Automatic Translation of UML Sequence Diagrams into PEPA Models

Automatic Translation of UML Sequence Diagrams into PEPA Models

Mirco Tribastone and Stephen Gilmore
The University of Edinburgh, Scotland

Abstract

The UML profile for Modeling and Analysis of Real Time and Embedded systems (MARTE) provides a powerful, standardised framework for the specification of non-functional properties of UML models. In this paper we present an automatic procedure to derive PEPA process algebra models from sequence diagrams (SD) to carry out quantitative evaluation. PEPA has recently been enriched with a fluid-flow semantics facilitating the analysis of models of a scale and complexity which would defeat Markovian analysis.

1. Introduction

Over the past few years software models have emerged as a promising approach to the design and development of complex software systems. Particularly, they play a crucial role in model-centric engineering methodologies, in which systems are designed in a platform-independent domain and executable program code is generated by automatic tools. Because models are employed throughout the development lifecycle, quantitative evaluation of these models has great appeal because it can be applied at any stage of the process to give valuable feedback for the next revision of the system.

Intimately linked to model-driven development technologies is the UML, a rich graphical language for the design of software systems (mainly object-oriented systems). The UML consists of a number of *views* which allow a system to be represented from different perspectives. The *behavioural* (or *dynamic*) view offers the appropriate context for the purposes of performance evaluation because it captures how components communicate with each other and react to events. Among the behavioural diagrams are *sequence diagrams* (SDs), in which the interaction between objects is modelled by means of ordered exchanges of messages between them. It must be pointed out though that the order is only relative and there is no formal way in the standard UML to assert timing properties of messages. To overcome this difficulty, frameworks for quantitative evaluation of UML models may benefit from the profile for MARTE [10], which enriches the language with a number of extensions to define non-functional properties. However, MARTE serves only as a mechanism for annotating diagrams with timing information and does not offer any numerical evaluation technique. The integration of analysis methods into the framework is left to the users of the profile.

In this paper we use the stochastic process algebra PEPA [6] as a performance engine for MARTE. We automatically translate SDs into PEPA descriptions. Each interaction lifeline is mapped onto a sequential automaton and messages sent between these are interpreted as synchronisation points (in process algebraic terms, *shared actions*) between sequential components. Since UML 2.0, SDs can express *rich* sequence traces, incorporating parallel fragments and alternative fragments. We use distinct sequential components to model the behaviour of a lifeline involved in a parallel fragment. Each component synchronises over shared actions enabled when the parallel fragment is entered or exited. Alternative fragments are mapped onto probabilistic choices. This approach is based on the classical compositional modelling paradigm of process algebra. Compositionality does not necessarily imply *parallelism*; an intrinsically sequential underlying model can be derived via a compositional translation technique. Nevertheless, a compositional technique proves to be an elegant way to capture parallelism in the interaction—either *implicit*, by events that are not related by a chain of intermediate messages, or *explicit*, by the use of parallel constructs.

The major motivation behind the use of PEPA in the software engineering process is that it provides a means to address the state space explosion problem which afflicts Markovian analysis techniques. Originally designed as a stochastic process algebra, PEPA has been recently provided with a continuous-space fluid-flow semantics which gives rise to a system of first-order ordinary differential equations [7]. This represents an effective way to deal with complex software design models of large-scale systems. In particular, the translation procedure from UML SDs to PEPA which is described in the present paper maps replicated software resources onto *arrays* of sequential components, i.e. a parallel composition of equally-behaving components. This takes full advantage of the scalability of the fluid-flow approach.

The remainder of this paper is organised as follows. Related work on the performance analysis of UML models is reviewed in Section 2. Section 3 provides an overview of PEPA and introduces some preliminaries which will be used throughout the paper. Sequence diagrams and their timing extensions are discussed in Section 4. Section 5 discusses the extraction of PEPA descriptions from SDs. Finally, Section 7 concludes the paper.

