Activity-Guided Abstraction of PEPA Models

Michael J. A. Smith*

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

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

Abstract

Stochastic process algebras such as PEPA allow complex stochastic models to be described in a compositional way, but this leads to state space explosion problems. To analyse such models, aggregation is often performed at the state level, resulting in an abstract model that safely approximates the original with respect to certain properties. Abstract Markov chains and stochastic bounds can be used to safely approximate transient and steady state properties respectively.

A vital part of any abstraction is to specify which states to aggregate. In this paper, we look at how aggregation of PEPA activities can be used to guide aggregation of states in the underlying Markov chain.

1 Introduction

Practical performance modelling often deals with models that are too large to analyse directly. This is particularly the case when we use compositional formalisms such as the Performance Evaluation Process Algebra (PEPA) [2], for which the size of the underlying Markov chain can grow exponentially in the number of components. In these situations, we need to abstract the model in such a way that it becomes small enough to analyse, ensuring that we preserve any properties of interest.

This preservation usually cannot be exact, and so we introduce the notion of a *safe approximation* – if the abstract model satisfies or violates the property then so does the original model, but there is a possibility that we *cannot determine* either of these cases for the abstract model. As an example, since we can determine the probability distribution over traces in a CTMC, we can test whether it satisfies a transient property (i.e. a collection of traces) with at least a certain probability. For example, that there is at least 99% chance that the system will be operational within 100ms of being switched on. In the abstract model, we may have uncertainty about the trace probabilities, and instead have an interval $[p_1, p_2]$ that contains the actual probability. Hence when verifying the same property, we may determine that the chance of being operational within 100ms is $[0.9, 1.0]$, which means that we *do not know* that the property holds with at least a 99% probability, but we *do know* that it holds with at least a 90% probability.

A common way of reducing the size of a Markov chain is to *aggregate* certain states that have similar behaviour. Such an aggregation can be described by an abstraction function $\alpha : S \to S^\sharp$, where S is the state space of the original Markov chain, and $S^\sharp$ that of the abstract system. In general, the abstract system will not be Markovian, but if it is, the Markov chain is said to be

*This work was funded by a Microsoft Research European Scholarship

ordinarily lumpable [4]. This property means that the transitions out of each abstract state are independent of which concrete state we are in. More formally, for a CTMC defined in terms of a rate function $r : S \to \mathbb{R}^+$, describing the exit rate of each state, and a probability transition matrix $\boldsymbol{P} : S \times S \to [0, 1]$, then it is ordinarily lumpable with respect to α if for all states $s, s' \in S$ such that $\alpha(s) = \alpha(s')$, then for all $s^\sharp \in S^\sharp$:

$$\sum_{\{t|\alpha(t)=s^\sharp\}} r(s)\boldsymbol{P}(s,t) = \sum_{\{t|\alpha(t)=s^\sharp\}} r(s')\boldsymbol{P}(s',t)$$

If this condition holds, then the steady state solution of the aggregate CTMC is equivalent to solving the original, and then aggregating it.

Most of the time, however, we want to abstract CTMCs that are not lumpable, and so we need an abstraction. The following are two ways of doing this, depending on whether we are interested in transient or steady state properties of the chain:

1. Determine the maximum and minimum possible transition rates between the abstract states. We then use a modified model-checking algorithm to verify properties over the set of Markov chains we defined. This is the approach of *abstract Markov chains* [1, 3].

2. Modify the original CTMC by altering the rates so that the abstraction becomes lumpable. In order for this to be useful, we ensure that the modification yields an upper (or lower) bound to the property that we are interested in. This is the approach of *stochastic bounds* [6].

Since this is not the primary focus of this paper, we will not elaborate any further on the details of the above, except to comment that they can both be applied compositionally to PEPA, with slight modification [5]. Our aim here is to investigate ways in which we can choose the abstraction function α. Rather than describing the sets of states that we want to aggregate, it often makes more sense to do so in terms of *activities*, and in doing so take advantage of the structure of the model. The purpose of this paper is to motivate this further, and present an approach to *activity-guided* abstraction.

