Exploiting Modal Logic to Express Performance Measures

Graham Clark[1], Stephen Gilmore[1], Jane Hillston[1], and Marina Ribaudo[2]

[1] LFCS, The University of Edinburgh, Scotland.
Email: {gcla, stg, jeh}@dcs.ed.ac.uk
[2] Dipartimento di Informatica, Universitá di Torino, Italy.
Email: marina@di.unito.it

Abstract. Stochastic process algebras such as PEPA provide ample support for the component-based construction of models. Tools compute the numerical solution of these models; however, the stochastic process algebra methodology has lacked support for the specification and calculation of complex performance measures. In this paper we present a stochastic modal logic which can aid the construction of a reward structure over the model. We discuss its relationship to the underlying theory of PEPA. We also present a performance specification language which supports high level reasoning about PEPA models, and allows queries about their equilibrium behaviour. The meaning of the specification language has its foundations in the stochastic modal logic. We describe the implementation of the logic within the PEPA Workbench and a case study is presented to illustrate the approach.

1 Introduction

It has long been recognised that whilst Markovian models of simple computer systems can be constructed without explicit notational support, for complex systems use of some high-level modelling formalism becomes essential. A variety of formalisms exist, for example queueing networks [1], generalised stochastic Petri nets (GSPN) [2], stochastic activity networks (SAN) [3] and stochastic process algebras (SPA) [4]. Unfortunately corresponding high-level notational support has not been developed for querying models and checking performance specifications will be met. At best, some support for constructing a reward structure which captures the desired measure is provided.

In this paper we use Performance Evaluation Process Algebra (PEPA) [4], a compact formal language for modelling distributed computer and telecommunications systems. PEPA models are constructed by the composition of components which perform individual activities or cooperate on shared ones. Using such a model, a system designer can determine whether a candidate design meets both the behavioural and the temporal requirements demanded of it. Markovian SPA, such as PEPA, are enhanced with information about the duration of activities and, via a race policy, their relative probabilities. Several such languages have appeared in the literature; these include PEPA [4], TIPP [5] and EMPA [6].

Essentially these all propose the same approach to performance modelling: a corresponding continuous time Markov chain (CTMC) is generated via a structured operational semantics; linear algebra can then be used to solve the model in terms of equilibrium behaviour. This behaviour is represented as a probability distribution over all the possible states of the model.

This distribution is seldom the ultimate goal of performance analysis; instead the modeller is interested in performance *measures* which must be derived from this distribution via a *reward structure* defined over the CTMC [7]. A recent case study by first-time users of PEPA [8] reported that a significant proportion of the effort was spent in deriving the performance measures once steady state analysis was complete.

The study of temporal and modal logics in conjunction with process algebras is well-established. These logics express properties of systems which have a number of states, and in which there is a relation of succession. A modal logic is used to express finite behaviour. In a temporal logic one or more operators are introduced allowing reasoning to be carried out over infinite behaviour. Over the last decade, process algebras have been extended to capture additional information about systems, such as the relative probability of choices and the timing of actions. Analogously, extensions have been made to the syntax of the logics which allow properties to be expressed which reflect the additional information being captured [9–12].

Here we present a stochastic modal logic and explain how it may be used to specify performance measures over a PEPA model. The logic has several attractive features:

- it expresses properties in a high-level manner, focusing on the possible behaviours of the model rather than the states;
- properties expressed in this way remain invariant under model transformations such as automatic aggregation;
- a specification can be constructed in a compositional manner reflecting the compositional structure of the model.

Since we are interested in steady state behaviour it is perhaps surprising that we use a modal, and not a temporal, logic. However, as we will explain, we have found that a modal logic is sufficient for specifying a reward structure over a model assumed to be in equilibrium. This approach to specifying performance measures over PEPA models has been incorporated into the PEPA Workbench [13]. Finally, recognising that the logic expressions may be intimidating to some users, we have developed a high-level model query language. This language has foundations in the stochastic logic.

Earlier work by Clark proposed the use of a modal logic to define the reward structure over a PEPA model [14]. While demonstrating feasibility, this work suffered from a major drawback. The logic used did not include any representation of the timing aspects of PEPA and consequently does not have a clear relationship to the equivalence relations which have been established for the language, such as *strong equivalence*. In the current work we address this problem by developing a stochastic logic which takes full account of the random variables used to represent the duration of activities in PEPA. An earlier version of this work appeared in [15]. Our reward language has now been extended to take advantage of the compositional structure of models. Moreover, in this paper, we additionally describe the implementation of the approach and illustrate it using a more substantial case study.

In the next section we give a succinct summary of the PEPA language and motivate the need for a formal notation for specifying the performance of a PEPA model. Since we provide only a brief summary of PEPA here, the reader should consult [4] for full details. The PEPA Reward Language and its associated stochastic modal logic are presented in Section 3, whilst the implementation is described in Section 4. In Section 5 we illustrate our ideas with a simple, yet realistic, example. Finally, conclusions and future directions for the work are presented at the end of the paper.

