A Tool for Abstraction and Model Checking 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. The stochastic process algebra PEPA is a widely used language for performance modelling, and a large part of its success is due to the rich tool support that is available. As a compositional Markovian formalism, however, it suffers from the state space explosion problem, where even small models can lead to very large Markov chains. One way of analysing such models is to use abstraction — constructing a smaller model that bounds the properties of the original.
We present a new tool for abstracting and model checking PEPA models, which is an extension of the Eclipse plug-in for PEPA. This extends the current tool with two additional views. The abstraction view is a graphical interface for labelling and aggregating states of PEPA components. The model checking view allows CSL properties to be specified and checked. Internally, we use the techniques of abstract Markov chains and stochastic bounds to analyse transient and steady state properties respectively. We have extended both techniques so that they can be applied compositionally to PEPA.

1 Introduction

In the distributed world of today, performance has a vital role in the construction of robust, reliable and scalable computer systems. A powerful technique for reasoning about the performance of these systems is to use mathematical modelling — in particular, working with high-level compositional languages, such as stochastic process algebras. One such widely used language is the Performance Evaluation Process Algebra (PEPA) [9], and a significant part of its success can be attributed to the many tools that are available [4, 8, 14, 20].

The PEPA language has two primary semantics — a Markovian semantics, which maps a model to a continuous time Markov chain (CTMC) [9], and an ordinary differential equation (ODE) semantics [10]. Both semantics, as well as stochastic simulation, are implemented in the PEPA plug-in tool [20], which is built on the Eclipse platform [1]. This provides an interface for editing PEPA models, and performing both steady state and time series analysis. An Integrated

* This work was funded by a Microsoft Research European Scholarship

Development Environment (IDE) such as Eclipse has the advantage of providing a standardised interface, with which users are already familiar, and removing much of the burden of user interface development from the tool developer.

The main problem faced by compositional formalisms like PEPA, is that even a relatively small model can lead to a CTMC that is much too large to analyse. This state space explosion problem can be avoided if we are only interested in the average behaviour of the system — namely, the average number of components in a given state at a particular time — in which case we can use the ODE semantics of PEPA. However, if we want to reason about *all* possible runs of the model, and not just the average case, we need to look at its Markovian semantics. And since the problem of state space explosion is unavoidable, we have no choice but to look for ways of dealing with it.

The basic principle for analysing a large model seems deceptively simple — we just need to make it smaller! This process of *abstraction* is more difficult than it sounds, because we need to preserve information about the properties we are interested in. There a few different ways for us to do this, but the approach we take in this paper is to *aggregate* states with similar behaviour. Since, most of the time, this throws away some of the information in the original model, we look at obtaining *bounds* on properties of the model. For example, if a system has a probability 0.001 that it fails within the first ten minutes of operation, the abstracted model might tell us that the probability lies in the interval $[0, 0.002]$. The important thing is that the bound is *accurate*, in that the actual probability always lies within the interval we obtain.

Whilst some abstractions can give very tight bounds on the probability of satisfying a property, others can give less useful information. Indeed, in the worst case, we could "discover" that the probability lies in the interval $[0, 1]$. Hence the key to obtaining useful information is the choice of abstraction — in other words, how do we select which states to combine? As there is currently no way, in general, to automatically decide this, it is important that we can easily try different abstractions. This leads to the topic of this paper — providing an interface for quickly and easily specifying abstractions of PEPA models, and for checking properties of the abstracted models.

There are a number of techniques for producing bounded abstractions of a CTMC, and we make use of two of them — *abstract Markov chains* [6, 12], and *stochastic bounds* [7, 19]. The former can be used to bound transient properties, and the latter for various monotone properties such as the steady state distribution. Both of these techniques were originally specified and used at the level of Markov chains, but we have extended them so that they can be applied *compositionally* to PEPA models [18]. This means that we can construct bounds for models whose underlying CTMC is too large to represent (let alone analyse), since we maintain its compositional structure when building the abstract model. It also means that the modeller can identify which states to aggregate in a more natural way, using the same component-level view as in the original model.

In this paper, we present a new extension to the Eclipse PEPA plug-in, which provides a graphical interface for abstracting PEPA models, and a model checker for properties in the Continuous Stochastic Logic (CSL) [2,3]. The tool can be downloaded from `http://www.dcs.ed.ac.uk/pepa/tools/plugin`, and extends the PEPA plug-in with the following two views:

1. The **Abstraction View** is a graphical interface that shows the state space of each sequential component in a PEPA model. It provides a facility for labelling states (so that they can be referred to in CSL properties), and for specifying which states to aggregate.
2. The **Model Checking View** is an interface for constructing, editing, and model checking CSL properties. The property editor provides a simple way to construct CSL formulae, by referencing states that are labelled in the abstraction view. Only valid CSL formulae can be entered.

In this paper, we will begin by describing the PEPA language, along with a small example, in Section 2. We will then describe the abstraction and model checking views in Sections 3 and 4 respectively. We give a brief description of the theory behind the approach, and some implementation details in Section 5. Finally, we consider a larger example in Section 6, to illustrate the capabilities of the tool, before concluding in Section 7.