2. Related Work

Over the last decade a substantial body of research has focussed on quantitative analysis of software models. For a survey of this field see [1]. In particular, since the advent of the UML, much effort has been devoted to the extraction of performance models from the elements of the language concerned with the dynamic description of software systems, such as activity diagrams, state machine diagrams, and SDs. A taxonomy of the research in this context may be based on four orthogonal factors: (1) the kind of diagrams from which the performance model is extracted; (2) the level of automation supported by the translation; (3) the methodology used to annotate the diagram with non-functional properties; and (4) the underlying analysis technique employed.

For instance, Layered Queueing Networks are used to enable performance analysis of UML specifications [11]. In [8] activity diagrams are translated into Stochastic Petri Nets. Both approaches infer the performance model from annotations with the profile for Schedulability, Performance, and Time.In the former the authors propose a direct translation. In the latter the translation is mediated by graph-transformation rules between the UML and the performance meta-models which are then performed at the XML level.

A similar approach toward the formalisation and generalisation of transformation techniques is proposed in [15, 12] where an intermediate representation—the Core Scenario Model (CSM)—abstracts away the UML elements which are not related to performance evaluation and acts as a simplified input meta-model for conversion into different target performance models. In the present paper we put forward a direct mapping from MARTE-annotated SDs onto performance models, although a mediation through CSM is possible in principle. As far as parallel and alternative fragments are concerned, the underpinnings of the translation are close to those presented in [15, 12, 13]. However, the interpretation of the synchronicity of the messages is based on a different approach. In our case, a synchronous message is modelled with a pair of asynchronous messages, between which the sender is blocked. This is complementary to the approach presented in the aforementioned works, in which asynchronous messages are modelled using an implicit fork at the sender's site.

The use of PEPA as the underlying formalism for the performance evaluation of UML models has been around for a while. In [3] a combination of state machine diagrams and collaboration diagrams is used to automatically extract PEPA models. In [2] the authors present a translation of activity diagrams into PEPA-Net models, though it is tailored to a specific case study and does not make use of standard performance annotation techniques. Those issues are addressed in [14], where an automatic algorithm for the derivation of PEPA models from activity diagrams is presented. It shares similar techniques to those presented in this paper, including the use of the MARTE profile for the specification of the non-functional properties of the system and the performance indices of interest.

3. Overview of PEPA

PEPA is a timed process algebra for the performance evaluation of models described using a parsimonious grammar. We consider PEPA models which can be defined using the following two-level grammar:

$$\begin{array}{rcl} S & := & (\alpha, r).S \,|\, S + S \,|\, A \\ C & := & S \,|\, C \underset{L}{\bowtie} C \,|\, C/L \end{array}$$

The first production is used to define *sequential components*, whereas the second allows composition of components. In this section an informal description of the language is presented. The interested reader should consult [6] for the formal definition.

Prefix $(\alpha, r).S$ is the most basic unit of execution of the language. It describes a component performing an activity of type α at rate r. When the activity completes, the component behaves as S. An exponential distribution of the delay is assumed.

Choice $P + Q$ indicates probabilistic choice among the activities of the sequential components P and Q.

Constant $A \overset{def}{=} S$ defines a component A which behaves as S.

Cooperation $P \underset{L}{\bowtie} Q$ allows composition. P and Q carry out their activities concurrently if the type of the activity is not in the cooperation set L. If the activity's type is in L, they perform a *shared action* at a rate which depends on the rates of the individual components involved in the cooperation. A rate of a shared activity may be left as unspecified at a particular sequential component by using the symbol $\top$. This signifies that the shared rate is specified by other components.

We will not use PEPA's hiding operator in this paper.

