Stochastic Modelling of Communication Protocols from Source Code

Michael J. A. Smith[1,2]

Laboratory for Foundations of Computer Science
University of Edinburgh
Edinburgh, United Kingdom

Abstract

A major development in qualitative model checking was the jump to verifying properties of source code directly, rather than requiring a separately specified model. We describe and motivate similar extensions to quantitative/performance analyses, with particular emphasis on communication protocols. The central aim is to extract a stochastic model (in the PEPA language) from such source code.
We construct a model compositionally, so that each function in the system corresponds to a sequential PEPA process. Such a process is derived by abstract interpretation over the state machine of a function, using interval abstraction to represent linear expressions of integer variables. We illustrate this by an analysis of a simple protocol.

Keywords: Performance modelling, Stochastic process algebra, Static analysis, Communication protocols

1 Introduction

Communication protocols are notoriously difficult to get right. Not only do the usual challenges of distributed and concurrent programming apply, but they provide a service that other applications depend upon. Thus the *performance* of a protocol is critical to its success. For example, if a routing protocol fails to react quickly to changes in topology, the network can be brought to a standstill. Similarly, a reliable transport-layer protocol must be able to maintain a reasonable throughput, even when the network is congested. Because of this, it is vital to understand the performance characteristics of these protocols.

There are currently two approaches to analysing such performance properties; either we dynamically take measurements from the real system, or we build an abstract model, which can then be analysed. The former includes techniques such as code profiling and operational analysis (as applied to the measurements taken), which can give accurate figures if we have access to the deployed system. The

[1] EPSRC, Microsoft Research
[2] Email:M.J.A.Smith@sms.ed.ac.uk

latter includes simulation and mathematical modelling (at various degrees of abstraction), which are much more useful for *explaining* the behaviour of the system, and *predicting* its behaviour before deployment, but require a separate model to be developed. Simulations are often complicated, and may contain bugs. On the other hand, mathematical modelling is beyond the skill of the typical programmer, and also prone to mistakes.

Stochastic extensions to existing formalisms in concurrency theory, such as process algebra, have considerably mitigated this last problem. In particular, the Performance Evaluation Process Algebra (PEPA) [9] is a high-level and compositional language, in which models describe continuous time Markov chains (CTMCs). This is arguably more intuitive, and less prone to error, than working directly with these mathematical structures.

Despite these advances, performance models are still very much removed from implementations. Work has been done to derive PEPA models from UML [3], but this is at a higher level than the implementation itself. In most cases, the model is validated empirically, by comparing its predictions to measurements taken from the real system (and refining the model if necessary). However, when the source code of the system is available, we can obtain a much more definite handle on what it means for a model to be *correct*. In this paper we present the first steps towards solving this problem, by describing an abstraction to a performance model, directly from source code.

In the world of *qualitative* model checking, where we are concerned with just the *possibility* of certain behaviours, this step has already been taken. SLAM [2] and Blast [8] both use *predicate abstraction* and *counter-example guided refinement* to verify such properties directly on real code, written in C. However, we cannot simply apply the same approach in a *quantitative* setting, since we do not have a well-defined notion of counter-example. Indeed, the problem is made much more difficult since we need to determine the *probability* of control-flow decisions, given some abstract environment of the program's variables. The search space of such abstractions is simply too large to explore by a sequence of refinements, and so we must avoid initially over-abstracting the program.

The benefits of such a technique for model extraction are numerous, and can be applied to more general distributed systems (web services being a prime example), rather than just communication protocols. Our main motivations are as follows. Firstly, we want to encourage wider application of performance analysis techniques, by providing a tool that non-specialists can use. Microsoft®'s Static Driver Verifier (SDV) [1] is a good example of how theory can be successfully applied in this way. Secondly, we want to allow *non-functional testing* to take place throughout the development cycle, rather than just at the end. We can do this by enabling performance evaluation of *partially completed code*. Finally, we wish to allow developers to verify that a protocol (or more general distributed system) satisfies some *performance contract*, or service-level agreement (SLA). The work in [7] takes some steps towards this, but at a more abstract level, in the context of web services.

In this paper, we begin by introducing the structure of the protocols we will be analysing, and the language of the source code we consider, in Section 2. We then briefly introduce the PEPA language in Section 3. In the following two sections

(4 and 5 respectively), we describe how to construct a PEPA model first at the structural level (i.e. how to build a model of the system from models of the functions) and then at the functional level (i.e. how to build a model of a function from its source code). To illustrate this, we analyse a simple transport protocol in Section 6. We conclude with some comments on future work in Section 7.

2 Communication Protocols and Source Code

In this paper, we will limit our analysis to that of *end-to-end* communication protocols. In other words, we will not consider hop-by-hop protocols, such as those used for routing, since representing the topology of such systems leads to an unmanageable state space. We do, however, wish to deal with *real* protocols, and so we need to analyse real-world languages. In this case, that means C.

There exist a number of tools for handling C, such as CCured [16], which together with the C Intermediate Language (CIL) [15] provides a cleaner (and type-safe) framework for analysis. However, even with the aid of these tools, the analysis of arbitrary programs is uncomputable (if we wish to retain some notion of the error involved). Fortunately, most protocols do not exhibit complex looping or recursive behaviour, and so we can justifiably consider only a subset of the language.

Let us take a subset of C, with only integer variables, boolean variables and enumeration types. In addition, we impose the following restrictions:

(i) *No pointers.* We intend to relax this in future work, but that is beyond the scope of this paper.

(ii) *No recursion.* This is beyond our ability to model in a Markovian setting, due to the memoryless property of states.

(iii) *We allow only linear conditions* of the form $\sum_{i=1}^{n} a_i x_i \ \{<, \leq, =, \geq, >\} \ c$, where a_i and c are integer constants.

(iv) *Loop variables must be memoryless with respect to previous iterations of the loop, or else vary linearly with time.* In other words, on each iteration, a loop variable must either be set independently of its previous value, or incremented/decremented each time by a constant.

The restriction on conditions is quite a strict one. In particular, we can see that procedures like exponential backoff do not satisfy this. We expect that this restriction can be relaxed somewhat, but that is the subject of future work.