2 The PEPA Language

The Performance Evaluation Process Algebra (PEPA) [9] is a compositional formalism with Markovian semantics. 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 (a, r), where $a \in \mathcal{A}$ is its action type, and $r \in \mathbb{R}_{\geq 0} \cup \{\top\}$ is the rate of the activity. This rate parameterises an exponential distribution, and if unspecified (denoted $\top$), the activity is said to be *passive*. In this case, another component must actively drive the rate of the action. PEPA terms have the following syntax:

$$C := (a,r).C \mid C_1 + C_2 \mid C_1 \underset{L}{\bowtie} C_2 \mid C/L \mid A$$

We briefly describe these combinators as follows:

– *Prefix* ($(a,r).C$): the component can carry out an activity of type a at rate r to become the component C.
– *Choice* ($C_1 + C_2$): the system may behave either as component C_1 or C_2. The current activities of both components are enabled, and the first to complete determines which component proceeds. The other component is discarded.
– *Cooperation* ($C_1 \underset{L}{\bowtie} C_2$): the components C_1 and C_2 synchronise over the cooperation set L. For activities whose action 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* (C/L): the component behaves as C, except that activities whose types are in L are hidden, and appear externally as the unknown type τ.
– *Constant* ($A \overset{def}{=} C$): component C has the name A.

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 continuous time Markov chain (CTMC).

$$
\begin{aligned}
P_{14} &= (reg_{14}, r).P_{14} + (move_{15}, m).P_{15} \\
P_{15} &= (reg_{15}, r).P_{15} + (move_{14}, m).P_{14} + (move_{16}, m).P_{16} \\
P_{16} &= (reg_{16}, r).P_{16} + (move_{15}, m).P_{15} \\
\\
S_{14} &= (reg_{14}, \top).(rep_{14}, s).S_{14} \\
S_{15} &= (reg_{15}, \top).(rep_{15}, s).S_{15} \\
S_{16} &= (reg_{16}, \top).(rep_{16}, s).S_{16} \\
\\
DB_{14} &= (rep_{14}, \top).DB_{14} + (rep_{15}, \top).DB_{15} + (rep_{16}, \top).DB_{16} \\
DB_{15} &= (rep_{14}, \top).DB_{14} + (rep_{15}, \top).DB_{15} + (rep_{16}, \top).DB_{16} \\
DB_{16} &= (rep_{14}, \top).DB_{14} + (rep_{15}, \top).DB_{15} + (rep_{16}, \top).DB_{16}
\end{aligned}
$$

$$
P_{14} \underset{\{reg_{14}, reg_{15}, reg_{16}\}}{\bowtie} (S_{14} \parallel S_{15} \parallel S_{16}) \underset{\{rep_{14}, rep_{15}, rep_{16}\}}{\bowtie} DB_{14}
$$

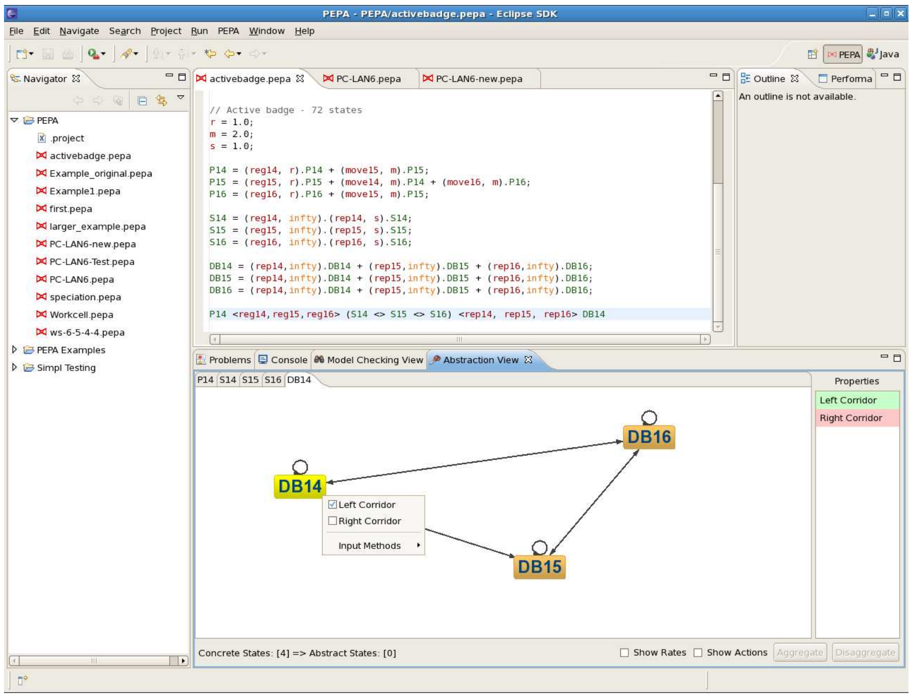
Fig. 1. A PEPA model of an active badge system

An example PEPA model with five components is shown in Figure 1. This is a model of an active badge sensor system, which was first presented in [5]. In the model, a person (component P) moves between three corridors, labelled 14, 15 and 16, which are arranged linearly. Each corridor i has a sensor S_i, which listens for a registration signal reg_i from the person, and informs the database DB. The state of the database effectively records where the person was last seen. The model has three rate parameters — r is the rate at which the badge sends a signal to the sensors, m is the rate of moving between corridors, and s is the rate at which the sensor updates the database.

