Performance modelling with PEPA nets and PRISM

Stephen Gilmore* Jane Hillston* Leïla Kloul*† Marina Ribaudo‡

Abstract

PEPA nets are coloured stochastic Petri nets which are used for modelling mobile software and computing systems. We describe the compilation of models in this language into an imperative language of static modules which has no built-in concept of mobility, the PRISM modelling language. We show the application of PEPA nets and PRISM to the modelling and analysis of the Mobile IP protocol.

1 Introduction

PEPA nets [1] extend the PEPA stochastic process algebra [3] by allowing PEPA components to be used as the tokens of a coloured stochastic Petri net. Communication between tokens at different places is not allowed; for tokens to communicate they must first meet at one of the places of the net. While they are together in the same place they may communicate but when one or other of them moves to another place the communication between them must cease until another firing of the net brings the tokens together again in the same place.

In addition to the PEPA components which are used as tokens, a PEPA net contains *static components*. Each static component is resident in a place of the net and is unable to move. Static components may communicate with tokens while they are present or they can communicate with other static components in the same place. Communication between static components at different places is not allowed.

Restricting communication according to these rules provides a credible formalism for representing mobile code systems where the tokens of a PEPA net model stateful mobile objects under a system of dynamic binding of names. A PEPA component has local state which can be modified by performing timed actions, either individually or in cooperation with another component. These activities define an interface analogous to the interface made up of the methods which can be invoked on an object.

The static components in a PEPA net represent immobile parts of the system which may be either hardware (such as servers) or software (such as databases). Modern real-world systems are typically made up of a mixture of mobile and immobile components in this way.

The legitimate analysis of such a system depends crucially on the accurate modelling of such patterns of communication, regulated by rules such as those characterised above (components in separate places cannot synchronise; and a mobile component can only communicate with a static one by migrating and communicating locally). However, many effective state-of-the-art analysis tools

*Laboratory for Foundations of Computer Science, The University of Edinburgh, Edinburgh EH9 3JZ, Scotland. Email: {stg, jeh, leila} @lfcs.ed.ac.uk

†On leave from PRISM, Université de Versailles, 45, Av. des Etats-Unis 78000 Versailles, France.

‡Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, Via Dodecaneso 35, 16146 Genova, Italia. Email: ribaudo@disi.unige.it

do not provide support for differentiating between simple local communication and the movement or migration of processes which can change the allowable pattern of communication.

In this paper we describe a method of compiling a model of a mobile code system into an equivalent model in a language which has no explicit notion of either location or mobility between locations. This method has been implemented in a software tool which we have applied to the example which we present in this paper.

The target of our translation is the modelling language used by the PRISM [4] probabilistic model checker. This language is a probabilistic extension of Alur and Henzinger's *reactive modules* [5]. The PRISM tool provides the ability to model check probabilistic CSL formulae against a stochastic model and to solve CTMCs for their equilibrium probability distribution. For this purpose it represents a CTMC as an MTBDD, providing a compact representation which allows the modeller to solve significantly larger models than those which can be handled by sparse matrix-based approaches such as those provided by the PEPA Workbench [6] and its extension to the PEPA Workbench for PEPA nets.

It is well understood in theory that BDDs can provide compact representation of large, symmetric state spaces. In addition to this we know from previous work that the PRISM tool is more efficient in practice at solving large PEPA models than is the PEPA Workbench. Accessing the BDD-based representation and solution technology offered by PRISM is the primary motivation for choosing it as the target language of our translation. The ability to model check CSL formulae is an additional benefit of this choice.

Structure of this paper: In the next section we present descriptions of the PEPA nets and PRISM modelling languages. In Section 3 we describe the method of translating PEPA nets to PRISM. In Section 4 we describe our Mobile IP [7] case study. In Section 5 we describe the implementation of our translation method as an extension to the PEPA Workbench for PEPA nets which connects to the PRISM probabilistic model checker. Conclusions are presented in Section 6.

2 Background

In this section we provide a brief overview of PEPA nets, PEPA and PRISM. Readers interested in supporting definitions and theory should consult [1], [3] and [8].

2.1 PEPA nets

The tokens of a PEPA net are terms of the PEPA stochastic process algebra which define the behaviour of components via the activities they undertake and the interactions between them. The syntax of PEPA nets is given in Figure 1. In that grammar S denotes a *sequential component* and P denotes a *concurrent component* which executes in parallel. I stands for a constant which denotes either a sequential or a concurrent component, as bound by a definition.

A PEPA net differentiates between two types of change of state. We refer to these as *firings* of the net and *transitions* of PEPA components. Each are special cases of PEPA activities. Transitions of PEPA components will typically be used to model small-scale (or *local*) changes of state as components undertake activities. Firings of the net will typically be used to model macro-step (or *global*) changes of state such as context switches, breakdowns and repairs, one thread yielding to another, or a mobile software agent moving from one network host to another. The set of all firings is denoted by $\mathcal{A}_f$. The set of all transitions is denoted by $\mathcal{A}_t$.

$$N \quad ::= \quad D^+ M \qquad \text{(net)}$$

(definitions and marking)

$$\begin{array}{rcll} M & ::= & (M_{\mathbf{P}}, \ldots) & \text{(marking)} \\ M_{\mathbf{P}} & ::= & \mathbf{P}[C, \ldots] & \text{(place marking)} \end{array}$$

(marking vectors)

$$\begin{array}{rcll} D & ::= & I \stackrel{\mathit{def}}{=} S & \text{(component defn)} \\ & \mid & \mathbf{P}[C] \stackrel{\mathit{def}}{=} P[C] & \text{(place defn)} \\ & \mid & \mathbf{P}[C, \ldots] \stackrel{\mathit{def}}{=} P[C] \underset{L}{\bowtie} P & \text{(place defn)} \end{array}$$

(identifier declarations)

$$\begin{array}{rcll} S & ::= & (\alpha, r).S & \text{(prefix)} \\ & \mid & S + S & \text{(choice)} \\ & \mid & I & \text{(identifier)} \end{array}$$

(sequential components)

$$\begin{array}{rcll} P & ::= & P \underset{L}{\bowtie} P & \text{(cooperation)} \\ & \mid & P/L & \text{(hiding)} \\ & \mid & P[C] & \text{(cell)} \\ & \mid & I & \text{(identifier)} \end{array}$$

(concurrent components)

$$\begin{array}{rcll} C & ::= & \text{`\_'} & \text{(empty)} \\ & \mid & S & \text{(full)} \end{array}$$

(cell term expressions)

Figure 1: The syntax of PEPA nets

A PEPA net is made up of PEPA *contexts*, one at each place in the net. A context consists of a number of *static* components (possibly zero) and a number of *cells* (at least one). Like a memory location in an imperative program, a cell is a storage area to be filled by a datum of a particular type. In particular in a PEPA net, a cell is a storage area dedicated to storing a PEPA component. The components which fill cells can circulate as the tokens of the net. In contrast, the static components cannot move.