2 PEPA

PEPA (Performance Evaluation Process Algebra) extends classical process algebra with the capacity to assign exponentially distributed durations to activities, which are described in an abstract model of a system. It is a concise formal language with a small number of grammar rules which define the well-formed terms in the language. An activity of action type α performed at rate r preceding P is denoted by $(\alpha, r).P$. Using the symbol $\top$ instead of a rate denotes *passive* participation in a shared activity. Choices are separated by $+$. Cooperation between P and Q over a set L of action types is $P \underset{L}{\bowtie} Q$ or $P \parallel Q$ if L is empty. Hiding the activities in L and thus denying their availability for cooperation gives the term P/L. The notation for definitional equality is $\stackrel{def}{=}$. The syntax may be formally introduced by means of the following grammar:

$$
\begin{aligned}
S &::= (\alpha, r).S \mid S + S \mid C_S \\
P &::= P \underset{L}{\bowtie} P \mid P/L \mid C
\end{aligned}
$$

where S denotes a *sequential component* and P denotes a *model component* which executes in parallel. C stands for a constant which denotes either a sequential or a model component, as introduced in a definition. C_S stands for constants which denote sequential components. The effect of this syntactic separation between these types of constants is to constrain legal PEPA components to be cooperations between sequential processes. This constraint is necessary for the underlying Markov process to be ergodic.

Using the structured operational semantic rules of the language it is possible to generate, directly from a PEPA model, a continuous time Markov process which faithfully encodes the temporal aspects of the PEPA model. The PEPA Workbench is used to check the well-formedness of PEPA models and to generate their Markov process representation. It detects faults such as deadlocks and cooperations which do not involve active participants. It is described in full in an earlier paper [13]. The steady state distribution may be found by applying any one of a number of linear algebra solution methods to the generator matrix. We have recently extended the Workbench with the capability to reduce models to a canonical form internally, thereby automatically aggregating the model [16]. This has considerable benefits in terms of tackling the state space explosion problem, but means that the states of the Markov process which is solved are no longer in one-to-one correspondence with the states of the PEPA model.

The formal aspects of PEPA have been exploited in developing the mapping from the language to the Markov process and in the automatic aggregation techniques. However, the extraction of performance measures from the resulting steady state probability distribution has been a largely ad hoc procedure. A reward structure is used to calculate appropriate expectations over the state space but determining which states should have a reward attached has relied on the knowledge of the modeller, and such states were characterised as syntactic terms [17]. Apart from relying on the modeller's insight, this technique also has the disadvantage of being incompatible with the automatic aggregation. Thus we have been motivated to develop a companion reward language for PEPA, centred on a stochastic modal logic, which characterises in behavioural terms the states to which rewards must be attached.

3 The PEPA reward language and stochastic modal logic

In this section we introduce a stochastic modal logic, which is used at the core of our reward language. In particular, the expression, and testing for satisfaction of equilibrium properties, can be seen to be closely related to the specification, and model checking of a formula expressed in *probabilistic modal logic* (PML [18]). We give a modified interpretation of such formulae suitable for reasoning about PEPA's continuous time models.

Previous work by Clark [14] proposed an approach to generating measures using traditional Hennessy-Milner logic (HML [19]). The idea was to capture the set of 'interesting' states of the model by partitioning the state space with a formula of the logic—those states that enjoy the property are then assigned a reward, such as a number, or a value based on 'local state' information, such as the rate at which the state may perform a particular activity. All uninteresting states are given a reward of 0. In this way, a reward vector is formally specified, and equilibrium measures such as utilisation and throughput may be calculated. However, the method was not ideal for several reasons. Firstly, it was ad hoc—the logic provided an initial partition only, meaning that a calculational technique was required in addition, in order to assign reward values. Secondly, the logic was qualitative only, in that it disregarded the *rate* at which a PEPA process could perform an activity, and only captured the fact that an activity was possible. These inadequacies led us to base our recent work on a more appropriate logic, namely Larsen and Skou's PML.