There are a number of interesting questions we might ask of the model. For example, what proportion of the time does the person spend in each of the corridors? Or, what is the probability that the database can move directly from state DB_{14} to state DB_{16}, missing the fact that the person must have passed through corridor 15? Since this particular example has only 72 states, we can easily analyse it directly without need for abstraction. We will, however, use it as a running example while we describe the features of the tool in the following two sections. In Section 6 we will look at a larger example, that better illustrates the power of abstraction.

3 Specifying State-Based Abstractions

In order to construct and check CSL properties of a PEPA model, we need some way of referring to states of the model. One way of doing this would be use the names of the sequential component states in the model, but this could lead to very long and cumbersome names — especially if we refer to a large set of states. Ideally, we would prefer to use a single, meaningful name. Our solution is to provide a graphical interface for labelling sets of states.



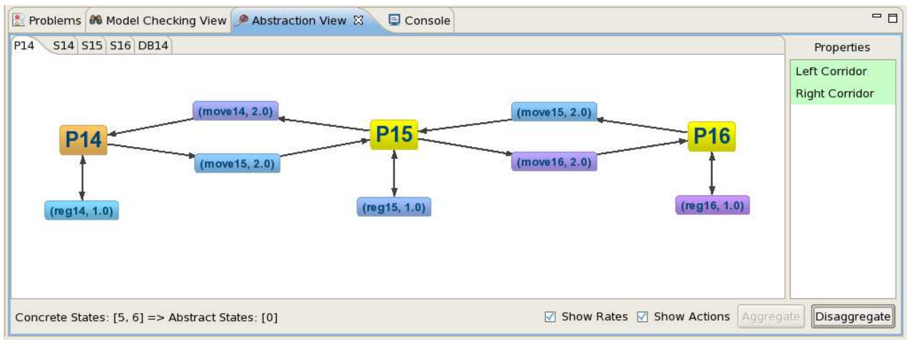
```
// Active badge - 72 states
r = 1.0;
m = 2.0;
s = 1.0;

P14 = (reg14, r).P14 + (move15, m).P15;
P15 = (reg15, r).P15 + (move14, m).P14 + (move16, m).P16;
P16 = (reg16, r).P16 + (move15, m).P15;

S14 = (reg14, infty).(rep14, s).S14;
S15 = (reg15, infty).(rep15, s).S15;
S16 = (reg16, infty).(rep16, s).S16;

DB14 = (rep14,infty).DB14 + (rep15,infty).DB15 + (rep16,infty).DB16;
DB15 = (rep14,infty).DB14 + (rep15,infty).DB15 + (rep16,infty).DB16;
DB16 = (rep14,infty).DB14 + (rep15,infty).DB15 + (rep16,infty).DB16;

P14 <reg14,reg15,reg16> (S14 <> S15 <> S16) <rep14, rep15, rep16> DB14
```

Fig. 2. The PEPA Eclipse Plug-In

An overall view of the PEPA plug-in is shown in Figure 2. Here, the active badge model of Figure 1 is open in the editor, and the abstraction view is in use. The majority of the abstraction view is taken up by a graphical representation of the sequential components in the system equation of the model. In this case, there are five components, and each corresponds to a tab in the view. Currently on display is the database component DB.

On the right of the abstraction view is a table showing the atomic properties for the model. Right clicking on this table brings up a menu, from which we can define a new property, or rename or delete an existing one. When we create a new property, the currently selected states in the graph will initially satisfy it, and the other states will not. We can change which properties a state satisfies by right clicking on the state — this allows us to select or deselect the atomic properties, as illustrated in the figure.

An additional feature of the property table is that clicking on a property will highlight all the states that satisfy it. Clicking on a state in the graph will shade the properties that it satisfies in green, and those it does not in red. This allows us to quickly see which states satisfy which properties. It is important to remember that all atomic properties are defined *compositionally*. A state in the system satisfies an atomic property if and only if its state in each component does. In Figure 2, the property "Left Corridor" is satisfied by DB_{14}, but not by DB_{15} and DB_{16}. Since we do not constrain the property for any of the other components, it is satisfied by all states of the system that have the database in state DB_{14}. An example would be the state $P_{14} \parallel S_{14} \parallel S_{15} \parallel S_{16} \parallel DB_{14}$.



Concrete States: [5, 6] => Abstract States: [0]

Fig. 3. The abstraction interface

The second function of the abstraction view is, as its name suggests, to specify an *abstraction* — namely, which states to aggregate. Figure 3 shows a close-up of the abstraction view, this time for the component P. In this case, we have selected to show both the actions and the rates on the transitions, but since this leads to a more cluttered graph these options are not selected by default.