2 Activity-Guided Abstraction

In a structured formalism such as PEPA, it is natural to characterise the states by what they can do, since it makes more sense to aggregate *similar* states than ones with very different behaviour. In particular, we are keen to aggregate compositionally, looking at each sequential component separately. Not only is this more natural than working with the state space of the entire system, but it allows us to exploit the structure of the model more readily.

Consider a sequential component with state space S, and activities with action types in Act. If $A \subseteq Act$ is a set of action types, then we define $\mathcal{S}(A) \subseteq S$ to be the set of states in the component that can perform activities of every action type in A:

$$\mathcal{S}(A) = \{s \in S \mid \forall \alpha \in A.\ \exists s'.\ s \xrightarrow{\alpha} s'\}$$

For example, $\mathcal{S}(\{\alpha, \beta\})$ is the set of states that can perform both activities of type α and β. For a state $s \in \mathcal{S}(A)$, we say that s has a *signature* A.

In itself, this form of state identification has limited use, since it requires all concerned action

types to be possible from every state in the aggregation. Consider the following simple example:

$$\begin{array}{lcl} Server & = & (request_A, \top).(process, r_A).(reply_A, r_{net}).Server \\ & + & (request_B, \top).(process, r_B).(reply_B, r_{net}).Server \\ & + & (request_C, \top).(process, r_C).(reply_C, r_{net}).Server \\ Client_A & = & (request_A, r_{net}).(reply_A, \top).Client_B \\ Client_B & = & (request_B, r_{net}).(reply_B, \top).Client_C \\ Client_C & = & (request_C, r_{net}).(reply_C, \top).Client_A \\ System & = & Server \underset{\{request_{A,B,C}, reply_{A,B,C}\}}{\bowtie} Client_A \end{array}$$

Here, we have a server that can respond to three different types of request from the client, each of which has a different rate of processing. A sensible abstraction of the model would be to combine the different request types into one, hence throwing away any detailed information. If we take $A = \{request_A, request_B, request_C\}$, then $\mathcal{S}(A)$ is useless for abstracting the client, since each state can perform only one of the action types.

Because of this, we define an operator $\oplus$ that can be applied to sets of activities. Intuitively, $A_1 \oplus A_2$ means that a state has a signature of either A_1 or A_2. Semantically:

$$\mathcal{S}(A_1 \oplus A_2) = \mathcal{S}(A_1) \cup \mathcal{S}(A_2)$$

If we now take $A = \{request_A\} \oplus \{request_B\} \oplus \{request_C\}$, then for the client we have $\mathcal{S}(A) = \{Client_A, Client_B, Client_C\}$ as required. We can do the same for the *process* and *reply* activities to complete our abstraction. It is natural to then question what would happen if the processing on the server were more complicated, and we will return to this later.

In the case of this simple example, we can aggregate both the states and activities without encountering any problems. Aggregation of activities means that we treat all the activities as being of the same type. Hence when we construct an abstract Markov chain, for example, we have a single interval of rates for all the activities. To see how this improves the abstraction, consider the aggregated client state without activity abstraction – we would have to say that $request_A$ occurs at some rate in the interval $[0, r_{net}]$, since only one of the states can perform a $request_A$ action. If, however, we aggregate the activities, then the rate of performing a *request* action is now $[r_{net}, r_{net}]$.

We have to be more careful than this in general, however. Since we can aggregate states that individually cannot perform every action, we will introduce the possibility of deadlock when we synchronise with other components that have been similarly aggregated. This is because, in the abstraction, we are unable to know which of the concrete states we were originally in. Hence a safe abstraction will consider all possible combinations of states, even though some may not have been possible originally. To avoid this, when we aggregate with respect to $A = A_1 \oplus \ldots \oplus A_n$, we require that the following condition holds of at least one of the synchronising components:

$$\mathcal{S}(A_1 \oplus \ldots \oplus A_n) = \mathcal{S}(A_1 \cup \ldots \cup A_n)$$

This ensures that one component has states in which all the action types in A can be performed, which is illustrated in Figure 1.