3.1 Probabilistic modal logic

The syntax of PML formulas is given by

$$F ::= \mathtt{tt} \mid \nabla_\alpha \mid \neg F \mid F_1 \wedge F_2 \mid \langle\alpha\rangle_\mu F$$

The models described in [18] are *probabilistic*, in that for any state P and any action α, there is a (discrete) probability distribution over the α-successors of P. Informally, the semantics of a formula ∇_α is the set of states unable to perform an α activity; and the semantics of $\langle\alpha\rangle_\mu F$ is the set of states such that each can make an α-transition with probability *at least* μ to a set of successors each of which satisfies F. We choose to modify slightly the interpretation of these formulae with respect to PEPA models. First we give a simple definition:

Definition 1 *Let S be a set of states. $P \xrightarrow{(\alpha,\nu)} S$ if and only if for all successors $P' \in S$, $P \xrightarrow{\alpha} P'$, and $\sum\{r : P \xrightarrow{(\alpha,r)} P', P' \in S\} = \nu$.*

Now let P be a model of a PEPA process. Then

$$\begin{array}{ll}
P \models \mathtt{tt} & \\
P \models \neg F & \text{if } P \not\models F \\
P \models F_1 \wedge F_2 & \text{if } P \models F_1 \text{ and } P \models F_2 \\
P \models \nabla_\alpha & \text{if } P \stackrel{\alpha}{\not\longrightarrow} \\
P \models \langle\alpha\rangle_\mu F & \text{if } P \xrightarrow{(\alpha,\nu)} S \text{ for some } \nu \geq \mu, \text{and for all } P' \in S, P' \models F.
\end{array}$$

Thus, the subscript μ present in formulae of the form $\langle\alpha\rangle_\mu F$ is now interpreted as a rate rather than a probability; if a state P is capable of doing activity α *quickly enough* arriving at a set of states S each of which satisfies F, then P satisfies $\langle\alpha\rangle_\mu F$. For the remainder of the paper, we will denote PML with this interpretation as PML_μ.

3.2 Relation of PML_μ to PEPA

In [18], Larsen and Skou show that PML exactly characterises *probabilistic bisimulation*, in the sense that two probabilistic processes are bisimilar if and only if they satisfy exactly the same set of PML formulae. With our modification to the semantics of PML, an analogous result holds for PEPA processes:

Theorem 1 (Modal characterisation of strong equivalence) *Let P be a model of a PEPA process. Then*

$$P \cong Q \text{ if and only if for all } F, P \models F \text{ if and only if } Q \models F$$

That is to say that two PEPA processes are *strongly equivalent* (in particular, their underlying Markov chains are *lumpably equivalent* [4]) if and only if they both satisfy the same set of PML_μ formulae (in our modified setting). A proof of this can be found in [15].

This result is not just of theoretical interest. It guarantees that if a transformation is applied to a model, resulting in the model being replaced by a strongly equivalent model, then we can expect that the new model satisfies the same formulae as the original model. Moreover if rewards are attached to equivalent states then the performance measures derived from the new model will be equal to the measures which would have been derived from the original.

The automatic aggregation procedure described in [16] and implemented in the PEPA Workbench, is based on the *isomorphism* relation of PEPA. However this relation is stronger than strong equivalence, meaning that any isomorphic models are necessarily strongly equivalent. Thus the above result implies that, using PML_μ formulae, the measures calculated from a model after aggregation will be identical to those that would have been calculated before. Therefore, from the user's point of view the aggregation remains transparent even when reward calculations are to be carried out. This is not the case for the other reward specification techniques used in SPA models.

Some PML_μ derived combinators are introduced in Equation 1. These add no expressive power to the logic, but will prove more succinct in expressing particular properties later. Informally, $[\alpha]_\mu F$ is the set of processes which can make an α-transition with rate at least μ, the derivative of which *must* satisfy F, and Δ_α is those processes which are able to perform an α activity.

$$
\begin{aligned}
\mathtt{ff} &\stackrel{def}{=} \neg\mathtt{tt} \\
[\alpha]_\mu F &\stackrel{def}{=} \neg\langle\alpha\rangle_\mu\neg F \\
\Delta_\alpha &\stackrel{def}{=} \neg\nabla_\alpha \\
F_1 \vee F_2 &\stackrel{def}{=} \neg((\neg F_1) \wedge (\neg F_2))
\end{aligned}
\tag{1}
$$

When specifying some performance measures it is natural to use the idea of model states, as well as model behaviour *in* a state. This can be smoothly reconciled with the use of a probabilistic logic, and the computation of the reward vector can thus be seen as a two-stage procedure. The method is simple, and standard in the theory of process logics—it is to extend the syntax of PML_μ with a set of *variables* V, and for a given model P with state space $\mathcal{S}$, to extend the semantics with a *valuation* function $\mathcal{V} : V \to 2^{\mathcal{S}}$.

$$
\begin{array}{l}
F ::= \mathtt{tt} \mid \nabla_\alpha \mid \neg F \mid F_1 \wedge F_2 \mid \langle\alpha\rangle_\mu F \mid X \\
P \models X \quad \text{if and only if} \quad P \in \mathcal{V}(X)
\end{array}
$$

The intuition is that a variable $X \in V$ represents a property which is true in a particular subset of the state space. This allows the expression of formulae such as $\neg(\langle\mathtt{transmit}\rangle_{120} FailState)$, where $FailState$ is understood to represent an undesirable portion of the state space—"it is not the case that it is possible to efficiently transmit a network packet and finish in a failure state". We have found that it is useful to compose an additional *monitor process* with the model, and use this to label states. This is a well-known technique in verification but, in general, it relies on the skill of the modeller to design such a process so that it does not alter the state space of the original model.

