Towards Stochastic Model Extraction

Performance Evaluation, Fresh from the Source

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

Laboratory for Foundations of Computer Science,
School of Informatics, University of Edinburgh,
James Clerk Maxwell Building, King's Buildings,
Mayfield Road, Edinburgh, EH9 3JZ

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 distributed systems. The central aim is to extract a stochastic model (in the PEPA language) from such source code.

We present interval abstraction as a technique for dealing with integer variables, and path abstraction for representing control-flow. To bring these together, a framework for deriving probabilities of control-flow decisions is presented, along with a procedure for constructing a PEPA model. Finally, we discuss possible refinement techniques, and give an brief description of future work.

1 Introduction

The typical computer program is a mysterious beast. Armed with the most basic of operations, and marching down a deterministic path laid out by the programmer, it seems as though its fate is predetermined. Yet to an outside observer, its actions may appear randomised, unpredictable, and even contradictory to its supposed purpose. Allowing programs to interact only makes their behaviour harder to predict, and even a seemingly well-behaved program can suddenly change for the worst if certain conditions are met.

The usual approach to modelling such observable behaviour, is to treat the system as being *stochastic* rather than deterministic. Doing so places a whole body of mathematics at our disposal, so that modelling the system becomes not only feasible, but practical. During the last ten years, traditional techniques for modelling concurrent processes, such as process algebras and Petri nets, have been applied to the stochastic setting with great success. A process algebra in particular is a formal language in which models can be built compositionally, with support for *abstraction*. This allows scalability as the size of the model increases. We will restrict our attention, from hereon, to the Performance Evaluation Process Algebra (PEPA) [10], due to its wide existing toolbase and user community.

To build a stochastic model, the most common strategy by far is one of iterative refinement. In other words, a simple model is initially proposed, then analysed to compare its behaviour to that of the real system (either by observation, or by what we expect). Since the model is virtually never satisfactory on the first attempt, it is refined by increasingly adding detail. This continues until we believe the model to be suitably accurate, or else are limited by the size of model we can handle.

This scientific approach, of suggesting a model and validating it by empirical evidence, seems suitable to certain application domains, such as biological systems modelling. However, when we look at a computer system, for which an implementation exists, we immediately have more information at our disposal – namely the source code. In particular, the notion of 'believing a model to be suitably accurate' seems highly unsatisfactory, and we ought to be capable of being more precise.

Indeed, the jump from abstract model to implementation has already happened in the world of *qualitative* model checking. Here, we are not concerned with the quantitative aspects of a system, such as the probability of an event happening, or its performance characteristics. Instead, we deal only with the *possibility* of a certain behaviour happening. This is used in the negative sense – to prove that the system satisfies some property, we prove that it is *not* possible to reach a state that *does not* satisfy the property.

Initially, such model checking occurred at the level of an abstract modelling language (e.g. process calculi, or richer languages such as SPIN's Promela [13]), along with a temporal logic for expressing properties (e.g. CTL, LTL). The obvious limitation is when we want to verify that some property holds of our *implementation*. In particular, some interesting properties *only make sense* in the context of the implementation, such as ensuring that a pointer to memory is not freed twice in a C program.

To automatically derive a model from the source code, the big advance was in techniques for *abstraction*. The state space of such code is much too large to deal with directly, hence information must be thrown away to leave us with a model small enough to be verified. The trick is knowing what information is important to keep, and what can be safely discarded. Two approaches of note are the following:

1. *Extended static checking*, as exemplified by ESC/Java [6], is dependent on user annotations. These take the form of assertions in the Java Modelling language (JML) [15]. The tool translates these assertions, and the code itself, into an intermediate language of guarded commands, from which it can generate a set of verification conditions. These are then passed to a theorem prover for verification. Loops are dealt with by unfolding a fixed number of times (specified as a parameter), to avoid having to compute fixed points.

   It should be pointed out that ESC/Java is neither sound nor complete, and intentionally so – it was designed as a pragmatic tool for finding bugs, not for proving correctness.

2. *Predicate abstraction and counter-example-guided refinement*, as exemplified by SLAM [2] and Blast [9], works by iteratively refining an over-approximation of the program (in both cases, in C). To describe a property, the user must specify a set of state variables (over a finite domain) and a finite state machine (FSM), containing one or more error states, that describes changes to this state. The program is then instrumented by inserting appropriate calls to the FSM.

   Predicate abstraction reduces the instrumented program to a boolean program, by throwing away all data except for predicates on the state variables. If a control-flow decision cannot be expressed in terms of these predicates, it is made non-deterministic. The resulting program has more behaviours than the original, so if a path to an error state is found, we must check whether the path is allowed by the program (this requires the use of a theorem prover). If so, we have a counter-example to the property holding, and if not, more predicates must be added to the abstract program to disallow the path. This process continues until no error state can be reached in the abstract program (hence they cannot be reached in the original), or a true counter-example is found.

Of these approaches, it is difficult to see how to extend either to a quantitative setting. Indeed, state-of-the-art quantitative model checkers, such as PRISM [14], only support relatively high-level modelling languages (e.g. PEPA). As it stands, the technique used by ESC/Java is inappropriate, since it throws away any looping (which is significant when modelling execution time), and is highly modular (performance properties are typically of a global nature). In any case, it is not clear what an appropriate annotation would be. Predicate abstraction, on the other hand, throws away too much information initially; control-flow decisions depend on the values of local variables, and this is vital to modelling the program's performance. There is no obvious way to guide the refinement of such an abstract model, at least without resorting to dynamic techniques such as code profiling. Some ideas from predicate abstraction will prove to be useful, however, as we shall see later.

1.1 Motivation

Before we begin to address this problem, we must first answer the obvious question – why do it at all? What need is there to extract a *stochastic* model from source code? We have already identified that such a model is a prerequisite to any non-trivial static analysis of a program's performance, but what is the benefit of this compared to dynamic methods?

It is true that profiling can be used to identify frequently executed code for optimisation, but this is rather crude, since we can only measure the frequency of execution along a particular trace. Furthermore, it does not easily extend to distributed programming, as we cannot use profiling to measure delays across the network, or response times under different loads. If we have a complete implementation that can be executed and monitored, we can use operational analysis to do the latter, but then we can only *describe* the behaviour, rather than giving a model to *explain* it. A simulation of the system can yield more information, but there is no formal basis on which to argue its accuracy. Even if we turn to more formal methods, and develop a separate performance model, the relationship with the implementation is similarly hard to reason about. Furthermore, the knowledge and experience required to correctly do this limits its application to mainstream programming.