3 The PEPA Language

The target of our abstraction is a PEPA model. In PEPA, a *system* is a set of concurrent *components*, which are capable of performing *activities*. An activity $a \in \mathcal{A}ct$ is a pair (α, r), where $\alpha \in \mathcal{A}$ is its action type, and $r \in \mathbb{R}^+ \cup \{\top\}$ is the rate of the activity. This rate parameterises an exponential distribution, and if unspecified (denoted $\top$), the activity is said to be *passive*. This requires another component in cooperation to actively drive the rate of this action. PEPA terms

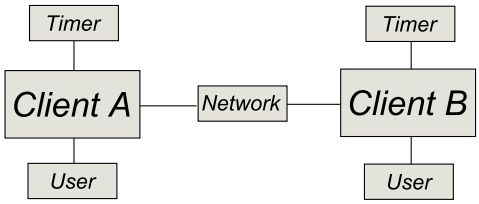

Fig. 1. The structure of an end-to-end protocol

have the following syntax:

$$P := (\alpha, r).P \mid P_1 + P_2 \mid P_1 \underset{L}{\bowtie} P_2 \mid P/L \mid A$$

We briefly describe these combinators as follows. For more detail, we refer the reader to [9].

- *Prefix* ($(\alpha, r).P$): the component can carry out an activity of type α at rate r to become the component P.
- *Choice* ($P_1 + P_2$): the system may behave either as component P_1 or P_2. The current activities of both components are enabled, and the first activity to complete determines which component proceeds. The other component is discarded.
- *Cooperation* ($P_1 \underset{L}{\bowtie} P_2$): the components P_1 and P_2 synchronise over the cooperation set L. For activities whose type is not in L, the two components proceed independently. Otherwise, they must perform the activity together, at the rate of the slowest component. At most one of the components may be passive with respect to this action type.
- *Hiding* (P/L): the component behaves as P, except that activities whose type is in L are hidden, and appear externally as the unknown type τ.
- *Constant* ($A \stackrel{def}{=} P$): the name A is assigned to component P.

The operational semantics of PEPA defines a labelled multi-transition system, which induces a *derivation graph* for a given component. Since the duration of a transition in this graph is given by an exponentially distributed random variable, this corresponds to a CTMC.

4 Structural Modelling

Assuming that we can model the behaviour of a function, what does a model of the system look like? For an end-to-end protocol, we have two clients, A and B, which communicate over a network. The operation of the protocol is driven by *events*, which fall into three categories – user interactions (i.e. telling the protocol to do something), receiving packets over the network, and timeouts. This is shown schematically in Figure 1, and leads to the following general form of the PEPA system equation, where the action sets U, T, R and S define the interfaces between the components:

$$(User \underset{U_A}{\bowtie} ClientA \underset{T_A}{\bowtie} Timer) \underset{S_{AB} \cup R_{BA}}{\bowtie} Network \underset{S_{BA} \cup R_{AB}}{\bowtie} (User \underset{U_B}{\bowtie} ClientB \underset{T_B}{\bowtie} Timer)$$

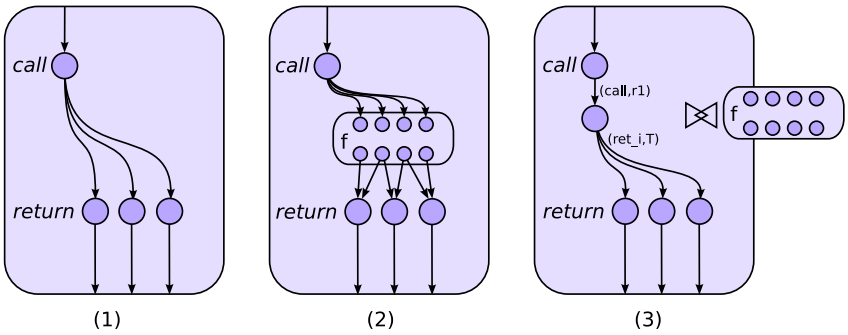

Fig. 2. The three ways of modelling a function call

Furthermore, a client X will have the following form at the top-level:

$$ClientX \stackrel{def}{=} \sum_i (recv_i, \top).RecvX_i + \sum_j (usercall_j, \top).UserCallX_j + (timeout, \top).TimeoutX$$

where i ranges over the abstract environment space of packets (i.e. it encodes an abstracted version of the packet contents), and j over that of the user interface (i.e. the API calls, and corresponding arguments, that the user can make). The states $RecvX$, $UserCallX$, and $TimeoutX$ correspond to models of the corresponding functions on the client, whose construction is described in Section 5. Note that this implies a single-threaded client, since only one function can be called at a time. We can model multi-threaded clients by composing the functions in parallel.

The network can be modelled in a number of ways, depending on the properties required. When the model of the system is constructed, we expect the user to choose the network from a library of components, so that they do not need to write the PEPA process themselves. For example, a half-duplex network with probability p of packet loss would look like the following:

$$Network \stackrel{def}{=} \sum_i (A\_send_i, \top).NetworkAB_i + \sum_j (B\_send_j, \top).NetworkBA_j$$

$$NetworkAB_i \stackrel{def}{=} \sum_j (B\_recv_i, (1-p).r_{network}).Network + (\tau, p.r_{network}).Network$$

$$NetworkBA_j \stackrel{def}{=} \sum_i (A\_recv_j, (1-p).r_{network}).Network + (\tau, p.r_{network}).Network$$

where i and j range of the abstract environment of packets sent by clients A and B respectively.

It should be apparent that we encode the passing of both arguments and return values (when calling a function) and the contents of packets (when communication across the network) by an interface of *actions*. We will consider this interface in more detail in the next section, but first we must discuss how a function call can be modelled. When we abstract a function, we will reduce each sequential block to a single transition in the model; namely a single action, with a single rate. Given this, there are three fundamental approaches to modelling a function call, as illustrated in Figure 2:

(i) Abstract the call to a single transition to the result state. This assumes that the function executes at a fixed rate, irrespective of the input, but this is often

good enough for our purposes, and simplifies the model considerably.