Due to the predicative semantics of PML_μ, i.e. formulae evaluate to a characteristic function over the set of states, it is straightforward to specify utilisation and reliability measures which only require reward values of 0 or 1. The relation of PML_μ to measures such as throughput which require real-valued rewards, is less direct. For this reason PML_μ is not, in general, used alone, but as part of the richer PEPA Reward Language.

3.3 PEPA Reward Language

The definition of a reward structure in the PEPA Reward Language is comprised of two parts:

- a *reward specification*, which associates a value with a logical formula, specifying a behaviour;
- an *attachment* which determines with which process derivatives a particular reward specification is associated, reflecting the compositional structure of the model.

The meaning of the reward specification will depend on how it is "attached" to a PEPA model—the associated value may depend on information local to the derivative under consideration. This will be explained when the semantics of the reward language is described below.

Formally, each reward specification can be considered as a pair consisting of a logical formula and a reward expression. Following the attachment, the formula is checked against a set of subcomponents within the context of the model as a whole. When the formula is satisfied the corresponding derivative is assigned a reward. The value of the reward corresponds to the evaluation of a simple arithmetic expression.

Syntax and Semantics of Reward Expressions The syntax of reward expressions, given below, is very simple; indeed, it captures little more than a straightforward syntax for arithmetic. The only additions to this are three bound variables.

$$\begin{aligned} e &::= (e) \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \mid e_1/e_2 \mid \mathsf{atom} \\ \mathsf{atom} &::= r \in \mathbb{R} \mid cur \mid rate(\alpha \in \mathcal{A}ct) \end{aligned}$$

The bound variables *cur* and $rate()$ will be used to denote real numbers. The meaning will be dependent on the reward structure being built, and the particular labelled multi-transition system which results from the PEPA model under consideration. They exist for pragmatic reasons—they are useful in specifying performance measures. The variable *cur* is intended to give the reward expression access to a "currently" assigned reward, allowing reward expressions to make use of previous assignments. The variable $rate()$, allows activity rates to be used in expressions—specifically, reward values can be assigned to a derivative P which make use of the transition rate from P to successor derivatives via an activity of type α[1]. This is the way in which timing information may be incorporated into reward specifications.

The objective is to define a reward function ρ, such that if P is the PEPA process under consideration, and ds computes its derivative set, then $\rho : ds(P) \longrightarrow \mathbb{R}$. That is, given a derivative (in fact a state of the transition system) ρ gives the reward assigned to that derivative. The complete reward structure may be built up by successively overlaying the effects of different reward functions—this explains the inclusion of the variable cur.

Given this reward assignment function, the semantic function relies on the context c of a PEPA process P to define the meaning of reward expressions; the semantics are given in Figure 1.

$$
\begin{aligned}
\| (e) \|_{P_c} &= \| e \|_{P_c} \\
\| e_1 \ op \ e_2 \|_{P_c} &= \| e_1 \|_{P_c} \ op \ \ \| e_2 \|_{P_c} \\
\| r \|_{P_c} &= r \\
\| cur \|_{P_c} &= \rho(c[P]) \\
\| rate(\alpha) \|_{P_c} &= \sum \{ r : c[P] \xrightarrow{(\alpha,r)} \}
\end{aligned}
$$

Fig. 1. Semantics of reward expressions

The notation $\| \ e \ \|_{P_c}$ denotes the evaluation of expression e with respect to P in the context c; $c[P]$ denotes P in the context c and the binary operator op is intended to capture the obvious binary operators defined in the syntax above. The following definition completes the definition of a reward specification.

Definition 2 *A* reward specification *is a pair* (F, e)*, where* F *is a* PML$_\mu$ *formula and* e *is a reward expression.*

Creating a Reward Structure with Attachments When the behavioural specification captured by the reward specification relates to subcomponents within a model, rather than to the model as a whole, an *attachment* may be used to guide how the formula is to be tested against the model. For instance, given a large PEPA model, it may be interesting to only examine the performance of a single component queue. It should be possible to disregard the behaviour of the rest of the model, at least up to its interaction with the queue under examination. To achieve this, contexts are employed.

Definition 3 *An* attachment*,* a*, is a triple* $(\sigma, c, \langle P_1, \ldots, P_n \rangle)$*, where* σ *is a reward specification,* c *is a context, and* P_i *are PEPA processes, for* $1 \leq i \leq n$*.*

[1] If α is passive in P $rate()$ is undefined.

The attachment allows the modeller to choose which subcomponents are of interest—the subcomponents are the processes P_i.