The ability to automatically extract a model from source code has the following benefits:

1. *Wider application of performance analysis* – it is surprisingly difficult to encourage the application of theoretical techniques in industry, even when the benefits are seemingly clear. This is largely due to the lack of specialist/technical knowledge required to apply such techniques successfully, amongst industrial developers. However, if we can provide access to these techniques via a *tool*, which the developer can interact with by more familiar means, then they are considerably more likely to be applied in practice. This is exemplified by the success of the DEGAS project [8] (which developed a tool for performance analysis of UML), and of Microsoft®'s Static Driver Verifier (SDV) [1] (a tool for automatically verifying certain required properties of Windows device drivers, using SLAM).

   The ultimate aim of our work is to provide such a tool for performance analysis of distributed systems, so that we can open up performance modelling to a wider audience.

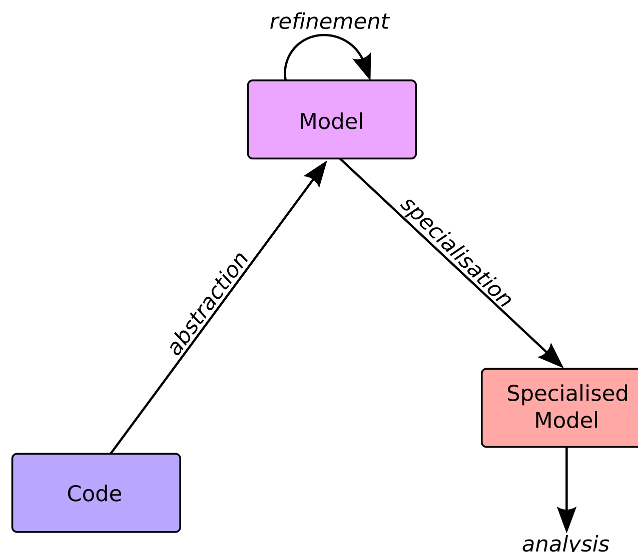
2. *Non-functional testing* – a major problem faced by programmers is how to incorporate non-functional testing into the development cycle. In particular, this is usually left until very late in the process, when the structure of the code is no longer malleable. The ability to derive a performance model from the code would mean that incomplete programs could be analysed, allowing performance criteria to guide the development process from an early stage.

3. *Performance guarantees* – for many distributed systems, such as web services, performance properties such as response time and scalability are vitally important, yet the resources for formal modelling and verification are not available. Such properties are specified by Service Level Agreements (SLAs). If we can automatically derive performance bounds for certain classes of program (to within a given error), then these are much more likely to be used in practice. For example, we could imagine certifying a web service with a performance contract, which would specify its conformance to an SLA.

   Indeed existing work, such as that in [5], instruments performance contracts at a more abstract level. Control-flow is abstracted to a 'service effect automaton', describing the structure of calls to other functions/services, while ignoring the data-flow. The duration of each call is given by a random variable, and the duration of the entire automaton by composition of these. Using the Quality of Service Modelling Language (QML) [7], properties of the required execution-time distribution can be specified and then verified. This work, however, does not take into account how data dependencies influence performance, and furthermore requires the service effect automaton to be pre-annotated with the probabilities of control-flow decision points.

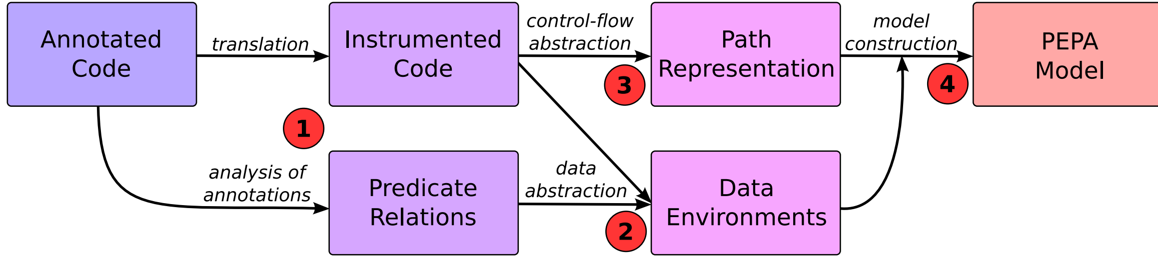
1.2 The General Approach

It seems clear from the outset that any attempt to produce a model from a given program in a *single pass* is going to fail. There is simply too much information to handle to be able to produce a manageably sized model. It is inevitable, therefore, that any general strategy will involve a series of *refinements*, before arriving at a final model. Unlike SLAM, however, it is not obvious how to refine a very crude abstraction by gradually adding more information. Instead, we take the opposite approach; keep as much information as possible when building the original model, then look to simplify it afterwards. A general view of this procedure is shown below:



Note that, in general, the model we produce in the initial abstraction needs to be *refined* (see Section 7) before it is small enough to be used. Furthermore, this model is of the *entire* program, whereas for a particular analysis, certain parts of the program are more relevant than others. We would therefore want to *specialise* the model before analysing it, based on information in user annotations, or from more explicit instructions from the user, in the form of a query language.

The subject of this paper, however, is the first (and in some ways most important) step of the above procedure; that of abstracting the code to an initial model. This in itself can be separated into four distinct stages, as shown in the diagram below:



1. We analyse the *annotations* in the source code to produce an instrumented version of the code, which has additional instructions to encode this information. The user may also specify the behaviour of function calls that are too complex to analyse, or whose code is unavailable. This information is extracted to a separate file, in the form of probabilistic relations on the program's predicates.

2. Using the predicates identified by the user, and a static analysis of the instrumented code, we construct an abstraction of the program's *data environment*. Primarily, we use a combination of predicate and interval abstraction.

3. We abstract the program's *control-flow* into sequential paths, each of which has a single condition that holds along it, and a number of entry points.

4. Finally, we produce a *PEPA model* whose states are the control-flow paths augmented with the abstracted data environments. In determining the rates, we must calculate the probability of each path being entered, which in the most general case requires Monte Carlo methods.

In Sections 3 to 6, we describe each of the above stages in turn. Prior to this, we describe the languages we intend to target in Section 2. We briefly describe potential approaches to model refinement and specialisation in Section 7, before concluding with a discussion of future work in Section 8.

2 Target Languages

Since we aim to provide a tool suitable for real-world applications, we must deal with the programming languages used in practice. In particular, for communications protocols, the predominant language is C. There exist a number of tools for handling C, such as CCured [19], which together with the C Intermediate Language (CIL) [18] provides a cleaner (and type-safe) framework for analysis.

However, even with such tools at our disposal, we cannot hope to analyse arbitrarily complex programs. We emphasise that we are not trying to analyse the average-case time complexity of algorithms. Rather, we are interested in the behaviour of highly distributed systems, where the important delays are due to *communication*. Most real programs of this nature have a relatively simple control-flow structure, which allows us to place constraints on the code we can analyse, without significantly restricting the class of programs we consider.