(ii) Explicitly embed a model of the function. This is more general, and is appropriate when the function has a more complex behaviour that we wish to capture. The disadvantage is that we must remember which environment we were in before calling the function, so that we can recover the correct state afterwards. In the worst case, this means that we need a separate copy of the function for each environment we call it from, and so it is undesirable unless essential to the behaviour of the model. We assume that such a call is synchronous.

(iii) Model the function as a separate process running in parallel, which synchronises over *call* and *return* actions. This separates the functionality of one function from another, at the expense of an exponential blowup of state when we do a Markovian analysis. We will use this abstraction when two components are communicating over the network, and the call is assumed to be asynchronous.

We rely on *user annotations* in the source code to tell us how to analyse each function. In the first case, the user must also provide a *summary* of the function's behaviour (i.e. how it affects the environment of the caller), so that we can model the system without having to analyse every function (e.g. libraries, system calls, etc.). Note that the second and third cases are essentially equivalent, except for whether the call is synchronous or asynchronous.

Once we have derived models for all the functions in the system, we must end up with a set of 'top-level' functions; namely those that are invoked externally (by a network event, timeout, or user call). These fit into the component *ClientX* described previously, which in turn forms part of the system equation. Other than having the user pick out which are the top-level functions, and how the network and users behave, the system equation can be constructed automatically – the synchronisation sets are just the interfaces we compute in the next section.

5 Functional Modelling

Up to now, we have looked only at how to compose a model of the system from its sequential components. These sequential components correspond to functions in the source code, and we will now describe their abstraction. There are two key ideas involved in this – that of abstracting the control-flow of the program, and that of abstracting the environment of its variables.

The steps that we will take can be summarised in four steps. Firstly, we convert the program to an abstract syntax, in Static Single Assignment (SSA) form. From here, we derive the control-flow state space. We define this as the fixed point of a reduction $\rightarrow_f$, but it can be viewed intuitively in terms of paths on the control-flow graph. Thirdly, we determine the data environment space. We abstract arithmetic expressions to *intervals* over the integers, for which considerations of *independence* are of vital importance. Finally, the PEPA model can be built. This involves determining the probability of moving from one state to another, which can be phrased as a conditional probability on the data environments of reachable states (in the control-flow).

5.1 Abstract Syntax and SSA

In order to proceed, we must first convert the function (written into the subset of C that we defined) into an abstract syntax that will be easier to analyse. A function definition has the form $f(X_1, \ldots, X_n) := C$, where the body of the function is a command C, defined as follows:

$$\begin{array}{rcl} C & \texttt{:=} & \texttt{skip} \mid \texttt{return}\ E \mid X\ \texttt{:=}\ E \mid X\ \texttt{:=}\ g(X_1, \ldots, X_n) \\ & \mid & C_1\ \texttt{;}\ C_2 \mid (\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2)\ \texttt{;}\ \Phi \mid \texttt{while}\ \Phi\ \texttt{;}\ B\ \texttt{do}\ C \end{array}$$

Here, X are variables (which we limit to integers and booleans), f, g are functions, E are arithmetic expressions, B are boolean expressions, and Φ are sequences of ϕ-functions, which will shortly be defined. E and B must be linear, so that any expression E can be written in the form $\sum_i a_i X_i$, where a_i are constants. It should be clear how to convert a C function to this form, and we will therefore adopt this syntax from now on.

To simplify the analysis of variable dependencies, we also convert the function into SSA form [17]. This ensures that each variable is only assigned to (statically) once, so we need not worry about a variable name being reused later, in an independent context. To transform the function, we alter its control-flow graph in two ways. First, at each node in the control-flow graph with more than one incoming edge (a join node), and for each set of conflicting definitions of the same variable, we insert a ϕ-function, $\phi(d_1, \ldots, d_n)$, such that $\phi(d_1, \ldots, d_n) = d_i$ if the node is reached via the ith in-edge. We then rename all variables so that each is only *statically* assigned to once.

In our abstract language, each node will have at most two in-edges (nested `if`-statements have seperate join nodes), so the arity of all ϕ functions will be two. If Φ is a sequence of n ϕ-functions, $X_1 := \phi(Y_1, Z_1)\ ;\ \ldots\ ;\ X_n := \phi(Y_n, Z_n)$, we define the following projections:

$$\Phi_L = X_1 := Y_1\ ;\ \ldots\ ;\ X_n := Y_n$$
$$\Phi_R = X_1 := Z_1\ ;\ \ldots\ ;\ X_n := Z_n$$

5.2 Control-Flow State Derivation

We can now define a state in our abstract system as a 4-tuple, $\langle L, C, P, U \rangle$, consisting of a label L, a command C, a predicate P and an update U. The *predicate* is a boolean expression on the program variables that is valid on entering the state. The *update* is a partial finite map from variables to expressions, indicating the change of state that takes place at that node. The *command* is the remainder of the program, to be executed after leaving the state.

To allow us to represent a function by these states, we introduce two more atomic commands; `goto` and `call`. The first of these specifies a set of labels, $L_1, \ldots, L_n$, which determine the set of reachable states that may follow. The second encodes the assignment of a variable X to a function call g, followed by a continuation L. The syntax of commands is extended as follows:

$$C \quad \texttt{:=} \quad \texttt{goto}\{L_1, \ldots, L_n\} \mid \texttt{call}(X, g, L)(E_1, \ldots, E_n)$$