Now let $\rho : ds(P) \longrightarrow \mathbb{R}$ represent a function constructing a reward structure, and let P be a PEPA process. Assume an initial value of $\rho(P') = 0$, for all $P' \in ds(P)$. The semantic function for attachments takes as an argument a reward assignment function ρ and evaluates to a new function, say ρ'. This possibly modified assignment function will reflect any new rewards that have been assigned to the PEPA model. Its argument is a *sequence* of attachments. A sequence is chosen so a reward structure can be built sequentially, allowing one reward expression to make use of the values present in the partially constructed reward structure. Evaluating such a sequence of attachments is trivial—each is evaluated individually, in order. This is shown below.

$$\begin{aligned} \| \langle\rangle \|_\rho &= \rho \\ \| \langle a_i, a_{i+1}, \ldots, a_m \rangle \|_\rho &= \| \langle a_{i+1}, a_{i+2}, \ldots, a_m \rangle \|_{\rho'} \quad \text{where } \rho' = \| a_i \|_\rho \end{aligned}$$

The meaning of an attachment can now be defined.

Definition 4 (Semantics of an attachment) *The meaning of an attachment* $a_i = ((F, e), c, \langle P_1, \ldots, P_n \rangle)$ *is a value determined as follows:*

$$\| a_i \| = \begin{cases} \| e \|_{c[P_1, \ldots, P_n]} & \textit{if } \langle P_1, \ldots, P_n \rangle \models_c F \\ 0 & \textit{otherwise.} \end{cases}$$

ρ' is created by ordinary function perturbation and the end result is a function which constructs a reward structure over the derivative space of a PEPA process. More details can be found in [20].

4 Implementation

The PEPA Workbench has been extended to allow the use of a subset of the PEPA Reward Language. This allows the modeller to express behavioural properties using PML$_\mu$, though currently the use of contexts is not implemented. The implementation automatically generates a reward structure which provably generates the same performance measures for any two strongly equivalent models. This means that the modeller may apply aggregation to a PEPA model without having to alter the description of any performance measures.

Given a PEPA model, the Workbench generates a representation of the model's generator matrix which is then solved. In order to generate the matrix, it is necessary for the Workbench to traverse the entire state space of the PEPA model. After this traversal, for each state of the model, a reward specification can be checked, and if satisfied, a reward assigned. The algorithm used to implement this subset of the reward language employs a simple model checking procedure for PML$_\mu$.

5 Case study: self-checking distributed computation

Our case study comes from the TIRAN project ("TaIlorable fault toleRANce frameworks for embedded applications", IT Project 28620). In this project example fault-tolerant applications are provided by industrial partners (Siemens and ENEL). The objective of the project is to build a modular framework in which the faults, errors and failures can be methodically considered. One module in the framework is the TIRAN *backbone*, responsible for tolerating internal and external faults. The backbone is currently under implementation and here we present a simplification of one of the key algorithms which is used.

The setting is an embedded system made up of a number of loosely coupled nodes without shared memory. Communication is by synchronous message passing. Agents run on each of these nodes. One of the agents is designated the *manager* of the others. In our simplified model, we assume that the manager never experiences failures. Without this simplifying assumption it would be necessary to describe a leadership election in addition to the processing which we describe here. We concentrate on the internal faults which are detected in the *self-checking* phase of the distributed computation. During self-checking the agents monitor their own progress and may declare themselves to be faulty.

The manager periodically broadcasts a *query* to all of the agents which asks "are you alive?". The manager waits to receive one of two possible replies: the agent responds "this agent is alive" (*alive*) or "this agent is faulty" (*faulty*). The latter means that the agent has detected and trapped a fault and is in an uncertain state. If no reply is received within a certain timeout the manager assumes that there is a hardware fault on the node on which the agent was running. In this case a recovery process is initiated for the node.

Each agent is composed of three sub-processes. These are responsible for fault *detection*, *isolation* and *restarting*. The detector sets a local flag to indicate that there is no fault at present. It then reports back to the manager that this agent is alive. The same flag is periodically unset by the isolation process. If the isolator process later reads the flag and finds it still unset then it reports back to the manager that this agent is faulty. If the agent is faulty, the restart subprocess ensures the correct re-initialisation of the agent.