Aggregating states in a sequential component is simply a matter of selecting the states, and clicking the *Aggregate* button. They can be separated again by clicking *Disaggregate*. Once a set of states have been aggregated, we can only select them as a group — clicking on any one of the states will select them all. Note that the aggregation of states is independent of both the labelling of atomic properties[1] and the definition of CSL properties in the model checking view. This means that we can quickly try out different abstractions — the only thing we need to do is to run the model checker each time.

4 Model Checking Abstract PEPA Models

To describe properties of a PEPA model, we need a logic for expressing them. The most widely used logic for model checking CTMCs is the Continuous Stochastic

[1] If we aggregate a set of states when only some of them satisfy a property, the abstract state will have a truth value of '?' (i.e. it 'maybe' satisfies the property).

Logic (CSL) [2,3]. CSL is a branching-time temporal logic, which allows us to talk about the *probability* of a state satisfying some temporal property, and the *time interval* in which a property must hold.

Formulae in CSL consist of *state formulae* Φ, and *path formulae* φ. The former are properties of individual states in the Markov chain — for example, that the steady state probability is greater than a certain value. The latter are properties that hold of paths (sequences of states) through the chain — for example, that a state property holds until some condition is met.

State formulae Φ are defined as follows, for $\trianglelefteq \in \{\leq, \geq\}$, $p \in [0,1]$ and atomic propositions a:

$$\Phi ::= \texttt{tt} \mid a \mid \Phi \wedge \Phi \mid \neg\Phi \mid \mathcal{S}_{\trianglelefteq p}(\Phi) \mid \mathcal{P}_{\trianglelefteq p}(\varphi)$$

In addition to atomic propositions and the standard logical connectives, there are two state formulae of interest:

- A *steady state measure* $\mathcal{S}_{\trianglelefteq p}(\Phi)$ is satisfied if the steady state probability of being in the set of states satisfying Φ is $\trianglelefteq p$.
- A *path measure* $\mathcal{P}_{\trianglelefteq_p}(\varphi)$ is satisfied of a state s if the integral of the probability measures of all the paths from s satisfying φ is $\trianglelefteq p$.

A useful extension to the basic CSL syntax, used by the PRISM [14] model checker and by us, allows us to test the value of a steady state or path measure:

$$\Phi_T = \mathcal{S}_{=?}(\Phi) \mid \mathcal{P}_{=?}(\varphi)$$

These are not state formulae in themselves, since they do not evaluate to a truth value, but to the *probability* of the property holding of a state. This does not affect the expressivity of CSL, but is convenient for users of a model checker.

Path formulae φ have the following syntax, where I is a non-empty interval over $\mathbb{R}_{\geq 0}$:

$$\varphi ::= X^I \Phi \mid \Phi\ \mathcal{U}^I \Phi$$

These two operators have the following semantics:

- The *next* operator, $X^I\Phi$, holds of a path if it leads to a state satisfying Φ in one transition, within the time interval I.
- The *until* operator, $\Phi_1\ \mathcal{U}^I \Phi_2$, holds of a path if it reaches a state satisfying Φ_2 within the time interval I, and until then only passes through states that satisfy Φ_1.

Classically, for a CTMC, a CSL formula will evaluate to either true or false, or a probability in the case of the test operators Φ_T. In our case, however, we want to model check *abstract* models, hence the test operators will evaluate to a *probability interval.* This describes the best and worst case probability of satisfying the formula, based on the information available in the abstraction. Since this means that we might not know whether or not the model satisfies a property, we need to use a three-valued semantics of CSL [12], with truth values of $\top$, $\bot$ and ? (true, false, and maybe).

To illustrate the properties we can express, consider the following formula, for the active badge model from Figure 1. Let us assume that we used the abstraction view to define the atomic properties *Left Corridor* and *Right Corridor*, meaning that the database is in states DB_{14} and DB_{16} respectively:

$$\mathcal{P}_{=?}(Left\ \ Corridor\ \mathcal{U}\ Right\ \ Corridor)$$

This asks the question, "what is the probability that the database will continue to think that the person is in the leftmost corridor, until it becomes aware that the person is in the rightmost corridor?" We can construct this formula using the CSL editor, as illustrated in Figure 4.

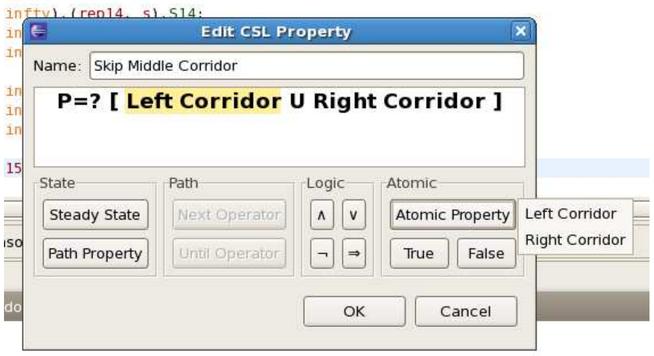
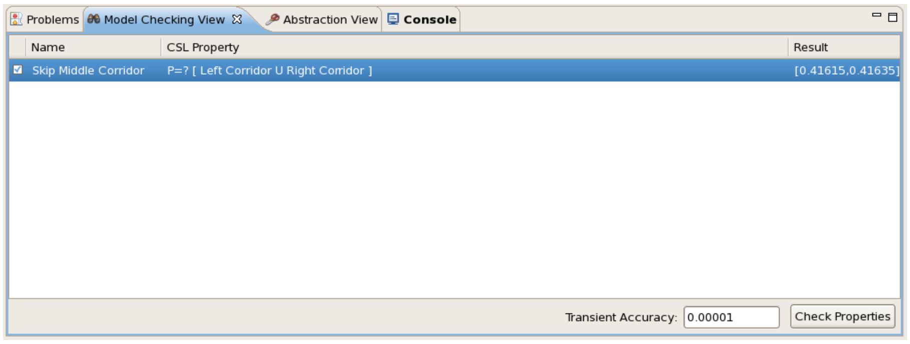