Definition 1 *A PEPA net $\mathcal{N}$ is a tuple $\mathcal{N} = (\mathcal{P}, \mathcal{T}, I, O, \ell, \pi, \mathcal{C}, D, M_0)$ such that*

- *$\mathcal{P}$ is a finite set of places;*
- *$\mathcal{T}$ is a finite set of net transitions;*
- *$I : \mathcal{T} \to \mathcal{P}$ is the input function;*
- *$O : \mathcal{T} \to \mathcal{P}$ is the output function;*
- *$\ell : \mathcal{T} \to (\mathcal{A}_f, \mathbb{R}^+ \cup \{\top\})$ is the labelling function, which assigns a PEPA activity ((type, rate) pair) to each transition. The rate determines the negative exponential distribution governing the delay associated with the transition;*
- *$\pi : \mathcal{A}_f \to \mathbb{N}$ is the priority function which assigns priorities (represented by natural numbers) to firing action types;*
- *$\mathcal{C} : \mathcal{P} \to P$ is the place definition function which assigns a PEPA context, containing at least one cell, to each place;*
- *D is the set of token component definitions;*
- *M_0 is the initial marking of the net.*

2.2 PRISM

PRISM [4] is a probabilistic model checker developed at the University of Birmingham. It supports three models: discrete-time Markov chains (DTMC), Markov decision processes (MDP), and continuous-time Markov chains (CTMC). Model analysis is performed through model checking specifications in the probabilistic temporal logics PCTL (for DTMC and MDP), and CSL (for CTMC).

In order to define and analyse a model this tool requires two input files: a *description of the system* under investigation and a *set of properties* to be checked against it.

The system is described in the input PRISM language: a state-based specification language whose fundamental components are *modules* and *variables*. A system is in fact composed of a number of modules which can interact with each other. The behaviour of each module is described by a set of *commands* and each module contains a number of local, integer-valued variables. The values of these variables at any given time constitute the state of each module. The global state of the whole system is determined by the local state of all modules.

Modules can run in parallel or can synchronise following the CSP-like synchronisation style: commands can be labelled with action names which can be used to force two or more modules to execute transitions simultaneously. The rate of the synchronisation transition is equal to the *product of the rates* of the individual transitions. Since the product of the rates does not always meaningfully represent the rate of a synchronised transition [9], a common technique is to have a passive/active synchronisation, where the rate of the passive transition is equal to one.

PRISM parses the model description and then constructs the corresponding probabilistic model. Depending on the type of model specified, this will either be a DTMC, an MDP or a CTMC. PRISM also computes the set of reachable states of this model and identifies any deadlock states.

Assuming there are no problems during the model construction, the user can then specify in a second file one or more properties to be verified against the model. These properties are written in temporal logic: PCTL for DTMCs and MDPs; CSL for CTMCs. PRISM performs model checking for each of these properties and determines whether or not each one is satisfied. The steady-state probability distribution for a CTMC can be automatically derived.

2.2.1 Importing PEPA models to PRISM

For CTMCs, PRISM can read model descriptions written in the stochastic process algebra PEPA. The tool compiles such descriptions into the PRISM input language and then constructs the model as normal. The language accepted by the PEPA to PRISM compiler is actually a subset of PEPA. The restrictions applied to the language are firstly that component identiers can only be bound to sequential components (formed using prefix and choice and references to other sequential components only). Secondly, each local state of a sequential component must be explicitly named. Finally, active/active synchronisations are not allowed since the PRISM definition of these differs from the PEPA definition. Multi-way synchronisation is permitted but every PEPA synchronisation must have exactly one active component.

In the next section we will discuss how to translate PEPA nets into standard PEPA models so that the PEPA to PRISM compiler can be reused to input PEPA nets to PRISM as well.

3 Translating PEPA nets to PEPA

The algorithm that translates PEPA nets into PEPA models is composed of a number of different steps which are in turn described in the following sections.

Before starting the translation, a *preprocessing* phase in necessary in order to rename firings that in the PEPA nets share the same input place and same action type. This is done by checking the arcs specification in the PEPA net file. For example, arcs $P_k \xrightarrow{(\alpha,r_1)} P_j$ and $P_k \xrightarrow{(\alpha,r_2)} P_i$ need to be changed into $P_k \xrightarrow{(\alpha_1,r_1)} P_j$ and $P_k \xrightarrow{(\alpha_2,r_2)} P_i$ and the triples (α, α_1, r_1) and (α, α_2, r_2) are stored in an array denoted by $Fire$. Several data types and a function are used in the algorithm and are introduced below. In addition we use the standard operators of the PEPA theory [3] such as $\vec{\mathcal{A}}(C)$, returning the complete action type set of C.

All	// *keeps track of all action types of the PEPA model*
$NumberSC[\,]$	// *keeps track of the number of occurrences of static components*
$Number\alpha_j$	// *keeps track of the number of renamings of activity* α_j
$OccurSC[\,]$	// *keeps track of the static components appearing several times*
$Fire[\,]$	// *keeps track of the triples resulting from the preprocessing phase*
$TargetP[\,]$	// *triples* (P, α, r) *where* P *is the output place of the arc and* (α, r) *is its label*
$Extract(\alpha_j)$	// *returns the original action type* α *from the triple in Fire, Nil otherwise*
$\mathcal{L}_{sync}(T_i, SC_k)$	// *synchronisation set between token type* T_i *and the static component* SC_k

3.1 Step 1: Translating Static Components

This step concerns the translation of static components. We need to count the occurrences of static components within the places of the PEPA net and rename action types and derivatives of each occurrence of the static component. This allows us to avoid erroneous synchronisations since in a PEPA net synchronisation is possible only when the components are resident in the same place.

```
for each static component SC_i
    NumberSC[i] ← 0;
    OccurSC[i] ← ∅;
    for each place P_j ∈ 𝒫
        if (SC_i in P_j)  then  NumberSC[i] ← NumberSC[i] + 1;
    make a copy of SC_i;        // retain the original unchanged
    All ← All ∪ Ā(SC_i);        // save all action types in All
    if (NumberSC[i] > 1)  then
        OccurSC[i] ← SC_i;
        for k = 1 ... NumberSC[i] − 1
            make an instance SC_ik of SC_i, renumbering activities and derivatives;
            All ← All ∪ Ā(SC_ik);        // save all action types in All
```

