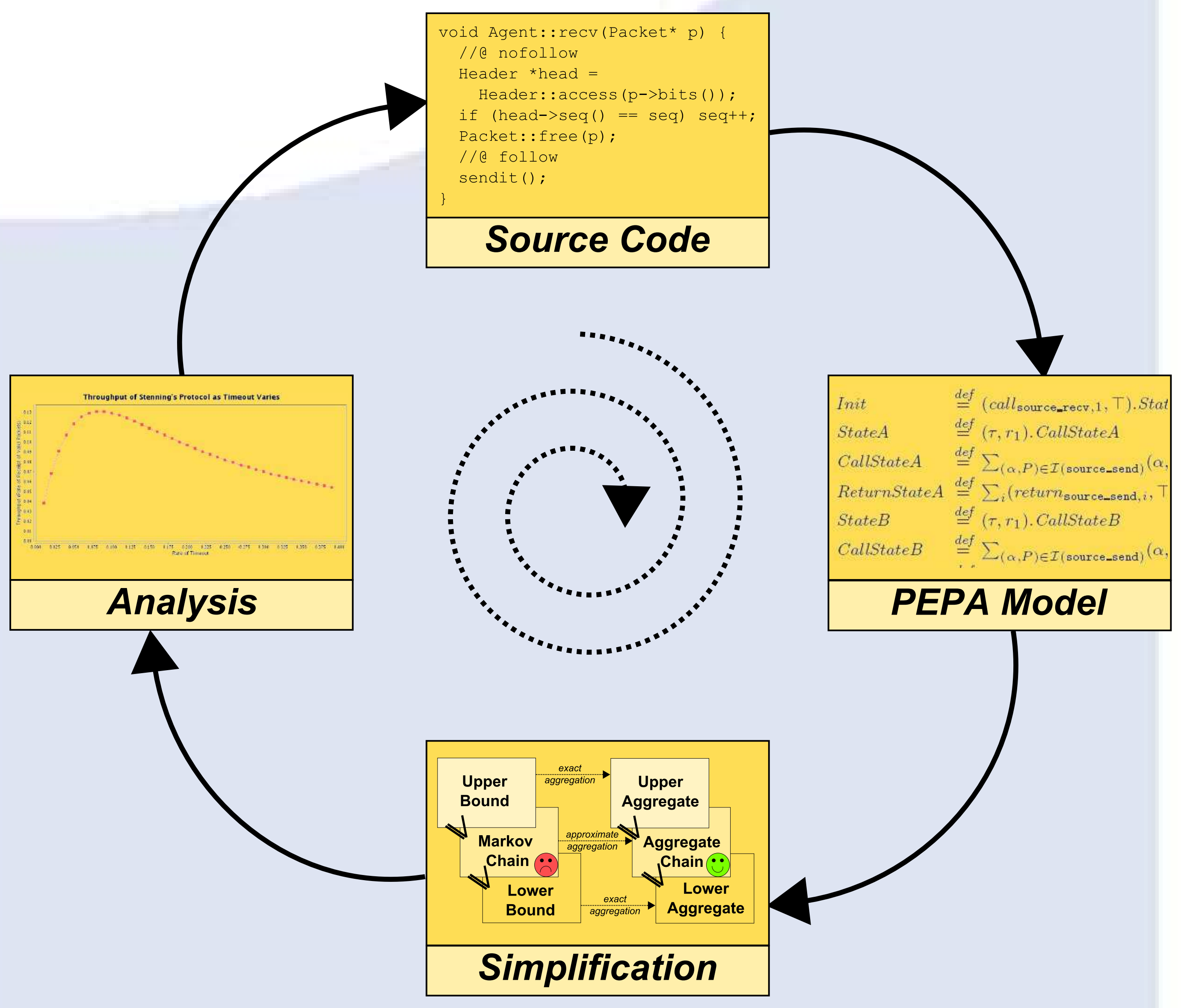
Michael J. A. Smith
M.J.A.Smith@sms.ed.ac.uk

Introduction



Ask the average developer to think about performance, and you might as well ask them in Chinese. Except for in a few specialised areas, there is a general culture of "write first, optimise later." This is worrying, because improving software performance is largely **not about optimising algorithms**. If you have a distributed system, its **communication architecture** will have the single biggest impact on performance, and fixing this afterwards will be too late!