Fig. 4. The CSL property editor

The aim of the CSL editor is to make it as easy as possible to construct a CSL formula. In particular, it ensures that we can construct only valid formulae. The buttons on the interface correspond to the various CSL operators and logic connectives, and are enabled by clicking on the part of the formula we want to edit. Hence we cannot enter a path formula where a state formula is required, or vice versa, and the test operators Φ_T can only be used at the top level of a formula. The path and state operator buttons produce a pop-up menu with the available choices — for example, timed versus untimed until operators.

The most useful feature of the CSL editor is that it presents us with a list of the atomic formulae that we defined in the abstraction view. Hence, we can easily refer to sets of states in the model, using the labels we created. Because the internal data structures are shared, if we change the name of a property in the abstraction view, it will automatically be updated in the CSL formulae that use it. Similarly, a property cannot be deleted from the abstraction view while it is being used in a formula.

Figure 5 shows the model checking view, from which the CSL editor can be opened. The main component of the view is a table of all the properties that are defined for the model. When the *Check Properties* button is pressed, all



	Name	CSL Property	Result
✓	Skip Middle Corridor	P=? [Left Corridor U Right Corridor]	[0.41615,0.41635]

Transient Accuracy: 0.00001

Fig. 5. The model checking interface

selected properties are model checked, and the results are displayed next to each property. In this case, we have not abstracted the model, so the result is very precise — a probability interval of $[0.41615, 0.41635]$[2]. The only error here is due to the termination condition of the model checker itself, and can be improved by modifying the *Transient Accuracy* field.

If we require more detailed information about the progress of the model checker, a log is available in the console view. For readability, we reproduce the output of the console, for model checking the above property, in Figure 6. In the next section, we will briefly discuss the implementation architecture in more detail, but first let us consider some results obtained by *abstracting* the active badge model — this is, after all, the purpose of our tool.

```
13:16:04 [badge.pepa] Model added.
13:16:04 [badge.pepa] Model parsed.
13:16:08 [badge.pepa] Kronecker state space derived. Elapsed time: 5 ms.
13:24:36 [badge.pepa] <Model Checker> Generating abstract CTMC...
13:24:36 [badge.pepa] <Model Checker> Optimising uniformisation constant to 7.0...
13:24:36 [badge.pepa] <Model Checker> Generated abstract CTMC with 72 states.
13:24:36 [badge.pepa] Property "P=? [ Left Corridor U Right Corridor ]" was checked in 28 ms.
```

Fig. 6. Console output from the model checker

Table 1 shows the result of model checking two transient properties of the active badge model under different abstractions. The first is the same untimed until property we have been considering up to now (where we shorten *Left Corridor* to *Left*, and *Right Corridor* to *Right*). The second is a *timed* until property, which states the same condition, but with the additional constraint that the database must enter state DB_{16} (the rightmost corridor) within one time unit. For each property, we investigate the effect of aggregating the states of each of the sensors, and then finally aggregating all three of them at the same time.

[2] We have validated our results for concrete models against PRISM.

CSL Property	**Aggregated States**	**State Space Size**	**Probability Interval**
$\mathcal{P}_{=?}(\mathit{Left}\ \mathcal{U}\ \mathit{Right})$	None	72	$[0.41615, 0.41635]$
	$\{S_{14}, rep_{14}.S_{14}\}$	36	$[0.41615, 0.41635]$
	$\{S_{15}, rep_{15}.S_{15}\}$	36	$[0.06246, 1.00000]$
	$\{S_{16}, rep_{16}.S_{16}\}$	36	$[0.00000, 0.90004]$
	All of the above	9	$[0.00000, 1.00000]$
$\mathcal{P}_{=?}(\mathit{Left}\ \mathcal{U}^{[0,1]}\ \mathit{Right})$	None	72	$[0.03018, 0.03019]$
	$\{S_{14}, rep_{14}.S_{14}\}$	36	$[0.03018, 0.03019]$
	$\{S_{15}, rep_{15}.S_{15}\}$	36	$[0.01556, 0.03199]$
	$\{S_{16}, rep_{16}.S_{16}\}$	36	$[0.00000, 0.62023]$
	All of the above	9	$[0.00000, 0.63213]$

Table 1. Abstract model checking of the active badge model

The results clearly show how the choice of abstraction affects the precision of the bounds. For both properties, abstracting sensor S_{14} has no effect on the bounds — hence we can halve the size of the model without losing precision. The story for the other sensors is quite different, however. Aggregating S_{16} gives a poor bound in both cases, whereas in the case of S_{15} the bound is much worse for the first property than the second. Finally, aggregating all three sensors results in the largest reduction in the size of the model, but at the cost of limited information for the second property, and no information for the first.