Finally, before we derive the control-flow state space, we need a notion of *environment variable*. There is no need for every variable to be represented in the abstract environment, as some will be uniquely determined by the others. If this is the case, we can eliminate the variable by substituting for its definition, hence it will never need to appear in an update U. Informally, a variable is an environment variable if it is an input to the function, the return value of a function call, or if its definition reaches over the backward branch of a loop. Formally, we define a function $\mathcal{I} : C \to X \to X$, which maps each environment variable onto an equivalence class:

$$\begin{aligned}
&\mathcal{I}(\texttt{skip}) && = \{\} \\
&\mathcal{I}(\texttt{return}\ E) && = \{\} \\
&\mathcal{I}(X := E) && = \{\} \\
&\mathcal{I}(X := g(E_1, \ldots, E_n)) && = \{X \mapsto X\} \\
&\mathcal{I}(X_1 := \phi(X_2, X_3)) && = \{X_1 \mapsto X_1, X_2 \mapsto X_1, X_3 \mapsto X_1\} \\
&\mathcal{I}((\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2)\ ;\ \Phi) && = \mathcal{I}(C_1) \cup \mathcal{I}(C_2) \\
&\mathcal{I}(\texttt{while}\ B\ \texttt{do}\ \Phi\ ;\ C) && = \mathcal{I}(\Phi) \cup \mathcal{I}(C) \\
&\mathcal{I}(C_1\ ;\ C_2) && = \mathcal{I}(C_1) \cup \mathcal{I}(C_2) \\
&\mathcal{I}(f(X_1, \ldots, X_n) := C) && = \{X_1 \mapsto X_1, \ldots, X_n \mapsto X_n\} \cup \mathcal{I}(C)
\end{aligned}$$

If $X \in \mathrm{dom}(\mathcal{I}(C))$ then X is an environment variable in C. Furthermore, if $\mathcal{I}(C)(X) = \mathcal{I}(C)(Y)$, then X and Y are the *same* environment variable. To simplify notation, if we have a function f defined as $f(X_1, \ldots, X_n) := C$, then we define $\mathcal{V}_f$ to be $\mathcal{I}(f(X_1, \ldots, X_n){:=}C)$. In other words, $\mathcal{V}_f$ determines the environment variables of the function f.

We can now define the state space of an abstract function as the fixed point of a reduction, $\to_f$. This reduction takes a 4-tuple, which represents some state in the function's execution, and partially evaluates the command. In general, this partial evaluation leads to a *set* of possible states, because the control-flow decisions are not completely determined statically. We define $\to_f$ as follows, where L' and L'' are fresh labels:

$$\begin{aligned}
&\langle L,\ C,\ P,\ U\rangle && \to_f \{\langle L,\ C,\ P,\ U\rangle\} \text{ if } C \text{ is atomic (}\texttt{call}\text{, }\texttt{goto}\text{ or }\texttt{return}\text{)} \\
&\langle L,\ (\texttt{skip})\ ;\ C,\ P,\ U\rangle && \to_f \{\langle L,\ C,\ P,\ U\rangle\} \\
&\langle L,\ (\texttt{return}\ E)\ ;\ C,\ P,\ U\rangle && \to_f \{\langle L,\ \texttt{return}\ E,\ P,\ U\rangle\} \\
&\langle L,\ (X := E)\ ;\ C,\ P,\ U\rangle && \to_f \{\langle L,\ C\{E/X\},\ P,\ U\rangle\} \text{ if } X \notin \mathrm{dom}(\mathcal{V}_f) \\
&\langle L,\ (X := E)\ ;\ C,\ P,\ U\rangle && \to_f \{\langle L,\ C,\ P,\ U\{E/\mathcal{V}_f(X)\}\rangle\} \text{ if } X \in \mathrm{dom}(\mathcal{V}_f) \\
&\langle L,\ (X := g(E_1, \ldots, E_n))\ ;\ C,\ P,\ U\rangle && \to_f \left\{ \begin{array}{l} \langle L,\ \texttt{call}(X, g, L')(E_1, \ldots, E_n),\ P,\ U\rangle\,, \\ \langle L',\ C,\ \top,\ \{\}\rangle \end{array} \right\} \\
&\langle L,\ (\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2);\ \Phi\ ;\ C_3,\ P,\ U\rangle && \to_f \left\{ \begin{array}{l} \langle L,\ C_1\ ;\ \Phi_L\ ;\ C_3,\ P \wedge B,\ U\rangle\,, \\ \langle L,\ C_2\ ;\ \Phi_R\ ;\ C_3,\ P \wedge \neg B,\ U\rangle \end{array} \right\} \\
&\langle L,\ (\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2)\ ;\ \Phi,\ P,\ U\rangle && \to_f \left\{ \begin{array}{l} \langle L,\ C_1,\ P \wedge B,\ U\rangle\,, \\ \langle L,\ C_2,\ P \wedge \neg B,\ U\rangle \end{array} \right\} \\
&\langle L,\ (\texttt{while}\ B\ \texttt{do}\ \Phi\ ;\ C_1)\ ;\ C_2,\ P,\ U\rangle && \to_f \left\{ \begin{array}{l} \langle L,\ \texttt{goto}\{L', L''\},\ P,\ U\rangle\,, \\ \langle L',\ C_1\ ;\ \texttt{goto}\{L', L''\}, P \wedge B, \{\}\rangle\,, \\ \langle L'',\ C_2,\ \neg B,\ \{\}\rangle \end{array} \right\}
\end{aligned}$$

Note that we require that a function ends in a `return` instruction, so all other commands must be followed by another. The only exception to this is a conditional

statement, which may appear as the final command if both branches terminate in a `return` instruction, hence the two cases above.

Informally, this reduction is amalgamating sequential states in the concrete control-flow graph, and expanding out conditional statements, so that each abstract state represents a *path* between two *interaction* points – namely calling or returning from a function, or reentering a loop. Importantly, in this abstraction, we only have one set of states for the body of a loop. Hence we can only model a loop if the probability of reentering it has a trivial dependency with respect to time (as we will see later).

To compute the fixed point of this reduction, we firstly define $\Rightarrow_f$ as a reduction over sets of states:

$$\frac{t_1 \rightarrow_f T_1 \qquad \ldots \qquad t_n \rightarrow_f T_n}{\{t_1, \ldots, t_n\} \Rightarrow_f T_1 \cup \ldots \cup T_n}$$

Now, the abstract state space $\mathcal{S}(f)$ of a function f is defined as follows:

$$\mathcal{S}(f) = T \text{ iff } \{\langle 0,\ C,\ \top, \{\}\rangle\} \Rightarrow_f^* T \wedge \forall T'.T \Rightarrow_f^* T' \Rightarrow T' = T$$

This fixed-point can be shown to exist by induction on the structure of C, under the assumption that C is well-formed; namely, $\forall T.\, \{\langle 0,\ C,\ \top, \{\}\rangle\} \Rightarrow_f^* T \Rightarrow \exists T'.T \Rightarrow_f T'$. We will hereafter refer to $\mathcal{S}(f)$ as the *control-flow* states of f.