3.2 Step 2: Translating Cells

In this step we consider all the PEPA net places containing cells and we associate one component with each cell within a place. Each cell definition is built by considering the input and output arcs to the place itself. Moreover, each cell can be in two states: *empty*, denoted by a subscript 0 in its derivative, and *full* denoted by a subscript 1. Firing a token into the place changes the state of the cell from empty to full. Firing a token out of the cell changes its state from full to empty. This allows us to prevent a token from moving to a place where all of the cells are already full.

for each place $P_j \in \mathcal{P}$

  for each cell i in P_j

    create a component $Cell_{i0}$;

    for each input arc $P_k \overset{(\alpha,r)}{\longrightarrow\Box\longrightarrow} P_j$

      $All \leftarrow All \cup \{\alpha\}$;

      $Cell_{i0} \leftarrow Cell_{i0} + (\alpha, r).Cell_{i1}$; // *avoiding redundant activities*

    for each output arc $P_j \overset{(\alpha,r)}{\longrightarrow\Box\longrightarrow} P_k$

      $All \leftarrow All \cup \{\alpha\}$;

      $Cell_{i1} \leftarrow Cell_{i1} + (\alpha, r).Cell_{i0}$;

3.3 Step 3: Translating Tokens

Token definitions need to be changed according to their dynamic behaviour. Two aspects are taken into account: the movement of a token into a new place after a firing and its interaction with static components within a PEPA net place. Again, in order to avoid erroneous synchronisations, we need to create new names for action types and derivatives. This is done for both net firings with the same labels and for actions on which tokens and static components synchronise. In this case, we consider only static components which appear several times in the net. Recall that in Step 1 the action types and the derivatives of each occurrence of the static component were renamed. Therefore here we need to copy the derivatives of the token where the synchronising action type appears and we do this for each renaming of the synchronising action.

for each token type T_i

// movement of the token in the net

for each action type $\alpha_j \in \mathcal{A}_f(T_i)$

$Number\alpha_j \leftarrow 0$;

$TargetP[j] \leftarrow \emptyset$;

for each arc $P_k \xrightarrow{(action,r)} P_l$

$\alpha_j^* \leftarrow Extract(action)$;

if $(\alpha_j = \alpha_j^*)$ **then** $Number\alpha_j \leftarrow Number\alpha_j + 1$;

$TargetP[j] \leftarrow P_l$;

if $(Number\alpha_j > 1)$ **and** $(TargetP[\,]$ items are not the same$)$ **then**

for each distinct place in $TargetP[\,]$

add the corresponding activity $(action, r)$ in $TargetP$ followed by a new component identifier C^*;

duplicate the sequential component C resulting from the execution of action α_j in T_i, replacing C with C^*;

else make a copy of the current derivative of T_i;

// interaction of the token with the static components

for each place $P_l \in \mathcal{P}$

if $(T_i$ in $P_l)$ **then**

for each static component SC_k in $OccurSC$

for each action type $\alpha_j \in \mathcal{L}_{sync}(T_i, SC_k)$

for $x = 1 \ldots NumberSC[k]$

duplicate the derivatives where α_j appears in T_i for each renaming of α_j during the translation of the static component SC_k;

3.4 Step 4: Building the System Equation

We build the system equation by putting in parallel all the components generated in the previous steps. Then we force them to synchronise on common action types.

3.5 Example: A mobile agent system

We present a small example to reinforce the reader's understanding of the translation algorithm. In this example a mobile software agent visits three sites. It interacts with static software components at these sites and has two kinds of interactions. When visiting a site where a network probe is present it interrogates the probe for the data which it has gathered on recent patterns of network traffic. When it returns to the central co-ordinating site it dumps the data which it has harvested to the master probe. The master probe analyses the data.

The structure of the application is as represented by the PEPA net in Figure 2. This marking of the net shows the mobile agent resident at the central co-ordinating site. As a memory aid, in both the net and the PEPA token definitions, we print in bold the names of those activities which can cause a firing of the net. In this example the activities which can cause a firing of the net are **go** and **return**.

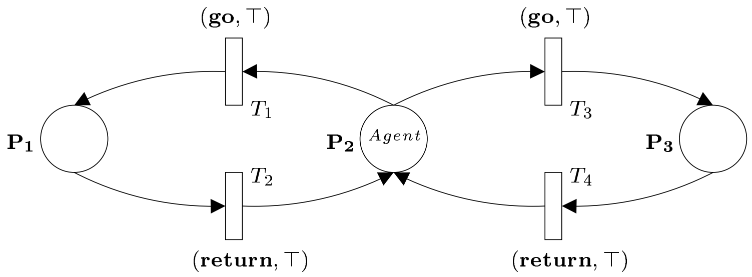

Figure 2: A simple mobile agent system

Formally, we define the places of the net as shown in the PEPA context definitions below.

$$\begin{aligned}
\mathbf{P_1}[Agent] &\stackrel{def}{=} Agent[Agent] \underset{\{interrogate\}}{\bowtie} Probe \\
\mathbf{P_2}[Agent] &\stackrel{def}{=} Agent[Agent] \underset{\{dump\}}{\bowtie} Master \\
\mathbf{P_3}[Agent] &\stackrel{def}{=} Agent[Agent] \underset{\{interrogate\}}{\bowtie} Probe
\end{aligned}$$

The behaviour of the components is given by the following PEPA definitions.

$$\begin{array}{rcl@{\qquad}rcl}
Agent & \stackrel{def}{=} & (\mathbf{go}, \lambda).Agent' & Master & \stackrel{def}{=} & (dump, \top).Master' \\
Agent' & \stackrel{def}{=} & (interrogate, r_i).Agent'' & Master' & \stackrel{def}{=} & (analyse, r_a).Master \\
Agent'' & \stackrel{def}{=} & (\mathbf{return}, \mu).Agent''' & Probe & \stackrel{def}{=} & (monitor, r_m).Probe + \\
Agent''' & \stackrel{def}{=} & (dump, r_d).Agent & & & (interrogate, \top).Probe
\end{array}$$