We can intuitively see why aggregating S_{14} has no affect on the precision, since it cannot cause the database to move from its initial state. The other two sensors have the power to move the database to a state that satisfies the property (in the case of S_{16}), or violates the property (S_{15}), hence aggregating either of them will have a big effect on the precision. Having said this, it is not obvious to begin with that it is safe to aggregate S_{14}, and in larger models safe abstractions can be even harder to find.

The advantage of our tool is that it allows us to experiment with different abstractions of the model, without worrying about whether or not it is safe to do so. The results of the model checker are always accurate, in that the actual probability of satisfying the property lies within the interval we obtain. Furthermore, if an abstract model satisfies or violates a particular CSL property, we can be sure that the original model also does. In the worst case, we might obtain imprecise bounds, but if we reduce the size of the model sufficiently, there is very little cost involved.

5 Architecture of the Implementation

So far, we have looked at the user interface for abstracting and model checking PEPA models without comment on the implementation details. Whilst a detailed description of the theory is beyond the scope of this paper, it is useful to know something about the type of abstraction we use, to make use of the tool more effectively.[3]. A simplified view of the implementation architecture is shown in Figure 7. The dotted box contains our addition to the PEPA plug-in, and we show how it interacts with the parts of the existing tool that we take advantage of — namely, the PEPA editor and parser, and the Markov chain solvers.

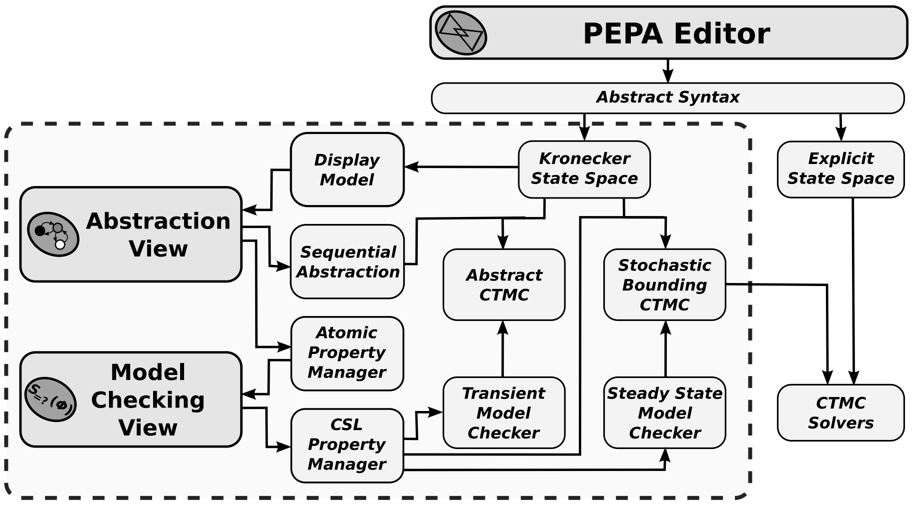


Fig. 7. The architecture of the abstraction and model checking engine for PEPA

The key to our approach is in using a *Kronecker* representation of the state space of a PEPA model. Rather than explicitly expanding a model, we maintain compositionality in our representation of its generator matrix. For a PEPA model consisting of n sequential components, $C_1, \ldots, C_n$, cooperating over all shared action types $a \in L$[4], we can write the generator matrix $\boldsymbol{Q}$ of model in the following form [11]:

$$\boldsymbol{Q} = \sum_{a \notin L} \bigoplus_{i=1}^{n} \boldsymbol{Q}_{i,a} + \sum_{a \in L} \min\{r_{i,a}\} \left(\bigotimes_{i=1}^{n} \boldsymbol{P}_{i,a} - \boldsymbol{I} \right)$$

where $\boldsymbol{P}_{i,a}$ is the probabilistic transition matrix of component C_i for action type a, and $r_{i,a}$ is a function describing the exit rate of activities of type a for each state in C_i. The operators $\oplus$ and $\otimes$ are the Kronecker sum and product respectively [15].

When we abstract a PEPA model, we do so compositionally on its Kronecker state space. It is only when we come to analyse the abstract model that we have to expand the Kronecker representation. The size of this representation is

[3] For a more thorough account, see [17].

[4] The tool internally and transparently transforms a PEPA model into this form, so there are no restrictions on the models we can analyse.

proportional to the sum of the number of states in each sequential component — as opposed to the product, in the worst case, for the expanded state space.

From the Kronecker state space, we generate a graph representation of the structure of each sequential component, which we call the *display model*. This is rendered by the abstraction view, which manages a *sequential abstraction* of each component, based on the states that the user is currently aggregating. It also stores the set of atomic properties for the model, which is shared with the model checking view. The model checking view in turn keeps track of a set of CSL properties for each model.