In this paper, we describe the analysis of a subset of C, with only integer variables, boolean variables[1] and enumeration types. In addition, we impose the following restrictions:

1. *No pointers.* We have yet to develop a framework for dealing with pointers, and pointer arithmetic. Although we intend to do so (for simple cases) in the future, this is beyond the scope of our work at present.

[1]In C, booleans are represented as integers; it is their *use* as a boolean that we refer to here.

2. *No recursion.* In our abstraction, there is a single environment space for variables. When we call a function, we need to remember the environment we called from, due to the memoryless property of Markov chains. If we did not, there would be no way to recover it when we return. Because of this, we cannot model a stack of environments, which is needed for recursion.

3. *We allow only linear conditions.* Specifically, a condition must be expressible in the form:

$$\sum_{i=1}^{n} a_i x_i \; \{<, \leq, =, \geq, >\} \; c$$

   where a_i and c are constants. In particular, this means that we can only assign a variable to a *linear* sum of other variables, if it is used in a subsequent condition. The reason for this restriction will become more apparent in subsequent sections.

4. *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. Note that, without any information from the user, we assume that the return value of a function is independent from its arguments.

The extent to which these constraints can be relaxed is the subject of future work.

The target of our abstraction is a PEPA model. PEPA is a high-level language, with multiple alternative semantics, and many performance analysis tools accept it as input. This avoids unnecessarily limiting ourselves to a specific formalism (hence limiting the analyses we can perform), as would happen if we generated a continuous-time Markov chain (CTMC) directly. We assume familiarity with the PEPA language, and refer the reader to [10] for more information.

3 User Annotations

Although our aim is to extract a model automatically, there is no way to do this without some guidance from the user. We require that they provide information, in the form of source code annotations, for a number of reasons:

1. To identify, and label, parts of the program state that we wish to refer to in our queries.

2. To trap certain function calls that we do not wish to model. For example, in a communications protocol, we need to trap the transmission of a packet at some level – it would not be sensible for our model to include all the details of the network card's device driver. Similarly, we do not want to explicitly analyse calls to auxiliary or library functions unless it is essential to model their detailed behaviour.

3. To allow the user to provide timing information; for example to artificially insert delays into the model, or to avoid the tool having to perform expensive profiling.

3.1 Syntax of Annotations

Annotations are embedded within comments, and must not appear within a statement of the program. In other words, we treat them as sequential statements within the program. The syntax of annotations is shown below. Here, P and I are identifiers corresponding to predicates and intervals respectively, and have separate namespaces to the variables in the code. Integer constants are given by c, and probabilities by p.

Annotation	:=	`//@ predicate` P `=` *BoolExpr*
	|	`//@` P `=` p
	|	`//@` P `=` $f(P'_1, \ldots, P'_m)$
	|	`//@ delay` c
	|	`//@ follow`
	|	`//@ nofollow`
	|	`//@ synchronise` *FName*
BoolExpr	:=	P
	|	`expr(`*source_code_expr*`)`
	|	`~` *BoolExpr*
	|	$BoolExpr_1$ `&&` $BoolExpr_2$
	|	$BoolExpr_1$ `||` $BoolExpr_2$

Here, we allow *relational expressions*, which define the probability of a predicate holding. Note that these can refer to functions, f, which take a vector of predicates as input and return a probability. Their general syntax is as follows:

```
function f(P1,...,Pn) =
        guard1   ->   p1
     | ...
     | guardk   ->   pk
     | _        ->   pk+1
```

As we will see later, the data environment completely determines the truth of a function's predicates. This means that, in a given environment, the user can specify that a certain predicate holds with fixed probability. For example, in TCP, if the predicate `fast_path` determines whether or not a packet is processed using the common-case 'fast path', consider the assertion:

```
//@ fast_path = f_fast_path(checksum_ok, seq_num_inorder)
```

where the function `f_fast_path` is defined as:

```
function f_fast_path(checksum_ok, seq_num_inorder)
      checksum_ok && seq_num_inorder   ->  0.9
    | _                                ->  0.0
```

This states that the fast path will be entered with a probability of 0.9, if both the checksum is correct, and the sequence number is in-order. Otherwise, the fast path will definitely *not* be entered.

We only allow the user to define the truth of *atomic* predicates, as defined in Section 4. There is no restriction on what predicates can be *defined*, however.

3.2 Analysis of Annotations

We analyse the user's annotations in order to produce an *instrumented* version of the code, such that the information they provide is encapsulated by a number of additional instructions. Further to this, we place relational information (when the user provides their own description of a function's behaviour) into a separate file to be used later. The reason for doing this is partly pragmatic, since we are using existing tools to parse the code (e.g. CIL), and these ignore any comments in the code.

There are four categories of annotation, each of which is treated differently:

1. Definitions of predicates (see Section 4) – these are converted into declarations of new variables within the code, renamed so as to be easily identifiable by later stages of the analysis (e.g. the predicate `checksum_ok` would become `PREDICATE_checksum_ok`.

2. Relational expressions – these define (probabilistically) the truth value of a vector of predicates, and are removed from the code, and placed in a separate file for use later.

3. Delays – the user can specify a delay (in milliseconds) that is inserted into the model. In the instrumented code, we represent this as a call to the function `delay`, with the duration as the argument.

4. Function modes – as we will discuss in more detail in Section 6.1, there are three ways to analyse a function. Either we construct a model explicitly (`follow`), or we do not analyse the function, and rely on the user to provide its behaviour (`nofollow`). The third alternative is to synchronise the function with another (`synchronise` *FName*), which must be specified by the user. This allows, for example, a call to a method for sending a packet to be synchronised with a remote receive method. The insertion of a function mode annotation causes all subsequent function calls within the current scope to be treated in that way.

   If the call is not to be analysed, we leave it in its original form, otherwise we insert a wrapper function around it to indicate how we should analyse it. In the case of '`follow`', we rename a function `f` to `FOLLOW_f`, and in the case of '`synchronise` *FName*' we rename to `SYNC_`*FName*.

When performing a transformation of this sort, we must be careful about its correctness. In this case, the transformation is very simple, and the control-flow of the original program is not altered. Indeed, the only thing we must be careful of, is if the definition of a predicate or interval would cause an exception to be thrown. However, we disallow function calls within such definitions, so the only exceptions possible are arithmetic (such as divide-by-zero), which are easy to spot.

4 Abstraction of Data

The biggest problem by far with modelling a program is managing the size of its state space. Even a small program can have hundreds of local variables, and when each is 32-bit integer, the size of this state space alone is astronomical. To build a model that is small enough to analyse, we must restrict both the *number* of variables that comprise the state, and the *values* that these variables can take.