5.3 Data Environments

As it stands, the state space $\mathcal{S}(f)$ is not sufficient as the state space of a stochastic model of the function. The reason is that the predicate at each state is the *weakest* condition that must hold there. We therefore lose all memory of any stronger conditions that hold (for example, due to the particular path through which we arrived at the state), which prevents us from communicating these conditions at a later point (e.g. as a return value).

A further problem with these states is the difficulty in relating the predicates P to one another. To connect the states together probabilistically, we need to determine probabilities of the form $\mathbf{Pr}(P'_U|P)$, where P'_U is the predicate P' with its variables updated by U. Essentially, this is the probability of one set of linear constraints holding, given another, which can be difficult to determine. We therefore need to consider a *data* abstraction, in addition to $\mathcal{S}(f)$.

Let $\mathcal{P}_f$ be the set of all predicates P in $\mathcal{S}(f)$, expressed in the following normal form, where $\circledcirc \in \{<, \leq, =, \geq, >\}$:

$$\bigvee_i \left(\bigwedge_j \left(\sum_k a_{ijk} x_{ijk} \right) \circledcirc c_{ij} \wedge \bigwedge_j p_j \right)$$

where a_{ijk}, c_{ij} are rational constants, x_{ijk} are integer variables, and p_j are boolean variables (atoms). Now, let $\mathcal{E}_f$ be the set of all expressions $\sum_k a_k x_k$ in $\mathcal{P}_f$, represented in vector form, $\mathbf{a} \cdot \mathbf{x}$, where $\mathbf{x}$ is the image of $\mathcal{V}_f$, expressed as a vector, and $\mathbf{a}$ is a column vector of integers. If there are N unique such expressions, and M variables (i.e. $\mathbf{x}$ has dimensionality $M \times 1$), then the $M \times N$ matrix $\mathbf{A}$ is defined by taking its rows to be the vectors $\mathbf{a}$.

To determine the data environment, we need to find a *basis* for these vectors. Given our assumption that the environment variables have no hidden dependencies between one another (i.e. all dependencies are from conditions within the function in question), the independence of two expressions (in the absense of any other information) corresponds to orthogonality between their **a**-vectors. Hence, we can determine an optimal basis using Principle Components Analysis (PCA) [11]. This equates to performing singular value decomposition of the matrix $\mathbf{A}$ into $\mathbf{U}\Sigma\mathbf{V}^T$. Here, $\mathbf{U}$ is an $M \times M$ orthogonal matrix (i.e. its columns form an orthonormal basis), which can be viewed as a linear map from the target basis into the original. Hence $\mathbf{U}^T$ is a linear map into the target basis:

$$\mathbf{A}' = \mathbf{U}^T\mathbf{A} = \mathbf{U}^T\mathbf{U}\Sigma\mathbf{V}^T = \Sigma\mathbf{V}^T$$

Hence $\mathbf{A}'$ is an $M \times N$ matrix whose columns represent the same expressions as in $\mathbf{A}$, but in the new basis $\mathbf{U}$.

For each basis vector $\mathbf{u}_i$ in $\mathbf{U}$, we associate two sets of rationals, s_i^U and s_i^L called the upper and lower *closed splittings* of $\mathbf{u}_i$. These splittings define a finite set of intervals, which will form the abstract data environment of the expression that $\mathbf{u}_i$ denotes. For example, if this expression is $x - y$, and the sets s_i^U and s_i^L are $\{0, 10\}$ and $\{10, 11\}$ respectively, then the abstract environment records only whether the value of $x - y$ lies in the interval $(-\infty, 0]$, $(0, 10)$, $[10, 10]$, $(10, 11)$ or $[11, \infty)$.

More formally, an open interval $(\underline{x}, \overline{x})$ over the rationals denotes the set $\{q \in \mathbb{Q} \mid \underline{x} < q < \overline{x}\}$, where $x \in \mathbb{Q} \cup \{-\infty, \infty\}$. Similarly, a closed interval $[\underline{x}, \overline{x}]$ denotes $\{q \in \mathbb{Q} \mid \underline{x} \leq q \leq \overline{x}\}$, for $x \in \mathbb{Q}$, and the definitions of $[\underline{x}, \overline{x})$ and $(\underline{x}, \overline{x}]$ are as expected. An *interval space* is a set I of intervals, such that $\bigcup i \in I = \mathbb{Q}$ and $\forall i_1, i_2 \in I. i_1 \neq i_2 \Rightarrow i_1 \cap i_2 = \emptyset$. To construct such an interval space, s_i^U and s_i^L define all the closed interval ends (for the upper and lower bounds on the interval respectively), causing the open ends to follow uniquely.

To determine the splittings for each basis, we consider the expressions in $e \in \mathbf{A}'$. For all e of the form $a'\mathbf{u}_i$, then for all conditions $e \circledcirc c$, we examine the normalised condition $\frac{e}{a'} \circledcirc' \frac{c}{a'}$. If $\circledcirc' \in \{<, \geq\}$, we add $\frac{c}{a'}$ to s_i^L, and if $\circledcirc' \in \{>, \leq\}$, we add $\frac{c}{a'}$ to s_i^U. Otherwise, for $\circledcirc' =$ '=', we add $\frac{c}{a'}$ to both s_i^L and s_i^U. These splittings define the *top-level* abstract data environment. All remaining conditions are on expressions of the form $\sum_i a'\mathbf{u}_i$. These define the remainder of the abstract data environment.

A state E in the abstract data environment completely determines the truth of all conditions in the function. We write $E = E_T \wedge E_S$, where E_T is the top-level environment, defining a *box* in the vector space that $\mathbf{U}$ determines. E_S can be seen as a conjunction of the remaining predicates and their negations; namely a set of linear constraints that must hold within E_T. We denote the abstract data environment space of a function f by $\mathcal{E}(f)$.