We reflect the structure of the system in the components which are used in the model description. We define a *Manager* component, and generic *Agent* components. The manager controls *Daemon* processes, one for each agent which it must manage. Within an agent, we have sub-components for detection, isolation and restarting. Since we work with exponential assumptions, the deterministically timed timeouts are here approximated by exponential distributions.

$$Manager \stackrel{\mathit{def}}{=} (query, q).Manager$$

$$Daemon_0 \stackrel{\mathit{def}}{=} (query, \top).Daemon_1$$

$$Daemon_1 \stackrel{\mathit{def}}{=} (alive, \top).Daemon_0 + (faulty, \top).Daemon_2 + (timeout, t_1).Daemon_3$$

$$Daemon_2 \stackrel{\mathit{def}}{=} (restartAgent, \top).Daemon_0 + (timeout, t_2).Daemon_3$$

$$Daemon_3 \stackrel{\mathit{def}}{=} (repairNode, rn).Daemon_0$$

The agent composes sub-processes, synchronising on the relevant activities.

$$
\begin{aligned}
Agent &\stackrel{\text{def}}{=} \left((Detector_0 \underset{L_1}{\bowtie} Isolator_0) \underset{L_2}{\bowtie} Restart_0 \right) \\
\text{where } L_1 &= \{ \, timeout, flag_0, flag_1, restartAgent, repairNode \, \} \\
L_2 &= \{ \, timeout, alive, faulty, restartAgent, repairNode \, \}
\end{aligned}
$$

The detector process is concerned with the value of the flag and with reporting that the agent is still alive. It registers any timeouts which occur and witnesses the recovery of the agent or the node.

$$
\begin{aligned}
Detector_0 &\stackrel{\text{def}}{=} (flag_1, f_1).Detector_1 + (flag_0, \top).Detector_2 + (timeout, \top).Detector_3 \\
Detector_1 &\stackrel{\text{def}}{=} (alive, a).Detector_0 + (timeout, \top).Detector_3 \\
Detector_2 &\stackrel{\text{def}}{=} (restartAgent, \top).Detector_0 + (timeout, \top).Detector_3 \\
Detector_3 &\stackrel{\text{def}}{=} (repairNode, \top).Detector_0
\end{aligned}
$$

The isolator process receives the “are you alive?” query from the manager. After checking the value of the flag it has the responsibility of sending the fault report, if this is appropriate. As with the detector process it registers timeouts and any recovery processes.

$$
\begin{aligned}
Isolator_0 &\stackrel{\text{def}}{=} (query, \top).Isolator_1 \\
Isolator_1 &\stackrel{\text{def}}{=} (flag_1, \top).Isolator_0 + (flag_0, f_0).Isolator_2 + (timeout, \top).Isolator_4 \\
Isolator_2 &\stackrel{\text{def}}{=} (faulty, f).Isolator_3 + (timeout, \top).Isolator_4 \\
Isolator_3 &\stackrel{\text{def}}{=} (restartAgent, \top).Isolator_0 + (timeout, \top).Isolator_4 \\
Isolator_4 &\stackrel{\text{def}}{=} (repairNode, \top).Isolator_0
\end{aligned}
$$

The final process is responsible for restarting the agent after a fault. It must witness the other reports from the other sub-processes in order that it does not restart the agent when this action is not necessary.

$$
\begin{aligned}
Restart_0 &\stackrel{\text{def}}{=} (alive, \top).Restart_0 + (faulty, \top).Restart_1 + (timeout, \top).Restart_2 \\
Restart_1 &\stackrel{\text{def}}{=} (restartAgent, ra).Restart_0 + (timeout, \top).Restart_2 \\
Restart_2 &\stackrel{\text{def}}{=} (repairNode, \top).Restart_0
\end{aligned}
$$

The complete system is configured as a composition of agents paired with an instance of the daemon. This structure in the model represents a single manager process in the system with channels to communicate with each of the agents running on the remote nodes. In order to model a broadcast of the “are you alive?” message, all of these paired agent and manager processes are required to synchronise when the message is sent. The replies are received asynchronously by each instance of the daemon and all must be received before another broadcast can be sent. In the case of a fault of an agent or a node the restart action must be performed on that node before the next broadcast can be sent. Below we show the system with two managed agents.

$$\begin{aligned} \mathit{ManagedAgent} &\stackrel{\mathit{def}}{=} \mathit{Agent} \underset{L}{\bowtie} \mathit{Daemon}_0 \\ \mathit{System} &\stackrel{\mathit{def}}{=} (\mathit{ManagedAgent} \underset{\{\,\mathit{query}\}}{\bowtie} \mathit{ManagedAgent}) \underset{\{\,\mathit{query}\}}{\bowtie} \mathit{Manager} \end{aligned}$$

The synchronisation set L between the agent and the associated daemon contains the activity types *query*, *alive*, *faulty*, *timeout*, *restartAgent*, and *repairNode*.

As further managed agents are added the state space of the system increases proportionally. Of course, the model exhibits symmetries which can be exploited in order to eliminate uninteresting variants of states. The aggregated state space is more compact to store and more efficient to solve in order to find the steady state probability distribution of the system. This aggregation is performed automatically using the PEPA Workbench.