Any accurate performance model of a program needs to model the control-flow decisions taken, and these in turn depend on the values of its variables (the concrete environment) when a condition is evaluated. However, from the point of view of our model, all concrete environments that produce the same control-flow decisions are equivalent. We therefore have an *abstract environment*, which consists of the truth of all control-flow-determining predicates.

This leads to the *predicate* as our fundamental unit of data abstraction. In particular, an *atomic predicate* is one of the following:

1. A boolean variable.
2. An equality test on an enumeration type.
3. A linear condition on integer variables.

The first and second of these are mapped directly onto the abstract environment. Boolean and enumeration type variables have precisely the same domain in the abstract environment as in the concrete. Conditions on integer variables are more complex, since there may in general be subtle dependencies between them. To handle this, we use *interval abstraction*, which we describe in Section 4.2.

We determine the abstract environment of a function by decomposing its control-flow-determining predicates into the above atoms. These must be in terms of the function's *independent variables*, which we define in Section 4.1. A compound condition (conjunction, disjunction and negation of atoms) simply defines a set of abstract environments.

4.1 Variable Dependencies

Consider the following code fragment:

```
x = y;
if (x > y) {
  C
}
```

We are interested in determining the probability that C will be executed. If we look at the condition '`x < y`' in isolation, we know nothing about the dependency between `x` and `y`, and so we cannot reason about its truth. In this case, since `x` and `y` are correlated, the probability of executing C is precisely zero.

For a function, we define its *independent variables* to be the following:

1. The arguments to the function.
2. The return value of each function call that is made.
3. Loop variables – those whose definition reaches over the backward branch of a loop.

We assume that these variables are independent, in the sense that they are not defined in terms of one another. We may, however, still have *predicates* on these variables, but this must form part of the environment. For instance, a function call may introduce a relationship between its arguments and return value, but we do not model this unless it affects a subsequent control-flow decision.

To solve the problem illustrated by the example above, we need to express all conditions in terms of these independent variables. We do this by first transforming the function into static single assignment (SSA) form [21], such that each variable is assigned to only once (when the function is viewed statically). After doing so, the code essentially becomes declarative, so we can repeatedly substitute the variables in an expression for their definition, until only independent variables remain. Of course, we can still introduce conflicting definitions at join nodes (i.e. when the two branches of a conditional rejoin), but this will be handled by our path abstraction, described in Section 5.

4.2 Interval Abstraction

In an idealised world, an integer variable takes values from the countably infinite set $\mathbb{Z}$. We can abstract this by considering only whether a value lies within a *subset* of $\mathbb{Z}$, for some finite collection of subsets. Since arbitrary subsets are difficult to specify, we restrict ourselves to contiguous sets, or *intervals*, over $\mathbb{Z}$. More formally, define:

- A *point* is an element of the set $\mathbb{Z} \cup \{\infty, -\infty\}$.
- An *interval* is a pair of points, $[\underline{x}, \overline{x}]$, such that:

$$[\underline{x}, \overline{x}] = \{z \in \mathbb{Z} | \underline{x} \leq z \leq \overline{x}\}$$

- An *interval space* is a set I of disjoint intervals, such that:

$$\bigcup_{\iota \in I} \iota = \mathbb{Z}$$

Note that a constant c can be represented as the interval $[c, c]$. If an expression e is in the interval $[a, b]$, this implicitly means that the concrete value of e is *uniformly distributed* over the elements of $[a, b]$. If this interval is not finite (i.e. $a = -\infty$ or $b = \infty$), then the probability of taking on any particular value is zero. In fact, the probability of the expression being in any finite subset of an infinite interval is zero.

Due to our restriction on the structure of conditions, we remind ourselves that any atomic predicate on the integer variables of a function can be expressed as follows:

$$\sum_{i=1}^{n} a_i x_i \ \{<, \leq, =, \geq, >\} \ c$$

where a_i and c are constants, and x_i are independent variables. In particular, if $\mathbf{x}$ is the vector of all the independent integer variables, then we can write all predicates in the form:

$$\mathbf{a} \cdot \mathbf{x} \ \{<, \leq, =, \geq, >\} \ c$$

where $\mathbf{a}$ is now a constant vector. Let us now group all predicates with the same vector $\mathbf{a}$. The values of c define an interval space in the obvious way. For example, if we have the following predicates:

$$x - y < 10, \quad x - y \geq 20, \quad x - y = 0$$

then the interval space they define is:

$$x - y \in \{[-\infty, -1], [0, 0], [1, 9], [10, 19], [20, \infty]\}$$

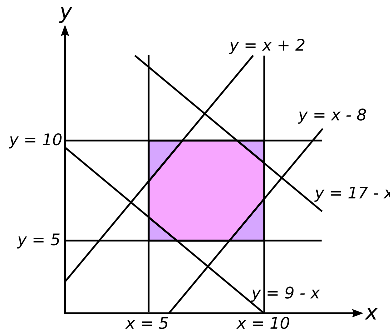
Now, in order to define the abstract environment, we need to determine which conditions are independent from one another. This corresponds to *orthogonality* between the vectors $\mathbf{a}$. For example, the conditions $x < 0$ and $y < 0$ (with $x \neq y$) are independent, since $\mathbf{Pr}(x < 0 \mid y < 0) = \mathbf{Pr}(x < 0)$. Similarly, $x < y$ and $x < -y$ are independent, assuming that we know nothing about the *values* that x and y can take.

In general, not all conditions will be independent, so we define a *hierarchy* of abstract environments. In each level of this hierarchy, the environment consists of a set of mutually independent expressions, and the domain of each is the interval space of its predicates, as defined above. To find the top-level environment, take the maximally orthogonal set of $\mathbf{a}$-vectors, A, such that:

$$\mathbf{a} \in A \Leftrightarrow \forall \mathbf{b} \in A \backslash \{\mathbf{a}\}.\ \mathbf{a} \cdot \mathbf{b} = 0$$

These are now removed from the pool of predicates, and we repeat the procedure to find each level of the environment, until no predicates remain. The actual abstract environment space is then the conjunction of all the environment levels.

We can see the nature of these environments better if we picture them geometrically. Consider the following, where there are two environment levels:



Here, the central shaded area represents the following abstract environment:

$$(x, y) \in ([5, 10], [5, 10]) \ \wedge \ (y + x, y - x) \in ([-8, 2], [9, 17])$$

5 Abstraction of Control-Flow

Having considered how to abstract the data environment of a program, we now turn to abstracting its control-flow. In the concrete system, there is a separate state for each statement, corresponding to the value of the program counter. Initially, we note that we can abstract this down to *basic blocks*. A basic block is a piece of sequential code which contains no branches out of or into the middle of it. In other words, once execution enters a basic block, all instructions are executed (in order) before it is exited. A *maximal basic block* is one which is as large as possible; its first instruction can accept control from more than one point, and its final instruction can transfer control to more than one point. Note that a function call ends a basic block (and the return from a call starts a basic block), since we cannot guarantee that it will return.