The initial marking of the net has one $Agent$ token in place $\mathbf{P_2}$ and no other tokens:

$$(Agent[\_] \underset{\{interrogate\}}{\bowtie} Probe, Agent[Agent] \underset{\{dump\}}{\bowtie} Master, Agent[\_] \underset{\{interrogate\}}{\bowtie} Probe)$$

3.5.1 Translating the mobile agent example into PEPA

The example has two static components $Master$ and $Probe$. As the latter appears twice in the PEPA net model (in places $\mathbf{P_1}$ and $\mathbf{P_3}$), the application of the algorithm generates two instantiations of this component $Probe$ and $Probe_1$. Thus in the PEPA model we have:

$$\begin{aligned}
Master &\stackrel{def}{=} (dump, \top).Master' \\
Master' &\stackrel{def}{=} (analyse, r_a).Master \\
\\
Probe &\stackrel{def}{=} (monitor, r_m).Probe + (interrogate, \top).Probe \\
Probe_1 &\stackrel{def}{=} (monitor_1, r_m).Probe_1 + (interrogate_1, \top).Probe_1
\end{aligned}$$

As each place contains one cell, the translation algorithm generates three new components $Cell_{i0}$, where $i = 1 \ldots 3$. Firings and transitions are no longer distinguished because there is only one class of activities in PEPA and so we no longer embolden the names *go* and *return*.

$$\begin{array}{rcl@{\qquad}rcl}
Cell_{10} & \stackrel{def}{=} & (go_1, \top).Cell_{11} & Cell_{20} & \stackrel{def}{=} & (return, \top).Cell_{21} \\
Cell_{11} & \stackrel{def}{=} & (return, \top).Cell_{10} & Cell_{21} & \stackrel{def}{=} & (go_1, \top).Cell_{20} + (go_2, \top).Cell_{20}
\end{array}$$

$$\begin{array}{rcl}
Cell_{30} & \stackrel{def}{=} & (go_2, \top).Cell_{31} \\
Cell_{31} & \stackrel{def}{=} & (return, \top).Cell_{30}
\end{array}$$

Component *Agent* is the token in the net and its definition is modified to take into account its complete dynamic behaviour in the net.

$$\begin{array}{rcl@{\qquad}rcl}
Agent & \stackrel{def}{=} & (go_1, \lambda).Agent' + (go_2, \lambda).Agent'_1 & Agent'' & \stackrel{def}{=} & (return, \mu).Agent''' \\
Agent' & \stackrel{def}{=} & (interrogate_1, r_i).Agent'' & Agent''' & \stackrel{def}{=} & (dump, r_d).Agent \\
Agent'_1 & \stackrel{def}{=} & (interrogate, r_i).Agent'' & & &
\end{array}$$

The PEPA system equation is as follows:

$$System \stackrel{def}{=} (Cell_{10} \underset{K_1}{\bowtie} (Probe \underset{K_2}{\bowtie} (Agent \underset{K_3}{\bowtie} (Cell_{21} \underset{K_4}{\bowtie} (Master \underset{K_5}{\bowtie} (Probe_1 \underset{K_6}{\bowtie} Cell_{30}))))))$$

$$\text{where} \quad \begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
K_1 & = & \{go_1, return\} & K_2 & = & \{interrogate\} & K_3 & = & \{go_1, go_2, return\} \\
K_4 & = & \{dump\} & K_5 & = & \{interrogate_1\} & K_6 & = & \{go_2, return\}
\end{array}$$

4 Case study: Mobile IP

In this section we present a larger case study of modelling with PEPA nets. Based on the Internet protocol, Mobile IP is a standard protocol that makes user mobility transparent to applications and higher level protocols like TCP. It allows a *mobile node* to freely roam between network links and to remain always accessible. To achieve this the mobile node uses two IP addresses: a *home address* which is statically assigned on its *home network* and a *care-of address* which changes at each new point of attachment. On the home network, a proxy known as a *home agent* is responsible for forwarding all packets which are addressed to the mobile node on to its current care-of address.

Whenever the mobile node moves, it sends to its home agent a binding update message containing its home address, its current care-of address and the lifetime for which the binding should be honoured [7]. The home agent may refresh a binding cache entry by regularly requesting the transmission of the latest care-of address.

When the mobile node receives, via its home agent, a packet from a correspondent, it sends a binding update message to this correspondent. The correspondent may maintain a binding cache allowing its transmit function to redirect the packets to the mobile node's current care-of address [7]. The mobile node maintains a list of all its current correspondents and has to send them a binding update message each time it changes its point of attachment. Figure 3 summarizes the main steps of the Mobile IP protocol.

In this study we assume that the system is composed of N domain or network hosts, besides the home network and the correspondent network. We assume that the home agent sends requests to the mobile node to update the care-of address. Once the correspondent has the care-of address, its transmit function redirects the packet to this address, saving one network hop relative to the route through the home agent [7].

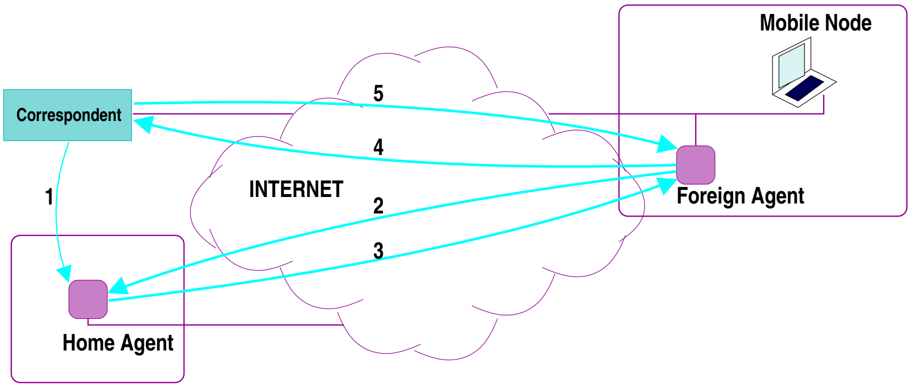

Figure 3: The Mobile IP Protocol

4.1 The PEPA net Model

The system is modelled using the PEPA net model depicted in Figure 4 where the home network of the mobile node is represented using a place called **HOME_M**. The networks to which the mobile node may move are modelled by places **DOMAIN**$_i$ where $i = 1, \ldots, N$. Place **HOME_C** models the home network of the mobile node's correspondent.