The heart of the tool is the abstraction engine. In general, if we aggregate states in a Markov chain, we end up with a non-Markovian model, since we are no longer memoryless. However, if all the states in each aggregate partition agree on the probabilities of moving to the other aggregate partitions — a condition called *ordinary lumpability* [13] — then we *do* still end up with a Markov chain. Unfortunately, this condition is rare in practice, and so we need to look for alternative techniques — in our case, we make use of two:

1. An *abstract Markov chain* [6,12] is the natural result of aggregating a Markov chain that is not lumpable. Since the states in an aggregate partition have different transition probabilities, we take the maximum and minimum probability of moving between two abstract states. This is similar to a Markov decision process [16], and indeed the model checking algorithms are similar [12]. Since an abstract Markov chain is not Markovian, it has no steady state solution — hence, we cannot use it to model check steady state properties. We can, however, model check all other CSL formulae bar the timed next operator[5].

   We can apply abstract Markov chains compositionally to PEPA models at a small loss of precision, by applying the abstraction before expanding the Kronecker representation. This results in an abstract CTMC that can be model checked in the usual way.
2. *Stochastic bounding* [19] of Markov chains is based on the principle of ordered probability distributions. For one Markov chain to bound the steady state distribution of another, it must ensure that its distribution is a bound at all times. For any Markov chain, given an ordering on its state space and a partitioning, we can algorithmically construct an upper (or lower) bound that is ordinarily lumpable with respect to that partitioning [7]. This can be applied compositionally to PEPA, so that the bound is constructed at the level of its Kronecker representation [18]. Since stochastic bounding produces another Markov chain, we can solve it in the usual way to compute its steady state distribution, which bounds that of the original model.

By combining these two distinct techniques, we are able to model check both transient and steady state properties of abstract PEPA models.

[5] The timed next operator is not preserved under uniformisation, which is used when we apply abstract Markov chains in a continuous time setting.

6 A Larger Example

Before we conclude this paper, we will examine a larger PEPA model. Figure 8 is a model of a round-robin server architecture, where the resources of a single server are shared between n computers. The server moves around each computer in turn — if there is a job waiting, it services it before moving onto the next computer. Jobs arrive at computer PC_i at rate λ_i, and the service rate of the server is μ. The server moves between computers at rate ω.

$$
\begin{aligned}
PC_i &= (arrive, \lambda_i).PC'_i + (walkon_{(i+1) \bmod n}, \top).PC_i \\
PC'_i &= (serve_i, \top).PC_i \\
Server_i &= (walkon_{(i+1) \bmod n}, \omega).Server_{(i+1) \bmod n} + (serve_i, \mu).Server'_i \\
Server'_i &= (walk_{(i+1) \bmod n}, \omega).Server_{(i+1) \bmod n}
\end{aligned}
$$

$$(PC_0 \parallel \ldots \parallel PC_{n-1}) \underset{\{walkon_0, \ldots, walkon_{n-1}, serve_0, \ldots, serve_{n-1}\}}{\bowtie} Server_0$$

Fig. 8. A PEPA model of a round-robin server architecture

Consider this model when $n = 6$, in which case the concrete PEPA model has 768 states. To avoid any symmetry in the model that could allow a more exact aggregation, we will assume that every computer has a different arrival rate, $\lambda_i = i + 1$. The results of model checking two distinct properties are shown in Table 2. The first property looks at the proportion of time spent in a $Server'$ state, where the server has completed a job, but has not yet moved to the next computer. The second property looks at the probability that the server will reach PC_2 within the first 0.1 time units (given that it starts with PC_0).

CSL Property	Aggregated States	State Space	Probability Interval
$\mathcal{S}_{=?}(Server')$	None	768	$[0.31184, 0.31184]$
	$\{Server'_{0\ldots5}\}$	7	$[0.00000, 0.33333]$
	$\{Server_{0\ldots5}\}$	7	$[0.00000, 0.75000]$
	$\{Server'_{2\ldots5}, Server_{3\ldots5}\}$	6	$[0.00000, 1.00000]$
$\mathcal{P}_{=?}(\top\ \mathcal{U}^{[0,0.1]}\ Server_2)$	None	768	$[0.53940, 0.53941]$
	$\{Server'_{0\ldots5}\}$	448	$[0.51954, 0.54567]$
	$\{Server_{0\ldots5}\}$	448	$[0.00000, 1.00000]$
	$\{Server'_{2\ldots5}, Server_{3\ldots5}\}$	384	$[0.53940, 0.53941]$

Table 2. Abstract model checking of the round-robin server model

Looking at the steady state property first, we see that aggregating all the $Server'$ states gives a good upper bound on the actual probability. Although the lower bound provides no information, this would be a useful result if we were interested in verifying that the server spends no more than a certain proportion of time in a $Server'$ state. This is especially true when we consider that the state space has been reduced by 99%. The other choices of aggregation yield poor results, however, which illustrates how the best abstraction depends entirely on the property we are analysing.