We shall be concerned with PEPA descriptions consisting of an arbitrary number of productions for sequential components and only one production for the composition of such components. Such a production shall be referred to as the *system equation*. Without loss of generality, we shall deal with PEPA descriptions which have at least two distinct sequential components cooperating over a (possibly empty) cooperation set. As discussed later in this paper, each lifeline of a SD will be associated to at least one sequential component. Hence, a PEPA description with only one sequential component would model an interaction with one single lifeline, which is of little practical interest. System equations with at least two sequential components are binary trees, whose internal nodes are cooperation operators, labelled by their cooperation sets. The external nodes represent sequential components, hence they may represent prefixes, choices, or component names (constants). However, the algorithm that performs the automatic translation of SDs will create trees with leaves which contain only component names. The interpretation of a cooperation as a binary tree will be used later to describe how a model description is manipulated during the traversal of the SD's interaction fragments.

In the Markovian semantics [6], PEPA descriptions are interpreted against an operational semantics which results in a labelled transition system whose states are PEPA components and transition labels are the *(type, rate)* pairs of the activities enabled by that state. A Continuous Time Markov Chain (CTMC) can be derived from the labelled transition system by associating each state of the system with a state of the Markov process. The generator matrix is extracted from the rates in the transition labels. The solution of this underlying CTMC ultimately allows for the performance evaluation of the system. Analysis techniques available in this context include transient analysis, steady-state analysis, and model checking.

As for the fluid-flow approximation [7], an algorithm is provided to automatically generate a system of ODEs from a PEPA model. The cost of the algorithm is low, as it traverses the model description statically, i.e. without generating the state space of the Markov chain. The comparison between the execution times of Markovian and time-series analyses presented in the paper has motivated us to exploit this computationally inexpensive technique in the software performance engineering context. Nonetheless, the use of PEPA serves other important purposes. Small or medium-sized performance models may still be suitable for Markovian analysis, providing software engineers with a wider range of analysis techniques at their disposal. PEPA plays a crucial role as an *intermediate language* here by making a wide range of mathematical tools available without impacting on the methodology for the extraction of the performance model from the design model. In addition, the formal semantics of the PEPA language makes it possible to reason about the model prior to the execution of the quantitative analysis. For instance, static analysis has been developed to check for freedom from deadlock and the absence of transient states or absorbing states.

4. Sequence Diagrams

A SD is the graphical representation of an *interaction* between a set of participants, i.e. *lifelines*, with emphasis on the sequence of messages exchanged. In this section are discussed the elements of interest in the performance evaluation context. For each element, the relevant performance annotations are discussed. For a complete and formal specification of UML Interaction, the reader is referred to [9].

Sequence Diagram A SD representing the behaviour to be analysed is stereotyped with *GaScenario*. Among its properties, *cause* references the applied workload, which is stereotyped with *GaWorkloadEvent*. Although MARTE supports the specification of various kinds of workload patterns, in this paper we shall be concerned with *closed patterns*. A closed pattern consists of a population of users which cyclically execute the behaviour. A delay, which we refer to as *thinking time*, is observed between successive requests. Adhering to a common practise for workload event annotation, we shall deal with SDs whose first message is stereotyped with *GaWorkloadEvent*. Let $\mathcal{W} = \{\omega_1, \omega_2, \ldots, \omega_n\}$ be the set of actions enabled by the PEPA component underlying the first interaction fragment, and N_W and r the values of the properties *population* and *extDelay*, respectively. The corresponding PEPA process is as follows:

$$\begin{aligned} \mathit{Thinking} &\overset{\mathit{def}}{=} (\mathit{think}, r).\mathit{Requesting} \\ \mathit{Requesting} &\overset{\mathit{def}}{=} (\omega_1, \top).\mathit{Thinking} \\ &+ (\omega_2, \top).\mathit{Thinking} \\ &+ \ldots \\ &+ (\omega_n, \top).\mathit{Thinking} \end{aligned} \tag{1}$$

Let *Interaction* be the PEPA process modelling the entire interaction. The workload is composed with *Interaction* as follows:

$$\mathit{Thinking}[N_W] \underset{\mathcal{W}}{\bowtie} \mathit{Interaction} \tag{2}$$