5.4 PEPA Model Construction

We can now bring together the control-flow and data abstractions to build a PEPA model. In this model, each state identifies a (path, environment) pair. The rates depend on both the expected duration of a path (which is determined by basic

block profiling), and the probability of moving from one environment to another. We describe this process more precisely as follows.

Recall that $\mathcal{S}(f)$ is the set of all control-flow states, and $\mathcal{E}(f)$ the set of all data environments, for the function f. Now, for $S = \langle L, C, P, U\rangle \in \mathcal{S}(f)$ we will denote the projection operations by $S_L \ldots S_U$ respectively. The states of the PEPA model are denoted $State_{i,j}$, where i and j range over the control-flow and data states respectively, and are indices into the sets $\mathcal{S}(f)$ and $\mathcal{E}(f)$ respectively (under some ordering). For conciseness, we will herein refer to just $\mathcal{S}$ and $\mathcal{E}$, in relation to the function f.

In constructing the model, we need to determine a *rate* for each state transition. This is partly determined by the probability of moving from one state to another (which we will show below how to calculate), but also dynamically by the average time taken to execute the instructions corresponding to that state. We can measure this execution time empirically using basic block profiling – namely, we profile the sequential instructions by running them multiple times, and averaging the duration with respect to a control. For a state S, we will denote the inverse of this duration (i.e. its rate) by S_R

In general, the probabilities we must calculate are of the form $\mathbf{Pr}(P \mid E)$, where P is a boolean expression containing linear conditions, and E is an environment. We can simplify matters by splitting the environments E into their two components, E_T and E_S, allowing us to compute the following:

$$\mathbf{Pr}(P \mid E_T \wedge E_S) = \frac{\mathbf{Pr}(P \wedge E_S \mid E_T)}{\mathbf{Pr}(E_S \mid E_T)}$$

The top-level environment, E_T, defines a rectilinear volume in n-dimensional space (where n is the number of basis vectors). Hence we are computing the probability of a set of linear conditions holding in this volume. To compute these probabilities generally, we apply a dart-throwing Monte Carlo method. The basic approach here is to take a sample of points from the environment defined by E_S, and evaluate the compound condition (left hand side of the conditional probability) for each of them. The proportion of points for which the condition evaluates to true gives an estimate of the probability of the condition being true over the population.

We define a function $reach(S)$ as the reachable control-flow states from S:

$$\begin{aligned}
reach(\langle L, \texttt{goto}\{L_1, \ldots, L_n\}, P, U\rangle) &= \{S | S_L \in \{L_1 \ldots L_n\}\} \\
reach(\langle L, \texttt{call}(X, g, L)(E_1, \ldots, E_n), P, U\rangle) &= \{S | S_L = L\} \\
reach(\langle L, \texttt{return}\ E, P, U\rangle) &= \{S | S_L = 0\}
\end{aligned}$$

When we define the PEPA processes for the states $State_{i,j}$, we have three different cases to consider. For $S = \mathcal{S}_i$, we look at the type of command, S_C. For a `goto` instruction, the state change is entirely internal, and so the action will be of type τ. For `call` and `return` instructions, however, we are interacting with another function, and so need a public *interface* of actions. When the function gets called, it must enter an initial control-flow state. Since it relies on the arguments it was called with to determine this, we must encode the control-flow state into a set of *call* actions. Similarly, the return value is passed by a set of *return* actions. The

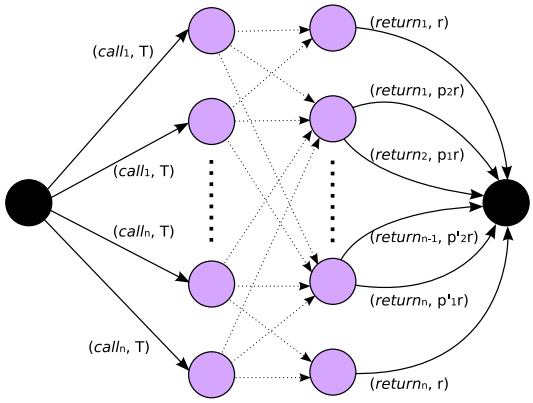

Fig. 3. The structure of a function interface

function begins and ends in a special state *Init*, where it passively waits to be called. This is shown diagramatically in Figure 3, where the black state is *Init*. Since it is the caller that must determine what environment to instatiate the function with, it needs to know what information the function requires. The interface $\mathcal{I}(f)$ of a function f is a set of pairs, (α, P), where α is an action and P is the predicate held by the action. In other words, if a function is called using α, P is the initial environment that will hold. Hence we define:

$$\mathcal{I}(f) = \{(call_{f,i}, P) | S_L = 0 \wedge S_P = P \wedge S = \mathcal{S}(f)_i\}$$

When we return from a function, it is the caller that needs to specify the abstract environment of the return value. We represent this *calling context* as a triple $(\mathcal{R}, l, X)$ of an interval space $\mathcal{R}$, an index l into that space, and a variable X to update. $R = \mathcal{R}_i$ is the environment from which the function was called, and the callee evaluates the probability of moving to other states in $\mathcal{R}$ when it returns. Note that only certain components of the environment space will be modified by the function call; namely those relating the return value to other environment variables. Hence we can optimise the implementation by only considering certain states.

In general, it follows that calculating the return value of a function depends on both the state of the function and that of the caller. In other words, we need to represent the calling environment in some way to generate a model of the function. There are two ways of doing this:

(i) Generate a seperate model of the function for each calling context. This corresponds to embedding the model of the function into that of the caller.

(ii) Use *functional rates* [10]. Here we only have one model of the function, in parallel with that of the caller, and the rates are expressed as a function of the state of the caller. To model this, we introduce an additional component, recording the calling context, which can be passively set by the caller, and queried by the function. We can think of this component as an *oracle*, allowing the correct probabilities of environment change to be represented in the model.

To simply the presentation that follows, we will make no assumption about which approach is taken; only that the calling context is available to the function.