Performance requirements need to be addressed **throughout the development process**. To encourage developers to do this, we need to give them some help. In the world of performance modelling, we have many high level languages, such as **stochastic process algebras** [2], which can describe a system's behaviour as a **stochastic process** that can be solved analytically. This sort of modelling is too difficult and time-consuming for the majority of developers, so we need to give them a tool that does it **automatically**. This means analysing and deriving a model from *real code*!



```
void Agent::recv(Packet* p) {
  //@ nofollow
  Header *head =
    Header::access(p->bits());
  if (head->seq() == seq) seq++;
  Packet::free(p);
  //@ follow
  sendit();
}
```

$$
\begin{aligned}
Init &\stackrel{def}{=} (call_{\mathtt{source\_recv},1}, \top).Stat\\
StateA &\stackrel{def}{=} (\tau, r_1).CallStateA\\
CallStateA &\stackrel{def}{=} \textstyle\sum_{(\alpha,P)\in\mathcal{I}(\mathtt{source\_send})}(\alpha,\\
ReturnStateA &\stackrel{def}{=} \textstyle\sum_i(return_{\mathtt{source\_send},i}, \top\\
StateB &\stackrel{def}{=} (\tau, r_1).CallStateB\\
CallStateB &\stackrel{def}{=} \textstyle\sum_{(\alpha,P)\in\mathcal{I}(\mathtt{source\_send})}(\alpha,
\end{aligned}
$$

Our view a performance-driven development process is shown above. The most important steps are the extraction of a model from source code (in this case in PEPA, a stochastic process algebra *[see below]*), and the simplification of the model to a manageable size. This can then be applied to **partially completed code**, so that the developer can iteratively modify the implementation, based on feedback from the model about properties such as **utilisation**, **throughput** and **response time**.

Performance Evaluation Process Algebra (PEPA)

PEPA [2] is a high-level modelling language, in which a *system* is a set of concurrently running *components*, which cooperate over certain *activities*. Each activity has a *rate* associated with it, which parameterises an exponential distribution determining its duration. Hence a PEPA model maps onto a continuous-time Markov chain. PEPA has the following syntax:

$$
\begin{array}{rcll}
S & := & (\alpha, r).S & \text{(prefix - action } \alpha \text{ at rate } r\text{)}\\
 & \mid & S_1 + S_2 & \text{(choice - behave as either } S_1 \text{ or } S_2\text{)}\\
 & \mid & A & \text{(constant - behave as component } A\text{)}\\
P & := & P_1 \underset{L}{\bowtie} P_2 & \text{(cooperation - over actions in } L\text{)}\\
 & \mid & P/L & \text{(hiding - actions in } L \text{ are hidden)}\\
 & \mid & S & \text{(sequential component)}
\end{array}
$$