If we look at the second property, by comparison, we see that we achieve the best results when we abstract all the states following $Server_2$, but before $Server_0$. In this case, we can halve the state space without affecting the precision. In fact, since Table 2 only looks at aggregating states on the server, we can do even better. If we aggregate the computer states for $PC_2 \ldots PC_5$, we can reduce the state space to just 24 states (a 97% reduction in size) without affecting the precision. The reason we can achieve such good results here, is that after the server passes through the $Server_2$ state, the property must be satisfied — hence we can ignore all subsequent states. The abstraction view allows us to take advantage of this aggregation very quickly, without requiring any modifications to the model.

7 Conclusions

In this paper, we have described a new tool for abstracting and model checking PEPA models, which is an extension of the Eclipse PEPA plug-in. It provides a graphical interface for labelling and aggregating states of PEPA components, and for constructing and model checking CSL properties. The key advantage of the tool is that it allows modellers to quickly experiment with different ways of abstracting their models, and to take advantage of direct model checking facilities in the PEPA plug-in, without requiring external tools such as PRISM. This is not to say that our tool is a replacement for other, more established model checkers, but we feel that it is a useful addition to the artillery of the performance modeller.

Whilst a great deal has gone into the development of this tool, we are aware that there are still many areas in which its capabilities can be expanded upon and improved. For example, we would like to add support in the near future for PEPA's aggregation notation, which provides a shorthand for specifying a number of identical copies of the same component (e.g. *Client*[100]). We also intend to add support for exporting and importing of CSL properties, so that the plug-in can more easily be used in conjunction with PRISM and other tools.

Overall, whilst we have demonstrated the capabilities of our tool in this paper, our ultimate objective is to provide a practical and useful contribution to the model checking community. We would be delighted if you, the reader, could take the time to download and experiment with the new PEPA plug-in, and to provide us with any feedback, comments, or suggestions for improvement. It is easy to install, and full instructions are given at `http://www.dcs.ed.ac.uk/pepa/tools/plugin`. May many happy abstractions await you!

References

1. The Eclipse platform. `http://www.eclipse.org`.
2. A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Verifying continuous time Markov chains. In *Computer Aided Verification (CAV)*, number 1102 in Lecture Notes in Computer Science, pages 269–276. Springer-Verlag, 1996.
3. C. Baier, B.R. Haverkort, H. Hermanns, and J-P. Katoen. Model checking continuous-time Markov chains by transient analysis. In *Computer Aided Verification (CAV)*, pages 358–372, 2000.
4. J. T. Bradley and W.J. Knottenbelt. The ipc/HYDRA tool chain for the analysis of PEPA models. In *QEST '04: Proceedings of the The Quantitative Evaluation of Systems, First International Conference*, pages 334–335, Washington, DC, USA, 2004. IEEE Computer Society Press.
5. G. Clark, S. Gilmore, and J. Hillston. Specifying performance measures for PEPA. In *ARTS '99: Proceedings of the 5th International AMAST Workshop on Formal Methods for Real-Time and Probabilistic Systems*, pages 211–227, London, UK, 1999. Springer-Verlag.
6. 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.
7. J-M. Fourneau, M. Lecoz, and F. Quessette. Algorithms for an irreducible and lumpable strong stochastic bound. *Linear Algebra and its Applications*, 386:167–185, 2004.
8. S. Gilmore and J. Hillston. The PEPA workbench: A tool to support a process algebra-based approach to performance modelling. In *Proceedings of the Seventh International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, number 794 in Lecture Notes in Computer Science, pages 353–368, Vienna, May 1994. Springer-Verlag.
9. J. Hillston. *A Compositional Approach to Performance Modelling.* Cambridge University Press, 1996.
10. J. 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, Sep 2005. IEEE Computer Society Press.
11. J. Hillston and L. Kloul. An efficient Kronecker representation for PEPA models. In *Proceedings of the Joint International Workshop, PAPM-PROBMIV 2001*, number 2165 in Lecture Notes in Computer Science, pages 120–135. Springer-Verlag, 2001.
12. 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.
13. J.G. Kemeny and J.L. Snell. *Finite Markov Chains.* Springer-Verlag, 1976.
14. M.Z. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In *Computer Performance Evaluation: Modelling Techniques and Tools*, number 2324 in Lecture Notes in Computer Science, pages 200–204, 2002.
15. B. Plateau. On the stochastic structure of parallelism and synchronization models for distributed algorithms. *SIGMETRICS Performance Evaluation Review*, 13(2):147–154, 1985.
16. M.L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* John Wiley & Sons, Inc., New York, NY, USA, 1994.

17. M.J.A. Smith. Stochastic modelling of communication protocols from source code. *Electronic Notes in Theoretical Computer Science*, 190(3):129–145, 2007.
18. M.J.A. Smith. Compositional abstraction of PEPA models, 2009. In submission to QEST '09: The Sixth International Conference on the Quantitative Evaluation of Systems. Available from: `http://lanther.co.uk/papers/QEST09.pdf`.
19. D. Stoyan. *Comparison Methods for Queues and Other Stochastic Models*. Wiley & Sons, New York, NY, USA, 1983.
20. M. Tribastone. The PEPA plug-in project. In M. Harchol-Balter, M. Kwiatkowska, and M. Telek, editors, *Proceedings of the 4th International Conference on the Quantitative Evaluation of SysTems (QEST)*, pages 53–54. IEEE Computer Society Press, 2007.