Modelling at the level of basic blocks, however, is not enough, since the states in a PEPA model are memoryless – the transition rates out of a state are independent of how we got there. This is contrary to real code, where previous control-flow decisions change the data environment, and therefore affect future decisions. Consider the following, for example:

```
if (x > y) {
  y = 1;
} else {
  y = -1;
}
if (y > 0) {
  C
  ...
```

Here, we cannot statically determine which definition of `y` will reach the second `if`-statement. In other words, we cannot compute the probability of executing command C without knowing which branch of the first `if`-statement was taken.

We solve this problem by abstracting control-flow at the level of *paths*, and taking the conjunction of the conditions along it. We then have a single probability of entering the entire path, rather than computing a separate probability at each decision point. By having all the conditions together at the same time, we can compute their combined probability more accurately, since we need not assume independence (see Section 6.3).

An *acyclic internal path* on a control-flow graph is a path from the input to the function, or the return from a function call, to either a return instruction, a function call, or the backward branch of a loop. The *path condition* is the conjunction of all the conditions along the path, which is logically tested before entering it. Because such paths are internal, i.e. they do not contain any calls to other functions, all the conditions along it can be expressed in terms of the independent variables that are known on entry to the path. Hence it is safe to lift these conditions to the start of the path.

At the end of the path, we record how the independent variables are updated. In general, we compute the probability of the each environment holding (with the variables substituted for their updated values), given the current environment. Each data environment uniquely defines a path, since the atomic predicates that comprise the path condition are precisely those in the environment. We describe how to compute these probabilities in Section 6.3.

We have to be careful about reentry into a path after looping, since part of the path's duration was due to instructions *before* the loop, which should not be repeated. Hence some paths must have more than one state

along them. More precisely, a path along which n loop guards are true will have $n+1$ states. This can be seen more clearly with an example:

```
B1
while c1 {
  B2
  while c2 {
    B3
  }
}
```

Here, the path along which c_1 and c_2 are both true must have three states, corresponding to the executions of B_1, B_2 and B_3 respectively. If we are in the inner-most loop, we need to be able to execute B_3 without executing B_1 and B_2, and if we are in the outer-most loop, we execute B_2 and B_3 without executing B_1.

We have described the amalgamation of multiple control-flow states along a path, but have not yet justified doing so. It seems sensible to assume that the duration of a single instruction follows an exponential distribution, since although it will usually execute very quickly, there is a small probability of it taking much longer – for example, if the pipeline stalls, or if the operating system preempts the execution. Thus, if a path contains k sequential instructions, its duration will follow a k-stage Erlang distribution.

We can represent this as a single state if the underlying stochastic process is *insensitive* to the shape the distribution, in the sense that the steady state solution depends only on its mean. If the path is executed in isolation, this is true – we either execute *all* the instructions along the path, or *none*. Insensitivity fails when the residual duration of an action comes into play. When there is a choice between two actions, they effectively race, such that the one completing first gets to proceed. When this happens, we must remember the residual time of the other action, so that in the next state, it can continue from where it left off, rather than starting from the beginning. If the distribution is exponential, the residual duration follows the same distribution, due to the memoryless property.

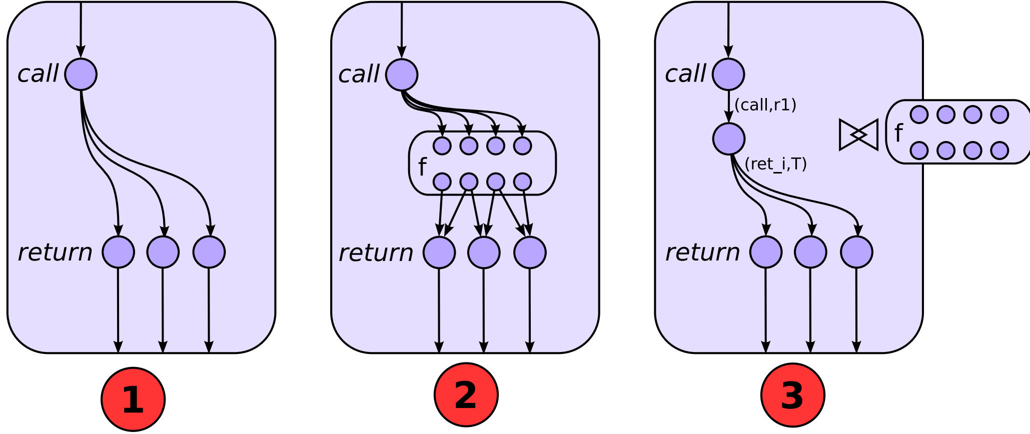
Unfortunately, when we run two processes in parallel, we introduce choice due to the interleaved semantics. This means that the durations of the paths are no longer insensitive, since their execution may be interrupted (in the Markov model) by the completion of another path running in parallel. We are therefore forced to assume that basic blocks follow an exponential distribution in order to justify our use of it. However, as with single instructions, there are many factors that determine a path's execution time, so this still seems to be a reasonable assumption.

6 Constructing a PEPA Model

Now that we have both data and control-flow abstractions, we can bring this together 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 in Section 6.3. Before we do so, we describe some particular features of modelling function calls and loops, in Sections 6.1 and 6.2 respectively.

6.1 Modelling Function Calls

There are three fundamental choices when modelling a function call, as the following diagram illustrates:



1. 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. In particular, when the user completely defines the behaviour of the function in their annotations, we use this method.

2. 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 is undesirable unless essential to the behaviour of the model.

3. 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. In general, we will only use this abstraction when two components are communicating over the network.

In most cases, we favour the first abstraction for its simplicity, and we will only use the second if the user explicitly requests it. Of interest to distributed systems is the third option, which relies on the function having a *public interface* of *call* and *return* actions. Each action is labelled with the environment being communicated, and so the function can define what predicates on its arguments are of interest to it.