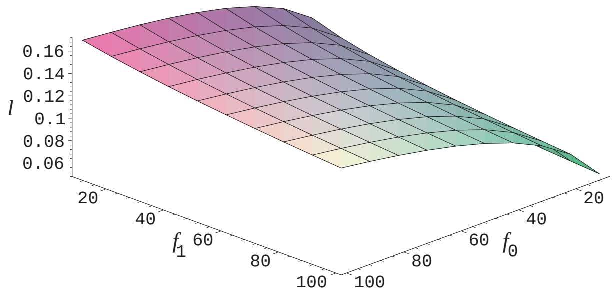
5.1 Investigations into the model

The effect of aggregation has been to eliminate some syntactic presentations of states. It may perhaps be the case that the states are still present, but that their presentation has been altered. This can be a distinct disadvantage to the modeller unless an expressive language is also provided which allows the definition of performance measures over the model without reference to the syntactic presentation of states. Without such a language, the modeller must have a thorough knowledge of the aggregation method deployed in order to identify the syntactic presentations which remain after aggregation. Even with complete knowledge of the aggregation method used, the definition of performance measures would still be made more awkward and unnatural for the modeller.

The logical notation which we provide for expressing performance measures avoids the need for the modeller to understand the aggregation method employed by the tool, and to reproduce its effect on the states of interest. The same definition of a measure can be used without change, whether or not aggregation is performed. Further, the definition of measures which we use is independent of the number of agents in the system, and can also be re-used unchanged for larger configurations of the system.

To illustrate the use of the PEPA Reward Language, we compute here two significant performance measures for the system. The first is the potential for *lag* in the system. This occurs when one of the agents has detected a local fault but its appointed daemon has not yet registered this information. This can be simply expressed as Δ_{faulty}. This captures all of the states which have the potential to perform a transition *faulty*. A reward of f (the fault registration rate) is assigned to these states.

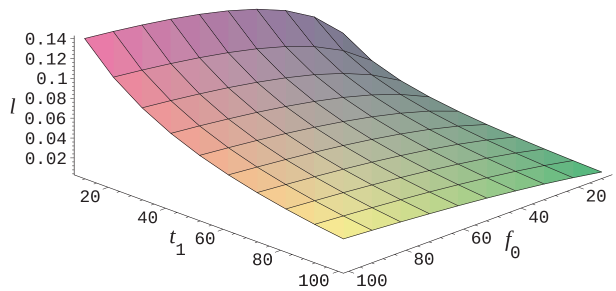
The effect of varying the flag registration rates, f_0 and f_1, on this measure is shown in Figure 2. As might be expected, the measure is more sensitive to variations in f_0 since this is the rate associated with the flag_0 activity, which leads to the states where the *faulty* message can be sent. In Figure 3 we show



Rates	Values
q	5
t_1	10
t_2	10
rn	1
a	50
f	50
ra	1

Fig. 2. Plot of lag l while varying rates f_0 and f_1

that the effect of timing the agents out at a faster rate is to pre-empt fault registration, causing agents to be restarted by the manager more often.



Rates	Values
q	5
t_2	10
rn	1
f_1	100
a	50
f	50
ra	1

Fig. 3. Plot of lag l while varying rates f_0 and t_1

The second measure which we define captures the potential for *successive timeouts*. This measure would be of interest if tuning the system to find the right balance between waiting for faulty agents to self-repair or pre-empting them by software on hardware resets.

Timeout activities are distinguished in this model because they are the only activity which is performed at more than one rate (t_1 and t_2). We express our measure with a third variable, t.

$$\langle \mathit{timeout} \rangle_t \langle \mathit{timeout} \rangle_t \mathtt{tt}$$

By varying the relative values of t, t_1 and t_2, this measure will include one, two or none of the classes of timeout activity. Thus the ability to use rate variables in reward expressions allows us to include just the activities which are of interest.

6 Related work

We have presented the PEPA Reward Language, a notation for the description of performance specifications which relate to stochastic process algebra models expressed in PEPA. The study of a self-checking distributed computation illustrates the way in which a modeller would use the PEPA Reward Language to reason about the performance of a model.

An alternative approach to constructing reward structures over SPA models is presented in [21]. In that paper, Bernardo extends the syntax of EMPA, so that each activity is augmented with a reward value,i.e. each activity is represented by a triple comprised of type, rate and reward values. In the generated reward structure, the reward assigned to each state is the sum of the rewards associated with activities enabled in that state. Bernardo has constructed an equivalence relation which respects the additional reward information.

Our choice of PML was motivated by its simplicity, and its link to PEPA's strong equivalence. Other research in the area of probabilistic verification has links to our approach. Logics such as that presented by Hansson and Jonsson [9] are able to specify bounds on probabilistic properties, but crucially these are probabilities over behaviours from a specified state. Recent work by de Alfaro [12] addresses the problem of specifying "long-run" average properties of probabilistic systems, with non-deterministic choices made by adversaries. De Alfaro defines *experiments* to represent interesting model behaviour patterns. These experiments associate a real-valued outcome with a pattern of behaviour, and are considered to occur infinitely often. In [22] Baier *et al.* describe how a temporal logic can be used to specify transient properties of CTMCs. Their model checking procedure aims to establish whether such properties hold or not. This is quite distinct from our use of logic to construct the reward structure used to calculate steady state performance measures.