Now that we have introduced this constraint, we see immediately that it is broken by our abstraction of the *reply* activities. This is because the aggregate state 'forgets' about the inherent coupling in the original model – it does not know whether the reply sent by the server is what the client is expecting. To capture the notion of coupling, we extend our description of activities with a further operator, '$\odot$'. Intuitively, $A_1 \odot A_2$ means that a state must have signature A_2, and all possible predecessor states must have signature A_1. More formally:

$$\mathcal{S}(A_1 \odot A_2) = \{s' \in S \mid \forall s.\ s \to s'.\ \forall \alpha \in A_1.\ \forall \beta \in A_2.\ \exists s''.\ s \xrightarrow{\alpha} s' \xrightarrow{\beta} s''\}$$

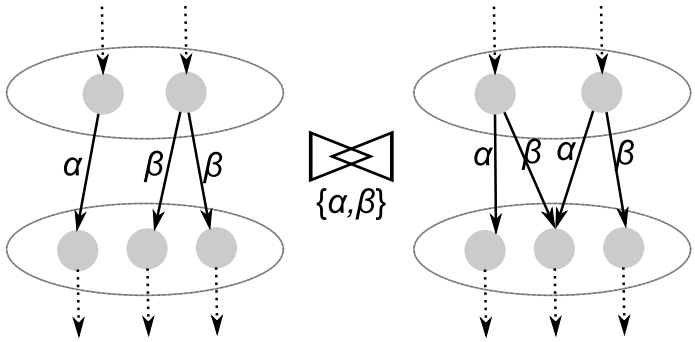

Figure 1: Condition for abstraction over $\{\alpha\} \oplus \{\beta\}$

The synchronisation condition holds for components synchronising over $(A_1 \odot A_1') \oplus (A_2 \odot A_2')$ if it holds for the same components synchronising over $A_1 \odot A_2$. In other words, once the two components are coupled, we can be sure that their signatures will match in future.

Returning to our example, we can now describe the abstract action types of the server as follows:

$$\begin{array}{lcl} A_1 & = & \{request_A\} \oplus \{request_B\} \oplus \{request_C\} \\ A_2 & = & (\{request_A\} \odot \{process\}) \oplus (\{request_B\} \odot \{process\}) \oplus (\{request_C\} \odot \{process\}) \\ A_3 & = & (\{request_A\} \odot \{process\} \odot \{reply_A\}) \oplus (\{request_B\} \odot \{process\} \odot \{reply_B\}) \\ & & \oplus\ (\{request_C\} \odot \{process\} \odot \{reply_C\}) \end{array}$$

And the abstract action types of the client are:

$$\begin{array}{lcl} A_1 & = & \{request_A\} \oplus \{request_B\} \oplus \{request_C\} \\ A_4 & = & (\{request_A\} \odot \{reply_A\}) \oplus (\{request_B\} \odot \{reply_B\}) \oplus (\{request_C\} \odot \{reply_C\}) \end{array}$$

Note that we allow two sequenced abstract action types to synchronise if they do not precisely match. We define the *interface* of such an abstract action type to be the list of signatures that contain action types over which we cooperate. If $\mathcal{L}$ is the set of such cooperating action types, then:

$$\begin{array}{lcl} \mathcal{I}(\{\alpha_1, \ldots, \alpha_n\}) & = & \{\alpha_1, \ldots, \alpha_n\} \\ \mathcal{I}(A_1 \odot A_2) & = & A_1 \odot \mathcal{I}(A_2) \text{ if } \mathcal{L} \cap A_1 \neq \emptyset \\ \mathcal{I}(A_1 \odot A_2) & = & \mathcal{I}(A_2) \text{ if } \mathcal{L} \cap A_1 = \emptyset \\ \mathcal{I}(A_1 \oplus A_2) & = & \mathcal{I}(A_1) \oplus \mathcal{I}(A_2) \end{array}$$

Two abstract action types are allowed to cooperate if and only if they have the same interface. Hence A_3 can cooperate with A_4, because $\mathcal{I}(A_3) = \mathcal{I}(A_4)$.