For a sequential component P, we use the notation $P[N]$—which we refer to as an *array of components*—to indicate a parallel composition of N copies of the component P. The parameter N may have a significant impact on the computational cost of the quantitative evaluation. If Markovian analysis is to be carried out, the state space size grows combinatorially with N, although there exists an efficient algorithm [5] based on an equivalence relation called *isomorphism* which results in a smaller Markov process. Conversely, its influence is almost negligible if fluid-flow approximation is used, since it is an invariant with respect to the structure of the system of ODEs underlying the PEPA description.

Lifelines A lifeline is a participant in an interaction. As in [10][1] we consider lifelines annotated with the *PaRunTInstance*, representing a run-time instance of a process. In the remainder of this paper we shall be concerned with models where deployment information is missing, a situation which may be encountered during early stages of the development lifecycle. In such a case, resource contention reduces to pure delay, as it is assumed that there exists an infinite pool of processors onto which processes are executed. (This assumption is discussed in [15]. A similar approach is taken in [8, 14].)

Of particular relevance is the property *poolSize*, an integer denoting the number of threads of the process. In the performance model it indicates the size of the array of the sequential components modelling the lifeline. As with the population size of the workload, the same arguments about the scalability of the fluid-flow approach hold for the thread pool size.

Messages A message models a communication in an interaction. The UML offers a rich semantics for messages, though for the purposes of quantitative analysis some of the differences between the kind of messages can be abstracted away. In particular, a message models either an Operation call or a Signal, though essentially they both represent a means of passing information. The nature of the message is not distinguished at the level of the performance model. The two attributes of a Message are *messageKind* and *messageSort*. As for the former, we consider *complete* messages, i.e. messages in which *sendEvent* and *receiveEvent* are present. Traversing the model through those properties allows the identification of the sender and the receiving lifelines. These are indeed the kind of messages that are used, for instance, in [10, 15, 12]. The latter property conveys the information about the synchronicity of the message, and it clearly has a profound impact on the performance model. Although return messages from synchronous calls may be omitted to reduce clutter, it is a conventional practise to show returns. In the remainder of this paper we assume that diagrams have return messages. We would like to point out that this is a very mild limitation, as the end of a call can be determined by inspection of its activation region should the return be omitted.

Unlike [10] we adopt a slightly different interpretation of the stereotypes applied to messages. In the formal specification of MARTE, a *PaStep* on the message denotes the demand on the receiving lifeline up until the end of the execution specification. If the lifeline is involved in nested operations, their demands are added in order to determine the overall duration of the operation. To indicate a duration of the message, the stereotype *PaCommStep* is used instead. We denote such a duration with *PaStep*; this stereotype application is required on all messages, as it is modelled as a timed shared action in PEPA. To indicate an exponentially distributed duration of a message with mean rate r times per second, the *PaStep* would have its *hostDemand* property set to `(exp(1/r),s)`. Notice that if the performance model is interpreted against the ODE-based semantics, then the mean duration is considered.

ExecutionSpecification For the purposes of performance modelling, no distinction is made between ActionExecutionSpecification and BehaviorExecutionSpecification as they are both regarded as opaque units of execution. We shall refer to either with their shared interface, ExecutionSpecification. To indicate the duration of an ExecutionSpecification, we directly apply *PaStep* to it. However, such applications may be omitted to model an operation whose demand is negligible with respect to the other durations involved in the interaction. This is an alternative representation for message annotated with MARTE, though the procedure for the automatic translation may seamlessly accommodate different interpretations.

Combined Fragments Combined fragments were introduced in UML 2.0 to add more expressiveness to Interactions by means of constructs for capturing flow-of-control. The notation for a fragment is a rectangle with a keyword in the top-left corner indicating the kind of construct that it represents. We support **alt**, **loop**, and **par** combined fragments. It is required that each supported combined fragment have a *PaStep* applied.