Acknowledgements

The authors gratefully acknowledge the helpful comments from the anonymous referees. Graham Clark and Jane Hillston are supported by the EPSRC COMPA grant. Stephen Gilmore is supported by the 'Distributed Commit Protocols' EPSRC grant and by Esprit Working group FIREworks. This work was completed whilst Marina Ribaudo was visiting the LFCS, supported by a grant from CNR.

References

1. P. G. Harrison and N. M. Patel. *Performance Modelling of Communication Networks and Computer Architectures.* International Computer Science Series. Addison-Wesley, 1993.
2. M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets.* John Wiley, 1995.
3. J.F. Meyer, A. Movaghar, and W.H. Sanders. Stochastic activity networks: Structure, behavior and application. In *Proc of Int. Workshop on Timed Petri Nets*, pages 106–115, Torino, Italy, 1985. IEEE Computer Society Press.

4. J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.
5. N. Götz, U. Herzog, and M. Rettelbach. Multiprocessor and Distributed System Design: The Integration of Functional Specification and Performance Analysis using Stochastic Process Algebras. In *Performance'93*, 1993.
6. M. Bernardo and R. Gorrieri. A tutorial on EMPA: A theory of concurrent processes with nondeterminism, priorities, probabilities and time. *Theoretical Computer Science*, 201:1–54, July 1998.
7. R. A. Howard. *Dynamic Probabilistic Systems*, volume II: Semi-Markov and Decision Processes, chapter 13, pages 851–915. John Wiley & Sons, New York, 1971.
8. J. Bryans, H. Bowman, and J. Derrick. Analysis of a Multimedia Stream using Stochastic Process Algebra. In C.Priami, editor, *Proc. of 6th Int. Workshop on Process Algebras and Performance Modelling*, pages 51–69, Nice, France, September 1998.
9. H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.
10. M. Huth and M. Kwiatkowska. Quantitative analysis and model checking. In *Proceedings, Twelth Annual IEEE Symposium on Logic in Computer Science*, pages 111–122, Warsaw, Poland, 29 June–2 July 1997. IEEE Computer Society Press.
11. V. Hartonas-Garmhausen. *Probabilistic Symbolic Model Checking with Engineering Models and Applications*. PhD thesis, Carnegie Mellon University, 1998.
12. L. de Alfaro. How to specify and verify the long-run average behavior of probabilistic systems. In *LICS: IEEE Symposium on Logic in Computer Science*, 1998.
13. S. Gilmore and J. Hillston. The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling. In G. Haring and G. Kotsis, editors, *Proc. of 7th Conf. on Mod. Techniques and Tools for Computer Perf. Eval.*, volume 794 of *LNCS*, pages 353–368, 1994.
14. G. Clark. Formalising the Specification of Rewards with PEPA. In *Proc. of 4th Process Algebras and Performance Modelling Workshop*, pages 139–160, July 1996.
15. G. Clark, S. Gilmore, and J. Hillston. Specifying performance measures for PEPA. In J.P. Katoen, editor, *Proc. of 5th Int. AMAST Workshop on Real-Time and Probabilistic Systems*, volume 1601 of *LNCS*, pages 211–227, Bamberg, Germany, 1999. Springer-Verlag.
16. S. Gilmore, J. Hillston, and M. Ribaudo. An efficient algorithm for aggregating PEPA models. To appear in *IEEE Transactions on Software Engineering*, 2000.
17. G. Clark, S. Gilmore, J. Hillston, and N. Thomas. Experiences with the PEPA Performance Modelling Tools. *IEE Proceedings—Software*, 146(1):11–19, February 1999. Special issue of papers from 14th UK Performance Engineering Workshop.
18. K. G. Larsen and A. Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94(1):1–28, September 1991.
19. M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, January 1985.
20. G. Clark. *Techniques for the Construction and Analysis of Algebraic Performance Models*. PhD thesis, The University of Edinburgh, 2000.
21. M. Bernardo. An Algebra-Based Method to Associate Rewards with EMPA Terms. In P. Degano, R. Gorrieri, and A. Marchetti Spaccamela, editors, *24th Int. Colloquium on Automata, Languages and Programming*, volume 1256 of *LNCS*, pages 358–368, Bologna, Italy, July 1997. Springer-Verlag.
22. C. Baier, J.-P. Katoen, and H. Hermanns. Approximate symbolic model checking of continuous time Markov chains. In *Proceedings of CONCUR'99*, volume 1664 of *LNCS*, pages 146–162, 1999.