6.2 Modelling Loops

In the control-flow abstraction we have described, we only represent a single iteration of a loop. Subsequent iterations are modelled by transitions leading back to the start of the loop. If the probability of exiting from the loop is p, the number of iterations is given by the random variable X:

$$\mathbf{Pr}(X = n) = (1 - p)^{n-1} p$$

This is a geometric distribution, with expectation and variance as follows:

$$\mathbb{E}(X) = \frac{1}{p}, \quad \mathrm{Var}(X) = \frac{1 - p}{p^2}$$

Our challenge is to set p such that it gives the correct expectation $\mathbb{E}(X)$, with respect to the concrete program. If we wish to perform a transient analysis, however, we must also worry about the variance. Unfortunately, since p is often quite small (e.g. consider a `for`-loop with 1000 iterations), the variance can be huge. In some cases, we can flatten out the loop into a single state – namely, if its body is entirely sequential, with no conditionals or function calls, so that we can partially evaluate it. Otherwise, we can unroll the loop n times, to increase p by a factor of n and give the following reduction in variance:

$$\frac{\mathrm{Var}'}{\mathrm{Var}} = \frac{1 - np}{n - np}$$

There is a trade-off between the gain in accuracy, versus the increase in the size of the model.

Due to the restriction on loop variables we placed in Section 2, they fall into the following two cases:

1. The loop variable is set *independently* of which iteration of the loop we are in. In other words, the probability of re-entering the loop is independent of how many times we have executed it already. Since this already satisfies the memoryless property, we can model such variables directly, without modification.

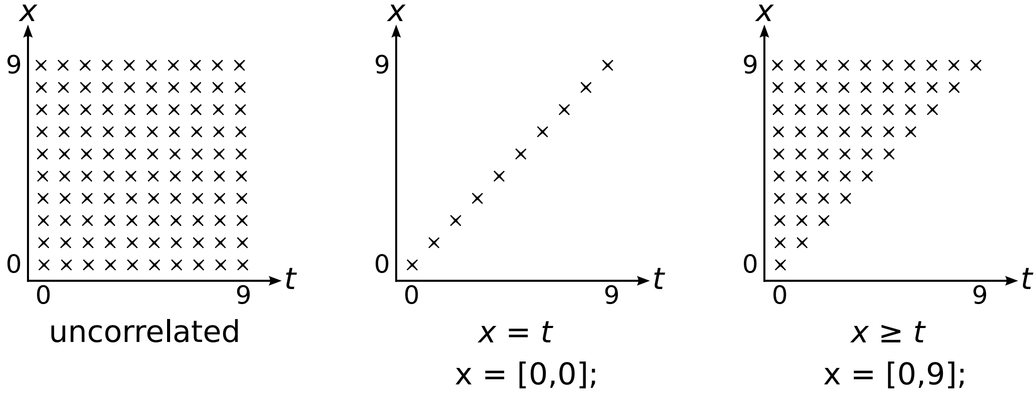
2. The loop variable is incremented or decremented by a *constant* on each iteration. The reason for this restriction is to ensure that the values the variable takes are *uniformly* distributed, and increase/decrease *monotonically*.

If we consider a loop variable x of this second case, we have the following structure:

```
x = [a,b];
while (x {<,>} c) {
  ...
  x += i;
}
```

Note that x can be initialised to an interval, rather than just a constant, as it can in general be assigned to an expression of the independent variables. If i is positive and the condition is '$x < c$', the environment of x within the loop will be $[a, c-1]$, otherwise it will be $[c+1, b]$ (assuming $i \neq 0$). The way we derive this is similar to the work on value-range propagation [20], except that we do not simply predict branch probabilities in isolation.

One problem with such loop variables is that they combine both *spatial* and *temporal* abstraction. If we take the above structure, with $i = 1$ and the condition '$x < 10$', we can observe what happens as we change the starting interval, where t is the number of loop iterations that have taken place:



The uncorrelated case corresponds to x being assigned to the interval $[0, 9]$ independently on every loop iteration, and is essentially what we abstract to. In reality, if x is initialised to zero, it will be correlated with t. However, if we initialise x to $[0, 9]$, and then proceed to increment x linearly, the uncorrelated abstraction fails. The solution is to introduce t as an auxiliary variable, and keep track of the constraint between x and t. Since the loop variable changes monotonically, this constraint will always be linear.

In the example here, we calculate the loop exit probability p as:

$$\begin{aligned} p &= \mathbf{Pr}(x = 9 \mid x \in [0,9] \wedge t \in [0,9] \wedge x \geq t) \\ &= \frac{\mathbf{Pr}(x = 9 \ \wedge x \geq t \mid x \in [0,9] \wedge t \in [0,9])}{\mathbf{Pr}(x \geq t \mid x \in [0,9] \wedge t \in [0,9])} \\ &= \frac{\frac{1}{10}}{\frac{11}{20}} = \frac{2}{11} \end{aligned}$$

Hence, we are approximately twice as likely to exit the loop than if we initialised x to zero, which is what we expect.

6.3 Putting It All Together

Let $\mathcal{P}$ be the set of all path states[2], and $\mathcal{A}$ be the set of all actions that a function can perform (namely the union of our own interface with the interfaces of all functions that we call). Now, for $P \in \mathcal{P}$:

- $\mathcal{C}(P)$ is the path condition.

[2]To simplify the presentation, we assume that the states corresponding to function calls have already been incorporated into $\mathcal{P}$.

- $\mathcal{R}(P)$ is the rate[3] of executing the instructions corresponding to state P.
- $\mathcal{V}(P)$ is the updated vector $\mathbf{v}$ of independent variables, after executing the instructions corresponding to state P.
- $\mathcal{X}(P) \subseteq \mathcal{A} \times \mathcal{P}$ is the set of paths states that we can move to from the current state, and the action we should use to do so.

Also, let $\mathcal{E}_T$ be the set of all top-level data environments, and $\mathcal{E}_S$ be the environment space of the remaining levels. The three sets $\mathcal{P}$, $\mathcal{E}_T$ and $\mathcal{E}_S$ are indexed by i, j and k respectively.

The states of the PEPA model are denoted $State_{i,j,k}$, where $\mathcal{P}(i)$ is the corresponding path state, $\mathcal{E}_T(j)$ the top-level environment, and $\mathcal{E}_S(k)$ the remaining environment. We define such a state as follows:

$$State_{i,j,k} \stackrel{def}{=} \sum_{(P',\alpha)\in\mathcal{X}(\mathcal{P}(i))} \sum_{E'_T\in\mathcal{E}_T} \sum_{E'_S\in\mathcal{E}_S} (\alpha, r).State_{i',j',k'}$$

where:

$$\begin{aligned} P' &= \mathcal{P}(i') \\ E'_T &= \mathcal{E}_T(j') \\ E'_S &= \mathcal{E}_S(k') \\ r &= \mathcal{R}(\mathcal{P}(i)) \times \mathbf{Pr}(\mathcal{C}(P') \mid E'_T \wedge E'_S) \times \mathbf{Pr}((E'_T \wedge E'_S)\{\mathcal{V}(\mathcal{P}(i))/\mathbf{v}\} \mid \mathcal{E}_T(j) \wedge \mathcal{E}_S(k)) \end{aligned}$$

The most important aspect in the rate computation is the final probability – this is specifying the probability of entering the new environment, given the current environment. Notice that we substitute the new values of the independent variables into the new environment, otherwise we would have zero probability of moving out of the current environment!