Note that for the sake of readability in Figure 4 the rates of the activities labelling the firings are omitted. Moreover only the arcs between **DOMAIN**$_1$ and **HOME_M** on one hand, and **HOME_C** on the other, are depicted. The arcs between the other domains and **HOME_M** and **HOME_C** are analogous.

To model the part of the protocol which manages the interaction between the home agent and the mobile node during its stay in place **DOMAIN**$_i$, we use two components *ProtoMA* and *CommuMA*. Similarly we use components *ProtoMC* and *CommuMC* to model the protocol interactions between the mobile node and the correspondent. Moreover the exchanges between the home agent and the correspondent are modelled using components $ProtoAC$ and $CommuAC$. Components $Mobile$, *Agent* and *Corresp* model the behaviour of, respectively, the mobile node, the home agent and the current correspondent of the mobile node. In contrast to $Mobile$, the last two components are static. All these components and places are explained in detail in the following.

4.1.1 The Components

- **Component ProtoMA$_\mathbf{i}$** models the protocol part which consists of updating the care-of address at the home agent level. When the mobile node changes its point of attachment, it generates a binding update message, action $generate_{bua}$. Here, the rate associated with this action is unspecified since *ProtoMA* does not generate this message, but has just to transmit it. This is modelled using the firing action $\mathbf{transmit_{bua}}$ with rate r_1. Once component *ProtoMA* is in **HOME_M**, it allows the home agent to update the care-of address using the synchronizing action *updateA* with rate λ. With each place **DOMAIN**$_i$, $i = 1, \ldots, N$, is

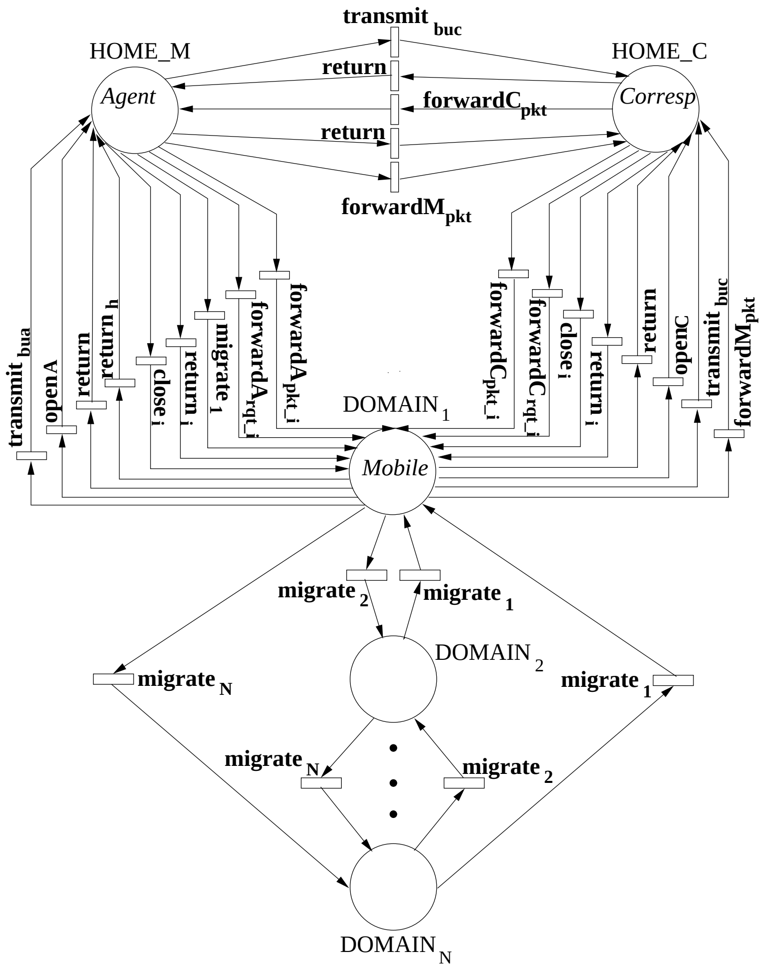

Figure 4: The PEPA net model

associated a component $ProtoMA_i$ whose behaviour is as follows.

$$ProtoMA_i \stackrel{def}{=} (generate_{bua}, \top).(\mathbf{transmit_{bua}}, r_1).ProtoMA_{i1}$$
$$ProtoMA_{i1} \stackrel{def}{=} (updateA, \lambda).(\mathbf{return_i}, r_2).ProtoMA_i$$

- **Component $\mathbf{CommuMA_i}$** allows us to model the exchanges between the home agent and the mobile node once the binding update message has been received by the home agent. Once the communication is established (via *newSessionA*), the home agent will either generate a packet or an update request to be forwarded to the mobile node using firing action $\mathbf{forwardA_{pkt\_i}}$ or $\mathbf{forwardA_{rqt\_i}}$. Back in place $\mathbf{DOMAIN}_i$, $CommuMA_i$ delivers the packet ($deliverA_{pkt}$) or the request ($deliverA_{rqt}$) to the mobile node. The delivery may fail if the mobile node has already migrated somewhere else or just returned to its home network. Component $CommuMA_i$ is associated with place $\mathbf{DOMAIN_i}$. Formally, the behaviour of this component is as follows.

$$
\begin{array}{lcl}
CommuMA_i & \stackrel{def}{=} & (newSessionA, \top).(\mathbf{openA}, \alpha_1).CommuMA_{i0} \\
CommuMA_{i0} & \stackrel{def}{=} & (generateA_{pkt}, \top).(\mathbf{forwardA_{pkt\_i}}, \alpha_3).CommuMA_{i2} \\
& & + (generateA_{rqt}, \top).(\mathbf{forwardA_{rqt\_i}}, \alpha_{\mathbf{5}}).CommuMA_{i4} \\
& & + (\mathbf{close_i}, \alpha_2).CommuMA_i \\
CommuMA_{i2} & \stackrel{def}{=} & (deliverA_{pkt}, \mu_1).CommuMA_{i3} + (fail, \mu_2).CommuMA_{i3} \\
CommuMA_{i3} & \stackrel{def}{=} & (\mathbf{return}, \alpha_4).CommuMA_{i0} \\
CommuMA_{i4} & \stackrel{def}{=} & (deliverA_{rqt}, \mu_3).CommuMA_{i3} + (fail, \mu_4).CommuMA_{i3}
\end{array}
$$