Stochastic Abstraction of Code

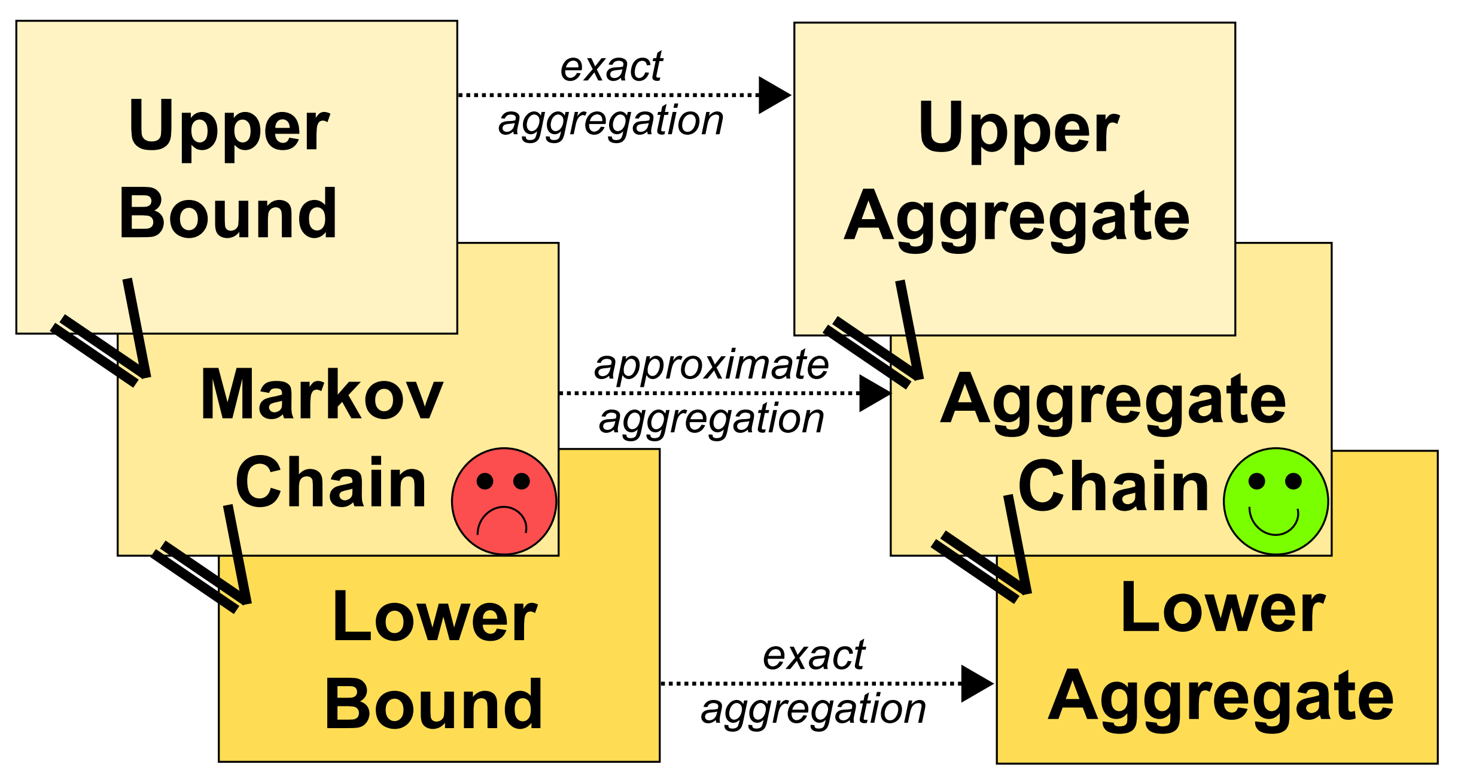
The first step in the above process is to arrive at a **PEPA model** from **user code**. The goal is to do this automatically, but we need some help from the user, in the form of **annotations**. This is discussed in more detail in [3]. The two main stages are modelling the **system**, or how all the user's functions fit together with the network, and modelling each **function**. The former is where we need the user annotations.

Modelling a function requires abstraction of both the **data environments** and **control-flow**. Since integer variables can take on such a large number of values, we abstract this using **intervals**. For example, if we have an `if`-statement that tests whether '`x < 0`', then just by knowing whether the value of `x` lies in the interval $[-\infty, -1]$ or $[0, \infty]$, we can determine which branch is taken. In abstracting control-flow, the main difficulty comes from loops. To keep the state space manageable, we only model a single iteration of the loop, and then determine the **probability** of re-entering the loop based on the current data environment.

Model Simplification

Even after abstracting the code, the state space can easily be too large to analyse. Model simplification means reducing the size and complexity of the model to make it **small enough to solve**, whilst keeping the **behaviour close to the original**. One approach is **state aggregation**, where we use a single state to approximate a collection of states with similar behaviour. But most *approximate* solution methods give no formal treatment of the errors involved, and exact methods can rarely be applied to real-world models.

A more sound approach is to use **stochastic bounds**. For a Markov chain M, we find upper and lower bounding chains, M_U and M_L, such that their steady state distributions are upper and lower bounds of the actual steady state. If we construct M_U and M_L so that they can be *exactly* aggregated, then we can solve them, and get an exact bound on the aggregated steady state of M.



Fourneau *et al* [1] give an algorithm for constructing such bounds for Markov chains. In our recent work, we have extended this to PEPA models, allowing stochastic bounds to be constructed **compositionally**.

Further Work

The implementation of this work is ongoing, and there are many technicalities when dealing with real-world languages such as C, but we have an algorithm for model extraction from a restricted subset of C [4], and a stochastic abstract interpretation for PEPA, to produce compositional stochastic bounds. Apart from the implementation, the two important next steps are to formalise the model extraction process as an **abstract interpretation**, and to improve the accuracy of stochastic bounds when we simplify.

References

[1] J. M. Fourneau, M. Lecoz, and F. Quessette. Algorithms for an irreducible and lumpable strong stochastic bound. *Linear Algebra and its Applications*, 386:167–185, 2004.

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

[3] M. J. A. Smith. Towards stochastic model extraction: Performance evaluation, fresh from the source. In *Proceedings of Process Algebra and Stochastically Timed Activities (PASTA) 2006*, 2006.

[4] M. J. A. Smith. Stochastic modelling of communication protocols from source code. In *Proceedings of the 5th Workshop on Quantitative Aspects of Programming Languages (QAPL)*, 2007.