We can rephrase this probability as follows, so that it becomes apparent why we separated the environment into two parts:

$$\mathbf{Pr}((E'_T \wedge E'_S)\{\mathbf{v}'/\mathbf{v}\} \mid E_T \wedge E_S) = \frac{\mathbf{Pr}((E'_T \wedge E'_S)\{\mathbf{v}'/\mathbf{v}\} \wedge E_S \mid E_T)}{\mathbf{Pr}(E_S \mid E_T)}$$

The top-level environment, E_T, acts as the basis for our computations. In other words, each condition on the independent variables is expressed in terms of the expressions comprising E_T. If we view the computation of each of the above probabilities geometrically, we can see that it equates to counting the integer solutions to a set of linear constraints over a rectangular subset of an n-dimensional space (where n is the number of independent variables).

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. The advantage of this is twofold – we can sample discrete values without having to use a continuous approximation, and it extends to arbitrarily complex systems of conditions.

It is quite possible that a system of conditions defines only a small volume of the environment. In this case, if we were to sample from the entire environment, we would have to take a relatively large number of samples to estimate the probability to within the required error. To improve this, we can instead sample within the minimum sub-environment that encloses the region satisfying the condition. To get the actual probability, we just multiply by the ratio of the volume of the sub-environment to the environment.

To find this sub-environment, we calculate the intersection of each pair of conditions. In general, this will not be a single point, but a line or plane, and so we must further calculate the minimum and maximum value of each variable, within the environment. Over all the intersection points, we find the maximum and minimum value of each variable, which define our sub-environment. We can be sure of the correctness of this, since the convex hull of the region forms a subset of the intersection points, due to our restriction to linear conditions. It is clear that this computation is exponential in the number of variables and conditions, and so there is a trade-off between its expense, versus the rate of convergence of the Monte Carlo procedure.

Let N be the number of points that we sample, and n be the number for which the condition is satisfied. If the actual probability of the condition holding is p, we can treat each individual sample as being the outcome

[3]To determine this rate, we profile the sequential instructions by running them multiple times, and averaging the duration with respect to a control.

of a random variable X (with sample space $\{0,1\}$), where:

$$\mathbf{Pr}(X = x) = \begin{cases} 1-p & \text{if } x = 0 \\ p & \text{if } x = 1 \end{cases}$$

Clearly $\mathbb{E}(x) = p$. If, in a trial, we take N samples of X, for which the outcomes are x_i, then we can estimate p by:

$$q = \frac{1}{N} \sum_{i=0}^{N-1} x_i$$

Now, the variance of X is given by:

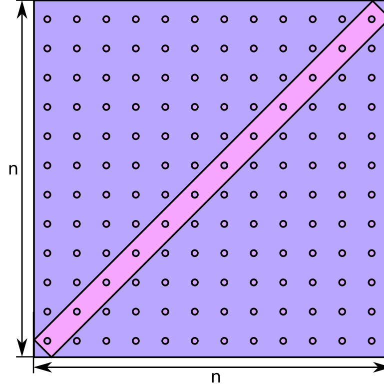
$$\sigma_X^2 = \mathbb{E}(X^2) - (\mathbb{E}(X))^2 = p - p^2 = p(1-p)$$

Since each sample is independent, the variance of q is given by:

$$\begin{aligned} \sigma_q^2 &= \mathrm{Var}\left(\frac{1}{N}\sum_{i=0}^{N-1} x_i\right) \\ &= \frac{1}{N^2}\left(\sum_{i=0}^{N-1} \sigma_X^2\right) \\ &= \frac{p(1-p)}{N} \end{aligned}$$

It can therefore be seen that the error in q is proportional to the inverse square root of the sample size. The error also depends on the actual probability p, however. For small values of p, although the absolute error in q is small, it is the relative error that we are more concerned about. In particular, if we approximate a very small probability as zero, this can lead to quite different behaviour in the model.

The worst case situation, with regard to the size of the region we are counting, is when we have a number of conditions that are parallel along some axis, and describe a small volume. This is illustrated below:



As the diagram shows, the worst case for a volume n^d is a probability of $\frac{n}{n^{1-d}}$, since only n points are in the region, and we cannot reduce the sample space any further by the simple method described above. Of course, we can construct even more degenerate cases, since we may have a disjunction of conditions that result in two small volumes that are separated by some distance. However, it is only the user that can insert disjunctions into conditions, and when we amalgamate conditions along a path, we consider only their conjunction. This limits the problem somewhat.

Now that we can build up a PEPA model of a function body, all that remains is to encapsulate it in an interface, so that it can be called by other functions. For each initial environment E that is possible on entry to the function, we define a corresponding action $call_E$. We now define an initial state $Init$, such that:

$$Init \stackrel{def}{=} \sum_{E} (call_E, \top).State_{i,j,k}$$

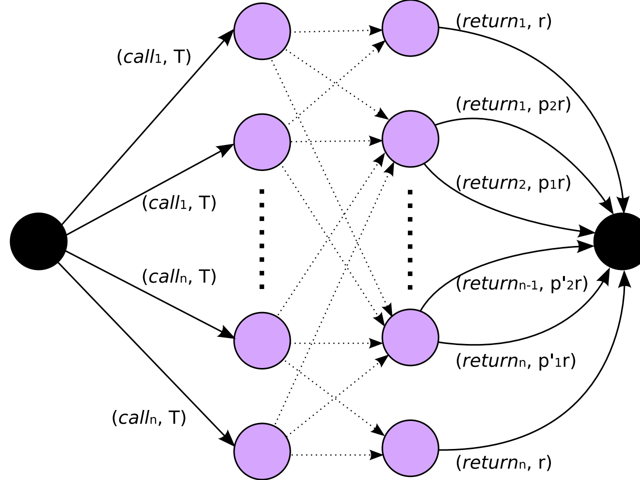
where, for $E = E_T \wedge E_S$, i, j and k are such that:

$$\begin{aligned} E_T &= \mathcal{E}_T(j) \\ E_S &= \mathcal{E}_S(k) \\ E_T \wedge E_S &\Rightarrow \mathcal{C}(\mathcal{P}(i)) \end{aligned}$$

Similarly, we define a set of return actions $return_E$, for all environments E that it is possible to return from. For a path P that ends in a return instruction, its next-state set is defined as:

$$\mathcal{X}(P) = \{(Init, return_E) \mid \mathcal{C}(P) \Rightarrow E\}$$

This interface can be seen more clearly in the following diagram:



Here the two black states are actually the same, and only separated to make the diagram readable. In other words, the function is called, executed, and returns some value (via a set of return actions in the same way as for the arguments), before returning to its initial state. This model can only handle one request at a time – if we were extending it to, for example, a server that can handle multiple requests, we simply have several instances of the function running in parallel[4].