We noted previously that the processing of requests in our example is very simple, but that it could be more complicated. In particular, the server could potentially have a number of internal stages to its processing. If all of these internal stages have the same signature A, such that no action types in A are cooperated over, then we can use the *iteration* operator A^+. This means that one or more states with signature A are passed through in sequence, and we can define it in terms of the '$\oplus$' and '$\odot$' operators:

$$A^+ = A \oplus (A \odot A) \oplus (A \odot A \odot A) \ldots$$

For example, if our server had multiple processing stages for each request type, we could rewrite the abstract state A_2 as follows:

$$A_2 \quad = \quad (\{request_A\} \odot \{process\}^+) \oplus (\{request_B\} \odot \{process\}^+) \oplus (\{request_C\} \odot \{process\}^+)$$

This would combine all the processing steps into a single state in the abstract model. In general, this is not a good thing to do – it is certainly safe, but can yield very poor abstractions. For example, in an abstract Markov chain the probability of exiting this state would be in the interval [0,1], since the intermediate processing states have zero chance of exiting, whereas the final processing state is certain to exit. We can do much better when so-called *insensitivity* [7] conditions hold, but this is beyond the scope of this paper.

To summarise, we have introduced the following regular expression-like operators for describing sets of states that we wish to aggregate:

- $\{\alpha_1 \dots \alpha_n\}$ – the state can perform all action types $\alpha_1 \dots \alpha_n$.
- $A_1 \oplus A_2$ – the state has a signature of either A_1 or A_2.
- $A_1 \odot A_2$ – the state has a signature of A_2, and all predecessor states have a signature of A_1.
- $A^+ = A \oplus (A \odot A) \oplus (A \odot A \odot A) \dots$

3 Conclusions

In this paper, we described an approach to specifying aggregations of states in a PEPA model using activities. This work is at quite an early stage, and it was our aim to give an outline of the ideas involved, rather than to provide a formal framework. Some of the details not discussed here are in relation to the particular abstraction methods used – for example abstract Markov chains or stochastic bounds – and are fairly straightforward to work out. There are, however, many challenging issues that remain to be fully explored, such as the interaction between insensitivity and abstraction, and the development of a simpler formalism for direct use by modellers wishing to abstract. The use of activities is an important step to providing a more intuitive way of specifying abstractions, but we do not yet know whether the current constructions are sufficient, or whether we need to extend our specification further.

References

[1] H. Fecher, M. Leucker, and V. Wolf. Don't know in probabilistic systems. In *Proceedings of SPIN'06*, number 3925 in Lecture Notes in Computer Science, pages 71–88, 2006.

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

[3] J-P. Katoen, D. Klink, M. Leucker, and V. Wolf. Three-valued abstraction for continuous-time Markov chains. In W. Damm and H. Herrmanns, editors, *Proceedings of 19th International Conference on Computer-Aided Verification (CAV'07)*, number 4590 in Lecture Notes in Computer Science, pages 316–329. Springer-Verlag, 2007.

[4] J.G. Kemeny and J.L. Snell. *Finite Markov Chains*. Springer-Verlag, 1976.

[5] M.J.A. Smith. Stochastic bounding of PEPA models. In *Proceedings of Process Algebra and Stochastically Timed Activities (PASTA)*, 2007.

[6] D. Stoyan. *Comparison Methods for Queues and Other Stochastic Models*. Wiley & Sons, New York, NY, USA, 1983.

[7] P. Whittle. Partial balance and insensitivity. *Journal of Applied Probability*, 22(1):168–176, 1985.