We will now define the PEPA model of f. Firstly, the initial state of the function

is defined as:

$$Init \stackrel{def}{=} \sum_{S' \in \mathcal{S} | S'_L = 0} \sum_{E' \Rightarrow S'_P \in \mathcal{E}} (call_{f,i}, \mathbf{Pr}((E' \mid S'_P).\top).State_{i',j'}$$

where $S' = \mathcal{S}_{i'}$ and $E' = \mathcal{E}_{j'}$. Note that the probability is not of the form $\mathbf{Pr}(P \mid E)$, but we can calculate it in exactly the same way by splitting the predicate into an orthogonal component P_T, and the remaining conditions P_S, as we did for environments in Section 5.3.

For all $S = \mathcal{S}_i$ such that S_C is a `goto` instruction:

$$State_{i,j} \stackrel{def}{=} \sum_{S' \in reach(S)} \sum_{E' \Rightarrow S'_P \in \mathcal{E}} (\tau, \mathbf{Pr}((E'\{S_U(x)/x\} \mid E).S_R).State_{i',j'}$$

where $S = \mathcal{S}_i$, $E = \mathcal{E}_j$, $S' = \mathcal{S}_{i'}$ and $E' = \mathcal{E}_{j'}$.

The states where S_C is a '`call`$(X, g, L)(E_1, \ldots, E_n)$' instruction are defined as:

$$\begin{array}{ll} State_{i,j} & \stackrel{def}{=} \sum_{E' \Rightarrow S'_P \in \mathcal{E}} (\tau, \mathbf{Pr}((E'\{S_U(x)/x\} \mid E).S_R).CallState_{i,j'} \\ CallState_{i,j} & \stackrel{def}{=} \sum_{(\alpha,P) \in \mathcal{I}(g)} (\alpha, \mathbf{Pr}(P\{E_1/arg_1 \ldots E_n/arg_n\} \mid E').r).ReturnState_{i,j} \\ ReturnState_{i,j} & \stackrel{def}{=} \sum_{k=0}^{|\mathcal{E}|} (return_k, \top).State_{i',k} \text{ such that } S' \in reach(S) \wedge \mathcal{E}_k \Rightarrow S'_P \end{array}$$

where $S = \mathcal{S}_i$, $E = \mathcal{E}_j$, $E' = \mathcal{E}_{j'}$, and r is the rate of calling the function (determined empirically). Intuitively, *CallState* corresponds to invoking the function, after which we enter *ReturnState* where we wait for it to return.

Finally, when S_C is a '`return` E_R' instruction, under the calling context $(\mathcal{R}, l, X)$ we define:

$$State_{i,j} \stackrel{def}{=} \sum_{k=0}^{|\mathcal{R}|} (return_k, \mathbf{Pr}(\mathcal{R}_k\{E_R/X\} \mid \mathcal{R}_l \wedge E\{S_U(x)/x\}).S_R).Init$$

where $S = \mathcal{S}_i$ and $E = \mathcal{E}_j$.

Finally, having derived a model for each function in the system, we can compose them as described in Section 4.

6 Stenning's Protocol – An Example

To illustrate the approach we have described, let us examine a simple protocol. Stenning's protocol [12] provides reliable end-to-end delivery of packets in the simplest possible way. A source and a sink each locally maintain a *sequence number* for the connection. When the source sends a packet, it attaches its current sequence number to it. When the sink receives the packet, it checks whether the sequence number matches what it was expecting, and if so increments its sequence number and sends an acknowledgement to the source. The source will send the next packet if it receives a correct acknowledgement; otherwise, after a timeout, it will retransmit the last packet.

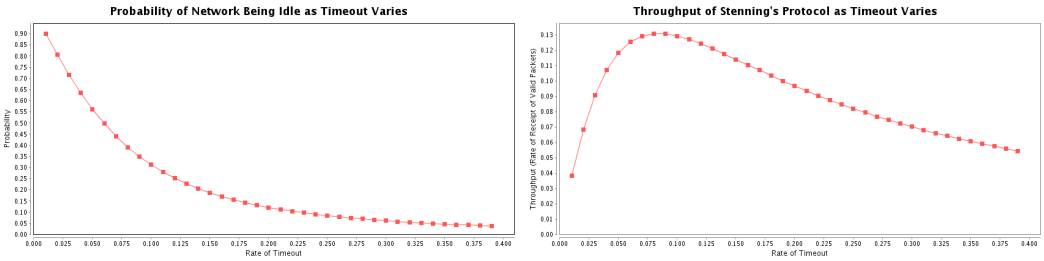

Fig. 4. Utilisation and throughput of Stenning's protocol, as the timeout varies

Ignoring the payload of the protocol, the abstract receive function for the source is as follows, where `pseq` is the packet sequence number, and `sseq` is that of the source:

```
source_recv(pseq) :=
  if (pseq == sseq) then (sseq := sseq + 1) else skip;
  _ := source_send(sseq);
  return 1;
```

Here, the control flow states are as follows:

$$\left\{ \begin{array}{l} \langle 0,\ \texttt{call}(\_, \texttt{source\_send}, 1)(\texttt{sseq}),\ \texttt{pseq} = \texttt{sseq},\ \{\texttt{sseq} \mapsto \texttt{sseq} + 1\} \rangle, \\ \langle 0,\ \texttt{call}(\_, \texttt{source\_send}, 2)(\texttt{sseq}),\ \texttt{pseq} \neq \texttt{sseq},\ \{\} \rangle, \\ \langle 1,\ \texttt{return}\ 1,\ \top,\ \{\} \rangle,\ \ \langle 2,\ \texttt{return}\ 1,\ \top,\ \{\} \rangle \end{array} \right\}$$