- **Component ProtoMC$_\mathbf{i}$** models the protocol part which consists of updating the care-of address at the correpondent level. Once the binding update message is generated, it is forwarded to the correspondent using firing activity $\mathbf{transmit_{buc}}$. Then it allows the home agent to update the care-of address using the synchronizing action *updateC* with rate λ_1.

$$
\begin{array}{lcl}
ProtoMC_i & \stackrel{def}{=} & (generate_{buc}, \top).(\mathbf{transmit_{buc}}, r_3).ProtoMC_{i1} \\
ProtoMC_{i1} & \stackrel{def}{=} & (updateC, \lambda_1).(\mathbf{return_i}, r_4).ProtoMC_i
\end{array}
$$

- **Component CommuMC$_\mathbf{i}$** allows us to model the effective exchanges between the mobile node and the correspondent. Once the communication is established (*newSessionC*) by the mobile node, the correspondent may either generate a packet or an update request that will be forwarded using firing transition $\mathbf{forwardC_{pkt\_i}}$ or $\mathbf{forwardC_{rqt\_i}}$ respectively. The delivery may succeed or fail if the mobile has already moved. In the first case, the mobile node may either stay silent (*silent*) or generate a packet ($generateM_{pkt}$) which is forwarded to the correspondent using firing transition $\mathbf{forwardM_{pkt}}$. The communication is considered finished when transition $\mathbf{close_i}$ is fired at the correspondent level. A component $CommuMC_i$ is associated with each place $\mathbf{DOMAIN}_i$, $i = 1, \ldots, N$.

$$
\begin{array}{lcl}
CommuMC_i & \stackrel{def}{=} & (newSessionC, \top).(\mathbf{openC}, \beta_1).CommuMC_{i0} \\
CommuMC_{i0} & \stackrel{def}{=} & (generateCM_{pkt}, \top).(\mathbf{forwardC_{pkt\_i}}, \beta_{\mathbf{3}}).CommuMC_{i1} \\
& & +(generateCM_{rqt}, \top).(\mathbf{forwardC_{rqt\_i}}, \beta_{\mathbf{5}}).CommuMC_{i4} \\
& & +(\mathbf{close_i}, \beta_2).CommuMC_i \\
CommuMC_{i1} & \stackrel{def}{=} & (deliverC_{pkt}, v_1).CommuMC_{i2} + (fail, v_2).CommuMC_{i5} \\
CommuMC_{i2} & \stackrel{def}{=} & (generateM_{pkt}, \top).(\mathbf{forwardM_{pkt}}, \beta_4).CommuMC_{i3} \\
& & +(silent, v_3).CommuMC_{i5} \\
CommuMC_{i3} & \stackrel{def}{=} & (deliverM_{pkt}, v_4).CommuMC_{i0} \\
CommuMC_{i4} & \stackrel{def}{=} & (deliverC_{rqt}, v_5).CommuMC_{i5} + (fail, v_6).CommuMC_{i5} \\
CommuMC_{i5} & \stackrel{def}{=} & (\mathbf{return}, \beta_{\mathbf{6}}).CommuMC_{i0}
\end{array}
$$

- **Component ProtoAC** models the case where the mobile node returns to its home network and has to send a binding update message to its current correspondent. This component is associated with place **HOME_M** and returns to it once the care-of address has been updated at the correspondent level using activity *updateCh* with rate λ_2.

$$
\begin{array}{lcl}
ProtoAC & \stackrel{def}{=} & (generateH_{buc}, \top).(\mathbf{transmit_{buc}}, r_5).ProtoAC_1 \\
ProtoAC_1 & \stackrel{def}{=} & (updateCh, \lambda_2).(\mathbf{return}, r_6).ProtoAC
\end{array}
$$

- **Component CommuAC** models the communication between the correspondent and the home agent if the mobile node has left its home network or with the mobile node itself if not. It is associated with place **HOME_C** and forwards the packets generated by the correspondent to the mobile node in its home network. This is modelled using the firing action $\mathbf{forwardC_{pkt}}$ with rate γ_1. If the mobile node is present, the packets are delivered to it with action $deliverC_{pkt}$. Component $CommuAC$ then forwards the packets generated by the mobile node to the correspondent. This is done with the firing action $\mathbf{forwardM_{pkt}}$ with rate γ_2. If the mobile node is not in its home network, the correspondent's packets are saved ($saveC_{pkt}$) by the home agent and component $CommuAC$ goes back to **HOME_C** using firing action **return** at rate γ_3.

$$
\begin{array}{lcl}
CommuAC & \stackrel{def}{=} & (generateCA_{pkt}, \top).(\mathbf{forwardC_{pkt}}, \gamma_1).CommuAC_1 \\
CommuAC_1 & \stackrel{def}{=} & (deliverC_{pkt}, w_1).CommuAC_2 + (saveC_{pkt}, w_2).CommuAC_4 \\
CommuAC_2 & \stackrel{def}{=} & (generateM_{pkt}, w_3).(\mathbf{forwardM_{pkt}}, \gamma_2).CommuAC_3 \\
 & & +(silent, w_4).CommuAC_4 \\
CommuAC_3 & \stackrel{def}{=} & (deliverMh_{pkt}, w_5).CommuAC \\
CommuAC_4 & \stackrel{def}{=} & (\mathbf{return}, \gamma_3).CommuAC
\end{array}
$$

- **Component Mobile** models the behaviour of the mobile node whatever its current point of attachment. The mobile node may either generate packets, receive packets or simply choose to move to another network. This is modelled using actions $generateM_{pkt}$, $deliverC_{pkt}$ and $\mathbf{migrate}_i$ respectively. When the mobile node changes its point of attachment, it first generates a binding update message for its home agent with action $generate_{bua}$ at rate τ_2 and then opens a new communication session ($newSessionA$). It may then receive from its home agent either an update request ($deliverA_{rqt}$) or a correspondent's packet ($deliverA_{pkt}$). In this last case, it generates a binding update message for the correspondent with action $generate_{buc}$ at rate τ_3 and establishes a new communication session ($newSessionC$). The mobile node may stay in the current network or with probability p_j moves to another network j with firing action $\mathbf{migrate}_j$, $j \neq i$, i being the current host network of the mobile node. It may also just return to its home network with the firing action $\mathbf{return_h}$.