7 Refinement and Specialisation

Until now, we have concentrated on abstraction, and how to build an initial PEPA model from the source code. Other than the functions that the user told us not to analyse, this models the entire program, and is likely to be very large. Furthermore, we are interested here in communications protocols, for which we have a number of clients running in parallel, and this causes a further exponential increase in the size of the state space. Before we analyse the model, we therefore need to both *refine* it, so that we can reduce its size as far as possible, and *specialise* it, with respect to the analysis we wish to perform.

In the context of Markovian analysis, there are a number of *approximate solution methods*, which all make use of some form of *state aggregation*. Broadly speaking, we aggregate states which exhibit similar behaviour, and then solve the abstracted model. There are a few ways to define what we mean by 'similar behaviour.' Structurally, we can look for properties such as *quasi-lumpability* [4], where we partition the states such that the rate of transition from one partition to another depends only on which partition we were in, and not on the particular state from that partition, to within some small error ϵ. Another approach is to look for *insensitivity* conditions [12], which allow a sequence of transitions to be replaced with a single exponentially-distributed transition.

Having amalgamated some of the states in the model, we can either remain in the abstract model, or approximate the solution to the original model using iterative methods. The work in [3] uses *iterative aggregation-disaggregation* to approximately solve nearly completely decomposable (NCD) Markov chains. The problem with this sort of technique, however, is that it requires us to explicitly construct the generator matrix of the Markov chain, and even then it is not easy to find the desired structure. An alternative approach more appropriate to the PEPA models we produce is *time-scale decomposition* [17]. Here we separate out 'fast' and 'slow' transitions, and build an abstract model with only the slow transitions between amalgamated states.

Our primary concern is how to deal with interacting components, and there are a number of techniques for doing so. If we want to do a Markovian analysis, we simply cannot afford to build the entire state space of any model of reasonable size. Instead, if we can identify a single interaction point[5] between two systems, we

[4]In this case, we would favour the ODE semantics of PEPA for our analysis, since a Markovian analysis would suffer from the state-space explosion problem.

[5]More precisely, a single-input, single-output (SISO) cut.

can split the model in two about that point. We then use *response-time approximation* [17] to solve one half at a time, while abstracting the other half to a single transition. In practice, this iterative technique tends to converge on a solution quite fast, but there is still some difficulty in finding suitable interaction points.

A more promising approach comes from the recent work on developing an ordinary differential equation (ODE) semantics of PEPA [11]. This avoids the state-space explosion problem by only recording the *number* of sequential components in each state, rather than precisely which component is in each state (as the Markov chain does). If we are modelling a communications protocol, or a web-service, this is precisely what we need, since it allows us to model large numbers of clients interacting, at no extra cost. The number of equations depends only on the number of states in each sequential component, so our refinement problem can be reduced to local state amalgamations.

In addition to refinement, it is useful to specialise our model for each particular analysis. For example, if we wish to determine the throughput of a protocol, we can essentially ignore individual packet-processing delays, and concentrate on how the window size changes over time. On the other hand, if we are interested in response time, packet processing will become more important; particularly when communicating with a server under high load. It is not yet clear how to provide a framework for such specialisation, but there should be a strong link with the user annotations in the source code. In the future, we envisage developing a query language with which the user can specify properties, with respect to state in the model/code (e.g. how a variable changes over time), but it remains to be seen how flexible it will be possible to make this.

8 Future Work and Conclusions

In this paper, we have described a lot of fairly general ideas for abstraction, without talking about any concrete system. Our ultimate aim is to be able to analyse real-world communications protocols, written in C, but this is too big a jump just yet, given the complexities of kernel-level network stacks, and their operating-system-specific implementations.

At the moment, we are developing a framework based around the network simulator `ns-2` [16]. Transport-level protocols are written in C++ as ‘agents’, which have a well-defined interface. For example, they must provide methods for receiving a packet, and responding to a timer expiring, among others. Our framework consists of two parts:

1. Analyse user annotations, and convert the C++ agent into instrumented C code. This is done in Java.

2. Parse the instrumented code using the C Intermediate Language (CIL) [18], and perform the sequence of abstraction steps, as described in this paper. This is done in OCaml. The output of this tool will be a PEPA model, which we will then be able to refine, specialise and analyse.

One advantage of using simulator code, is that we will be able to validate our model with results from the simulator. Furthermore, when moving to real implementations, only the front-end parsing will need to change. The back-end abstraction tool will remain the same.

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. There are certainly many more challenges to be faced, but we have taken the first few steps, and look forward to continuing along this path.

References

[1] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers, 2006. Not yet published. To appear in EuroSys’06. See `http://www.foment.net/byron/papers/eurosys.pdf`.

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

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

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

[5] Viktoria Firus, Steffen Becker, and Jens 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.

[6] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nielson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of PLDI 2002*, June 2002.

[7] Svend Frolund and Jari Koistinen. QML: A language for quality of service specification. Technical Report HPL-98-10, Hewlett Packard Software Technology Laboratory, 1998.

[8] Stephen Gilmore, Valentin Haenel, Leïla Kloul, and Monika Maidl. Choreographing security and performance analysis for web services. In Mario Bravettit and Gianluigi Zavattaro, editors, *Proceedings of the 2nd International Workshop on Web Services and Formal Methods*, LNCS, Versailles, France, September 2005. Springer-Verlang.

[9] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire 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.

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

[11] Jane Hillston. Fluid flow approximation of PEPA models. In *Proceedings of the Second International Conference on the Quantitative Evaluation of Systems*, pages 33–43, Torino, Italy, September 2005. IEEE Computer Society Press.

[12] Jane Hillston and Joanna Tomasik-Krawczyk. Amalgamation of transition sequences in the PEPA formalism. In *ICALP Satellite Workshops*, pages 523–534, 2000.

[13] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[14] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM: Probabilistic symbolic model checker. In P. Kemper, editor, *Proceedings of Tools Session of Aachen 2001 International Multiconference on Measurement, Modelling and Evaluation of Computer-Communication Systems*, pages 7–12, 2001.

[15] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioural Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

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

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

[18] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of Conference on Compiler Construction*, pages 213–228, 2002.

[19] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the Principles of Programming Languages*, 2002.

[20] Jason R. C. Patterson. Accurate static branch prediction by value range propagation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 67–78, 1995.

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