There is only one expression occuring in the above predicates, namely $\texttt{sseq} - \texttt{pseq}$. This has the interval space $I = \{(-\infty, 0), [0, 0], (0, \infty)\}$, since the only comparison on this expression is to the constant zero. In this example, the data environment is trivial (in that there are no non-orthogonal components), and so we can proceed directly to construct the model. We will assume a calling context $(\mathcal{R}, l, X)$. Naming the control-flow states A to D, and for empirically-derived rates r_{call}, r_1 and r_2 we reach:

$$\begin{array}{ll} \mathit{Init} & \stackrel{def}{=} (\mathit{call}_{\texttt{source\_recv},1}, \top).\mathit{StateA} + (\mathit{call}_{\texttt{source\_recv},2}, \top).\mathit{StateB} \\ \mathit{StateA} & \stackrel{def}{=} (\tau, r_1).\mathit{CallStateA} \\ \mathit{CallStateA} & \stackrel{def}{=} \sum_{(\alpha,P)\in\mathcal{I}(\texttt{source\_send})} (\alpha, \mathbf{Pr}(P\{\texttt{sseq}/\mathit{arg}_1\} \mid \texttt{sseq} - \texttt{pseq} \in (0,\infty)).r_{call}).\mathit{ReturnStateA} \\ \mathit{ReturnStateA} & \stackrel{def}{=} \sum_i (\mathit{return}_{\texttt{source\_send},i}, \top).\mathit{StateC} \\ \mathit{StateB} & \stackrel{def}{=} (\tau, r_1).\mathit{CallStateB} \\ \mathit{CallStateB} & \stackrel{def}{=} \sum_{(\alpha,P)\in\mathcal{I}(\texttt{source\_send})} (\alpha, \mathbf{Pr}(P\{\texttt{sseq}/\mathit{arg}_1\} \mid \texttt{sseq} - \texttt{pseq} \in [0,0]).r_{call}).\mathit{ReturnStateB} \\ \mathit{ReturnStateB} & \stackrel{def}{=} \sum_i (\mathit{return}_{\texttt{source\_send},i}, \top).\mathit{StateD} \\ \mathit{StateC} & \stackrel{def}{=} \sum_{k=0}^{|\mathcal{R}|} (\mathit{return}_{\texttt{source\_recv},k}, \mathbf{Pr}(\mathcal{R}_k\{1/X\} \mid \mathcal{R}_l \wedge \texttt{sseq} - \texttt{pseq} \in (0,\infty)).r_2).\mathit{Init} \\ \mathit{StateD} & \stackrel{def}{=} \sum_{k=0}^{|\mathcal{R}|} (\mathit{return}_{\texttt{source\_recv},k}, \mathbf{Pr}(\mathcal{R}_k\{1/X\} \mid \mathcal{R}_l \wedge \texttt{sseq} - \texttt{pseq} \in [0,0]).r_2).\mathit{Init} \end{array}$$

Note that the probability calculations in *CallStateA* and *CallStateB* are *functional rates*, since they depend upon the state of the sink. Due to space considerations, we cannot describe this process in detail, but it is essentially the same as that used for transmitting the return value of a function call.

Constructing a similar model for the sink, and composing these together with a

network and timer as described in Section 4, we can build a complete model of the system. Some results from the analysis are shown in Figure 4. The second graph shows how the timeout rate affects the throughput; if we timeout too slowly, we have to wait a long time after a packet was lost to get the retransmission, but if we timeout too quickly, we send too many unnecessary retransmissions, therefore wasting bandwidth.

7 Conclusions

In this paper, we presented an abstract interpretation from source code to a performance model. There is still much work to be done in formalising this with respect to the *errors* involved, and it seems that a proper formulation in the context of abstract interpretation [5] would be appropriate. Furthermore, there is a great deal of scope to apply existing simplification techniques [4,6,14] to reduce the size of the Markov models we generate.

The ultimate aim of this work is to produce a tool for semi-automatic derivation of performance models from real code. We did not mention any implementation details in this paper, but at the time of writing, we are completing a prototype implementation of the abstractions described, in the context of network simulator `ns-2` [13] agents. This will allow us to validate the models we generate against simulation results, and is the first step towards a tool that can deal with 'native' protocols written in C.

Whilst this work is still in its early stages, the abstraction techniques we have considered seem to be feasible, and future work looks to be promising. This is certainly a tool that is needed, and would be widely appreciated by both the software engineering and performance evaluation communities. Although there are many challenges yet to be faced, we have taken the first few steps, and look forward to continuing along this path.

References

[1] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *EuroSys'06: European Systems Conference*, 2006.

[2] T. Ball and S. K. Rajamani. The SLAM toolkit. In *Conference on Computer Aided Verification*, 2001.

[3] C. Canevet, S. Gilmore, J. Hillston, M. Prowse, and P. Stevens. Performance modelling with UML and stochastic process algebras. *IEE Proceedings: Computers and Digital Techniques*, 150(2):107–120, March 2003.

[4] W. Cao and W. J. Stewart. Iterative aggregation/disaggregation techniques for nearly uncoupled Markov chains. *Journal of the ACM*, 32(3):702–719, July 1985.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[6] T. Dayar and W. J. Stewart. Quasi lumpability, lower-bounding coupling matrices, and nearly completely decomposable markov chains. *SIAM J. Matrix Anal. Appl.*, 18(2):482–498, 1997.

[7] V. Firus, S. Becker, and J. Happe. Parametric performance contracts for QML-specified software components. In *Proceedings of Formal Foundations of Embedded Software and Component-Based Software Architectures*, Electronic Notes in Theoretical Computer Science. ETAPS 2005, 2005.

[8] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with Blast. In *Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN), Lecture Notes in Computer Science 2648*, pages 235–239. Springer-Verlag, 2003.

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

[10] J. Hillston and L. Kloul. Formal techniques for performance analysis: Blending SAN and PEPA. *Formal Aspects of Computing*, 2006.

[11] I. T. Jolliffe. *Principle Component Analysis*. Springer Series in Statistics. Springer, 2002.

[12] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[13] S. McCanne and S. Floyd. The Network Simulator, `ns-2`. `http://www.isi.edu/nsnam/ns`.

[14] V. Mertsiotakis. *Approximate Analysis Methods for Stochastic Process Algebras*. PhD thesis, Institut für Mathematische Maschinen und Datenverarbeitung, 1998.

[15] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of Conference on Compiler Construction*, pages 213–228, 2002.

[16] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the Principles of Programming Languages*, 2002.

[17] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of 15th ACM Symposium on Principles of Programming Languages*, pages 12–27. ACM Press, 1988.