$$
\begin{array}{lcl}
Mobile & \stackrel{def}{=} & (generateM_{pkt}, \tau_1).Mobile + (deliverC_{pkt}, \top).Mobile \\
 & & +\sum_{i=1}^{N}(\mathbf{migrate}_i, p_i \times \delta_1).Mobile_1 \\
Mobile_1 & \stackrel{def}{=} & (generate_{bua}, \tau_2).(newSessionA, s_1).Mobile_2 \\
Mobile_2 & \stackrel{def}{=} & (deliverA_{pkt}, \top).Mobile_3 + (deliverA_{rqt}, \top).Mobile_1 + (\mathbf{return_h}, \delta_2).Mobile \\
Mobile_3 & \stackrel{def}{=} & (generate_{buc}, \tau_3).(newSessionC, s_2).Mobile_4 \\
Mobile_4 & \stackrel{def}{=} & (deliverA_{rqt}, \top).Mobile_7 + (deliverA_{pkt}, \top).Mobile_4 + (deliverC_{rqt}, \top).Mobile_8 \\
 & & +(deliverC_{pkt}, \top).Mobile_4 + (generateM_{pkt}, \tau_4).Mobile_4 + (\mathbf{return_h}, \delta_2).Mobile_5 \\
 & & +\sum_{j=1/j\neq i}^{N}(\mathbf{migrate}_j, p_j \times \delta_5).Mobile_6 \\
Mobile_5 & \stackrel{def}{=} & (generateh_{buc}, \tau_5).Mobile \\
Mobile_6 & \stackrel{def}{=} & (generate_{bua}, \tau_6).(newSessionA, s_1).Mobile_3 \\
Mobile_7 & \stackrel{def}{=} & (generate_{bua}, \tau_7).Mobile_4 \\
Mobile_8 & \stackrel{def}{=} & (generate_{buc}, \tau_3).Mobile_4
\end{array}
$$

- **Component Agent** models the home agent's behaviour. It is given by the following PEPA net equations:

$$
\begin{aligned}
Agent &\stackrel{def}{=} (updateA, \top).Agent_1 + (saveC_{pkt}, \top).Agent_3 \\
Agent_1 &\stackrel{def}{=} (saveC_{pkt}, \top).Agent_2 + (generateA_{rqt}, \nu_1).Agent_1 + (updateA, \top).Agent_1 \\
Agent_2 &\stackrel{def}{=} (generateA_{pkt}, \nu_2).Agent_1 + (updateA, \top).Agent_2 + (saveC_{pkt}, \top).Agent_2 \\
Agent_3 &\stackrel{def}{=} (saveC_{pkt}, \top).Agent_3 + (updateA, \top).Agent_2
\end{aligned}
$$

- **Component Corresp** models the behaviour of a correspondent of the mobile node. It may generate packets to send to the mobile node in its home network ($generateCA_{pkt}$) and receive packets from the mobile node when it is still there ($deliverMh_{pkt}$). It may also receive a binding update message from the mobile node ($updateC$). In this case, the correspondent may generate packets ($generateCM_{pkt}$) or update requests ($generateCA_{rqt}$) to send to the mobile node in its current attachment point. Action type *updateCh* models the case where the correspondent receives a binding update message from the mobile node back in its home network.

$$
\begin{aligned}
Corresp &\stackrel{def}{=} (generateCA_{pkt}, c_1).Corresp + (deliverMh_{pkt}, \top).Corresp \\
&\quad +(updateC, \top).Corresp_1 \\
Corresp_1 &\stackrel{def}{=} (generateCM_{pkt}, c_2).Corresp_1 + (generateCM_{rqt}, c_3).Corresp_1 \\
&\quad +(deliverM_{pkt}, \top).Corresp_1 + (updateC, \top).Corresp_1 \\
&\quad +(updateCh, \top).Corresp
\end{aligned}
$$

4.1.2 The Places

The places of the PEPA net are defined as follows:

$$
\begin{aligned}
HOME_M &\stackrel{def}{=} \Big(ProtoAC\ [ProtoAC] \underset{L_1}{\bowtie} \Big(Mobile\ [Mobile] \underset{L_2}{\bowtie} \Big(CommuAC\ [\_] \underset{L_3}{\bowtie} \\
&\quad \Big((CommuMA_1\ [\_]|| \ldots ||CommuMA_N\ [\_]) \underset{L_4}{\bowtie} \Big(Agent \underset{L_5}{\bowtie} \\
&\quad (ProtoMA_1\ [\_]|| \ldots ||ProtoMA_N\ [\_])))))) \\
HOME_C &\stackrel{def}{=} \Big(\Big(\Big(\Big(ProtoAC\ [\_] \underset{L_6}{\bowtie} Corresp\Big) \underset{L_7}{\bowtie} (CommuMC_1\ [\_]|| \ldots ||CommuMC_N\ [\_])\Big) \underset{L_8}{\bowtie} \\
&\quad (ProtoMC_1\ [\_]|| \ldots ||ProtoMC_N\ [\_])) \underset{L_9}{\bowtie} CommuAC\ [CommuAC]\Big) \\
DOMAIN_i &\stackrel{def}{=} \Big(\Big(\Big(\Big(ProtoMA_i\ [ProtoMA_i] \underset{L_{10}}{\bowtie} Mobile\ [\_]\Big) \underset{L_{11}}{\bowtie} CommuMA_i\ [CommuMA_i]\Big) \underset{L_{12}}{\bowtie} \\
&\quad ProtoMC_i\ [ProtoMC_i]) \underset{L_{13}}{\bowtie} CommuMC_i\ [CommuMC_i]\Big)
\end{aligned}
$$

where $i = 1 \ldots N$ and the synchronizing sets are defined as follows

$$
\begin{aligned}
L_1 &= \{generateH_{buc}\} & L_8 &= \{updateC\} \\
L_2 &= \{deliverC_{pkt}, generateM_{pkt}\} & L_9 &= \{generateCA_{pkt}, deliverMh_{pkt}\} \\
L_3 &= \{saveC_{pkt}\} & L_{10} &= \{generate_{bua}\} \\
L_4 &= \{generateA_{pkt}, generateA_{rqt}\} & L_{11} &= \{newSessionA, deliverA_{pkt}, deliverA_{rqt}\} \\
L_5 &= \{updateA\} & L_{12} &= \{generate_{buc}\} \\
L_6 &= \{updateCh\} & L_{13} &= \{newSessionC, generateM_{pkt}, deliverC_{pkt}, deliverC_{rqt}\} \\
& & L_7 &= \{deliverM_{pkt}, generateCM_{pkt}, generateCM_{rqt}\}
\end{aligned}
$$

5 Implementation

We have implemented our translation from PEPA nets to PEPA as an extension to our existing tool, The PEPA Workbench for PEPA Nets. We process the PEPA output from the Workbench with the compiler from PEPA to PRISM which we have implemented previously. The PEPA-to-PRISM compiler is a component of PRISM v1.3 [10].