The keyword **alt** indicates an alternative between two or more choices, indicated as sub-fragments separated by horizontal lines. Alternative paths will be modelled as PEPA choices. The annotation indicates the delay for the decision of following one path (through the *hostDemand* property). Each operand must be annotated with a *PaStep* to denote the probability with which the path is taken (*prob* property).

The keyword **loop** designates an iteration over its operand. The number of iterations is obtained through the *rep* property of its operand.

Finally, **par** indicates a parallel behaviour between its operands. The parallel sequences implicitly synchronise over two events—they are started when the fragment is executed, and they are stopped when the fragment is exited. The value of *hostDemand* characterises the duration of the synchronisation.

[1]See, for instance, the examples in Sect. 17.4.

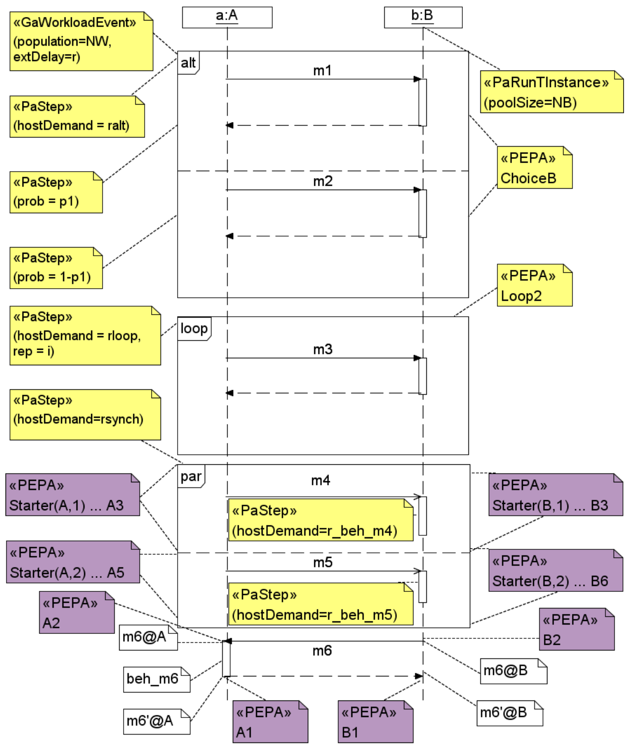

«GaWorkloadEvent» (population=NW, extDelay=r)
a:A
b:B
alt
m1
«PaRunTInstance» (poolSize=NB)
«PaStep» (hostDemand = ralt)
«PaStep» (prob = p1)
m2
«PEPA» ChoiceB
«PaStep» (prob = 1-p1)
«PEPA» Loop2
«PaStep» (hostDemand = rloop, rep = i)
loop
m3
«PaStep» (hostDemand=rsynch)
par
m4
«PEPA» Starter(A,1) ... A3
«PaStep» (hostDemand=r_beh_m4)
«PEPA» Starter(B,1) ... B3
«PEPA» Starter(A,2) ... A5
m5
«PEPA» Starter(B,2) ... B6
«PEPA» A2
«PaStep» (hostDemand=r_beh_m5)
«PEPA» B2
m6@A
m6
beh_m6
m6@B
m6'@A
«PEPA» A1
«PEPA» B1
m6'@B

Figure 1. Sample SD supported by the automatic procedure.

4.1. Example

Figure 1 shows a sample SD amenable to automatic translation into a PEPA model. According to the standard UML notation, asynchronous messages are drawn with an open arrowhead, whereas synchronous messages use a filled arrowhead. To reduce clutter, the labels for the replies to the synchronous calls (i.e., m_1, m_2, m_3, and m_6) are not shown. Let m'_i be the label of the reply to call m_i. Let req_{m_i} be the rate of synchronous calls and asynchronous calls with label m_i, and rep_{m_i} be the rate of the synchronous reply to message m_i. All the message are assumed to have MARTE annotations, not shown here for the sake of readability. The annotation for lifeline A is similar to that of lifeline B (the *poolSize* property is set to N_A). Unless explicitly annotated in the diagram, execution specifications take no time. The diagrams shows a number of comments with the keyword *PEPA* attached to the interaction's fragments. Such comments provide a visual mapping between representative UML elements and their corresponding terms in the performance model, and they will be discussed throughout the description of the translation procedure in Sect. 5. Due to space constraints, the interaction shows only two lifelines. However, the algorithm is general and supports an arbitrary number of lifelines. We would like to note that the algorithm also allows nesting of combined fragments. However, to avoid unnecessary intricacies during its walk-through, nested fragments are not presented in the example.

5. Automatic Extraction of the PEPA Model

The rationale behind the translation algorithm is to model each participating lifeline as one sequential component. If the lifeline is involved in a **par** fragment, a sequential component is used for each lifeline and for each operand. All the sequential components of a **par** fragment synchronise over shared actions, in order to make them start when the parallel fragment is entered, and stop when they finish their execution.

An **alt** fragment is interpreted stochastically—a timed decision-making activity is assigned to the fragment, and each operand has an execution probability. In PEPA, this is modelled with the choice operator. A similar treatment is given to a **loop** fragment, the ratio between executing an iteration and exiting the loop is governed by the annotated number of iterations, normalised by the rate associated to the fragment. For instance, a loop with N iterations may be modelled by *unrolling* the process definitions of the loop N times. Alternative options for **loop** are not further discussed in this paper.

A MessageOccurrenceSpecification is used to create a shared action type between the communicating lifelines. In PEPA, the individual rates involved in a shared activity may be either active or passive—at least one rate must be active, however. In this context, a shared activity for a message will always involve two components. We assign the active rate to the sequential component of the sending lifeline, whereas the component of the receiving lifeline is made passive. This is just a convention—an equivalent performance model would be obtained if the rates were swapped because the composition of the two components would evolve with the same overall rate.

5.1. Methodology

In the UML meta-model an interaction is defined in terms of interaction fragments. Fragments are defined recursively, i.e. multiple levels of nesting are allowed. The strategy for the visit of a SD is a depth-first traversal of an interaction fragment. The operands of a combined fragment are visited in the same order as they are drawn in the diagram. There are fragments, such as InteractionOperand, with a list-valued attribute of ordered fragments. When such a fragment is encountered, the elements of the list are visited in reverse order. As a result, if two message occurrence specifications are connected by a chain of events (e.g., one is the *send* event and the other is the *receive* event of the same message) the successor is traversed first. This visit strategy is recursive, and the termination condition is represented by the visit of basic fragments such as instances of ExecutionSpecification, MessageOccurrenceSpecification, or ExecutionOccurrenceSpecification. Whilst the first two convey performance-related information, as discussed above, an ExecutionOccurrenceSpecification is not significant here, and its visit does not alter the performance model. The SD in Fig. 1 has two instances of such a fragment, corresponding to the finish events of the behaviour execution specifications of m_4 and m_5.

The construction of the performance model is incremental, i.e. during the visit of a fragment some manipulations are carried out on the PEPA description. In the remainder of this section an algorithm is presented for each supported interaction fragment. All the algorithms take as input an array of PEPA constants. Each element of the array is a derivative of the sequential component of the lifeline, and it indicates the PEPA behaviour of the lifeline that comes after the visited fragment. This array will be referred to as the variable *targets* in the pseudo-code descriptions. We assume that the lifelines are indexed over integers $1 \ldots L$. A target of a lifeline can be accessed by a pointer to the lifeline or by the lifeline index, interchangeably. Finally, each algorithm will return an array of constants. Each constant refers to the behaviour of a lifeline when the visited fragment is executed.