By factoring the translation into two phases in this way we were able to make a clear separation of concerns between two significant phases of the translation. During the first phase we eliminate the token mobility in the input PEPA net. During the second phase we re-express a PEPA model in PRISM's reactive modules language of imperative guarded commands. In addition to providing this helpful separation of concerns, this factorisation allowed us to re-use one of our existing software components (the PEPA-to-PRISM compiler) in its entirety.

We have found the PRISM tool to have a very efficient state-space generation method. For example, we compiled our Mobile IP model above with one domain from the original PEPA net into a PEPA model input to PRISM. The PRISM tool generated a state space of over 2.8 million states (actually 2,800,202) and over 16 million transitions (actually 16,410,778) in 13.2 seconds on a 1.6GHz Pentium IV with 256Mb of memory running RedHat Linux 7.1. The compilation from the PEPA net to PEPA took 0.15 seconds. PRISM additionally provides an expressive description language which allow us to easily express quantities such as the proportion of packets which have to be forwarded in the Mobile IP protocol.

Compiling our PEPA nets models first to PEPA conveys an additional benefit. There are many efficient and powerful analysis tools which already support the PEPA stochastic process algebra as one of their input languages. The approach of compiling PEPA nets to PEPA allows us now to also use these tools for the more expressive modelling language of PEPA nets. For example, we could now simulate or solve PEPA nets models with the Möbius multi-paradigm modelling framework [11] because it provides support for PEPA [12]. As another example, we could use the distributed Dnamaca Petri net solver to perform transient analysis or compute first passage time distributions for large PEPA net models by using the PEPA Workbench together with Jeremy Bradley's translator from PEPA to Dnamaca.

Set against this, there are losses which are attendant to using PEPA as an intermediate language in this way. The primary loss is that we are unable to process PEPA net models which use priorities because the PEPA stochastic process algebra does not support them. A direct translation from PEPA nets to PRISM could have preserved any uses of priorities in the input PEPA net so we have to view this as a shortcoming of our two-phase translation but we consider it to be a relatively minor one.

The PEPA tools are available for download from the PEPA Web site located at `http://www.dcs.ed.ac.uk/pepa/` together with documentation, example models and copies of papers on PEPA and PEPA nets.

6 Conclusions

Building performance models of realistic real-world systems is an activity which requires careful attention to detail in order to accurately model the intended behaviour of the system. Proceeding with care during this part of the modelling process is a wise investment of effort. If the initial performance model contains errors then all of the computational expense incurred in solving the model and all of the intellectual effort invested in the analysis and interpretation of the results obtained would at best be wasted. In general interpreting a model with errors could lead to making flawed economic or strategic decisions based on erroneous conclusions made from erroneous results. For this reason we consider it important to work with structured, high-level modelling languages which directly support the concepts and idioms of the application domain, such as code mobility. In this paper we have shown how a mobile code modelling language, PEPA nets, can be mapped into another language, the reactive modules of PRISM, where mobility is not directly expressible in the language and there is no notion of physically or logically separated locations being a barrier to communication.

Our translation from PEPA nets to PRISM is not excessively complicated but it has enough subtleties that it was worthwhile to implement it in a software tool to prevent making mistakes that could easily be made if applying the method by hand. We extended our existing workbench for PEPA nets for this purpose. We applied this to a case study of modelling the Mobile IP protocol.

Acknowledgements: Gilmore, Hillston and Kloul are supported by the DEGAS (Design Environments for Global ApplicationS) project IST-2001-32072 funded by the FET Proactive Initiative on Global Computing. Their collaboration with Ribaudo was supported by the Dipartimento di Informatica, Università degli Studi di Torino. The authors have benefited from their collaboration with Linda Brodo and Corrado Priami on mapping PEPA nets into the stochastic π-calculus and Stochastic CCS. The PEPA to PRISM compiler was developed in collaboration with Gethin Norman and Dave Parker of The University of Birmingham.

References

[1] S. Gilmore, J. Hillston, L. Kloul, and M. Ribaudo. PEPA nets: A structured performance modelling formalism. To appear in *Performance Evaluation*. Extended version of [2], 2003.

[2] S. Gilmore, J. Hillston, and M. Ribaudo. PEPA nets: A structured performance modelling formalism. In T. Field, P.G. Harrison, J. Bradley, and U. Harder, editors, *Proceedings of the 12th International Conference on Modelling Tools and Techniques for Computer and Communication System Performance Evaluation*, number 2324 in Lecture Notes in Computer Science, pages 111–130, London, UK, April 2002. Springer-Verlag.

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

[4] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. In J.-P. Katoen and P. Stevens, editors, *Proc. 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 52–66. Springer, April 2002.

[5] R. Alur and T.A. Henzinger. Reactive modules. *Formal Methods in System Design: An International Journal*, 15(1):7–48, July 1999.

[6] 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.

[7] D. Johnson, C. Perkins, and J. Arkko. Mobility support in IPv6. IETF Mobile IP Working Group Internet-Draft `draft-ietf-mobileip-ipv6-20`, January 2003.

[8] D. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems.* PhD thesis, University of Birmingham, 2002.

[9] J. T. Bradley and N. Davies. Reliable performance modelling with approximate synchronisations. In *Proceedings of PAPM'99: Process Algebra and Performance Modelling*, pages 99–118, Prensas Universitarias de Zaragoza, September 1999.

[10] D. Parker. *PRISM 1.3 User's Guide.* University of Birmingham, February 2003. `http://www.cs.bham.ac.uk/~dxp/prism`.

[11] G. Clark, T. Courtney, D. Daly, D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. Webster. The Möbius modeling tool. In *Proceedings of the 9th International Workshop on Petri Nets and Performance Models*, pages 241–250, Aachen, Germany, September 2001.

[12] G. Clark and W.H. Sanders. Implementing a stochastic process algebra within the Möbius modeling framework. In L. de Alfaro and S. Gilmore, editors, *Proceedings of the first joint PAPM-PROBMIV Workshop*, volume 2165 of *Lecture Notes in Computer Science*, pages 200–215, Aachen, Germany, September 2001. Springer-Verlag.