5.2. Notation

In the pseudo-code which describes the translation procedure for each supported element, the variable *self* will be referred to as a pointer to the currently visited interaction fragment. We also assume that the tagged values of interest are accessible as fields of *self*. For instance, *self.rate* denotes the rate associated with the interaction fragment. Likewise, the navigation of the UML model is achieved through fields. For instance, in a message occurrence specification, *self.lifeline* will be the pointer to the lifeline which covers that occurrence specification. Finally, a number of helper functions will be used during the traversal. Details on their implementation are straightforward and are omitted due to lack of space.

addComponent(Constant[N]) Adds an array of sequential compoentns to the system equation of the PEPA model and composes it with the existing system equation. The set of the new cooperation is empty. The array represents a sub-tree of the system equation, whose root is added to a set denoted by $\mathcal{P}$.

createAction(InteractionFragment) Creates a PEPA action type uniquely associated with the given interaction fragment. It accepts instances of MessageOccurrenceSpecification and ExecutionSpecification.

createConstant(OccurrenceSpecification) Returns a new PEPA constant which uniquely identifies the behaviour of the occurrence specification.

isCovering(int, InteractionFragment) Returns true if the lifeline index by the integer is covering the interaction fragment.

participant(MessageOccurrenceSpecification) Returns the other lifeline involved in the communication related to the message occurrence specification.

update(Constant[], Lifeline, Constant) Replaces the current target of the lifeline with the constant supplied.

visitList(List, Constant[]) Visits an ordered list of InteractionFragments, according to Sect. 5.1.

5.3. Algorithms

We now describe the automatic translation procedure by means of pseudo-code descriptions. The procedure has three steps. The first step consists of a preliminary set-up of auxiliary data structures. Then, the UML Interaction representing the SD is traversed. During this step new definitions are added to the PEPA model and nodes are added to the system equation. Finally, the third step visits the system equation and manipulates its cooperation sets to enable synchronisation between the sequential components.

5.3.1 Initialisation

This phase will set up a skeletal system equation of the performance model by associating an array of components with each lifeline. The size of the array is extracted from the *poolSize* property of the lifeline. Initially, such arrays will be composed with empty action sets. As the diagram is visited, new sequential components are added to the system equation to reflect the presence of **par** fragments. The pseudo-code is shown in Algorithm 1. Line 8 starts off the translation by visiting the fragments of the interaction.

Algorithm 1 Initialisation

```
1:  $System \Leftarrow \emptyset$
2:  initTgt $\Leftarrow$ new Constant[$L$]
3:  for $i = 1$ to $L$ do
4:     $Const \Leftarrow$ createConstant(lifeline$_i$)
5:     addComponent($Const$[lifeline$_i$.poolSize])
6:     initTgt[$i$] $\Leftarrow$ $Const$
7:  end for
8:  sdTgt $\Leftarrow$ visitList(interaction.fragments, initTgt)
9:  for $i = 1$ to $L$ do
10:    initTgt[i] $\overset{def}{=}$ sdTgt[i]
11: end for
```

5.3.2 MessageOccurrenceSpecification

The translation for a MessageOccurrenceSpecification is shown in Algorithm 2. Notice that this algorithm is also applied for messages whose occurrence specifications are on the same lifeline. However, in this case the delay may be omitted (line 4). If present, it is modelled as an unshared (individual) action.

Algorithm 2 MessageOccurrenceSpecification

```
1: $action \Leftarrow$ createAction(self)
2: rate $\Leftarrow$ self.rate
3: if partner = self then
4:     if rate = null then
5:         return targets
6:     end if
7: else
8:     if self.event is a receive event then
9:         rate $\Leftarrow \top$
10:    end if
11: end if
12: $LocalState \Leftarrow$ createConstant(self)
13: $LocalState \overset{def}{=} (action, \text{rate}).targets[self.lifeline]$
14: return update(targets, self.lifeline, $LocalState$)
```

5.3.3 ExecutionSpecification

Algorithm 3 shows the translation of an execution specification into an independent activity performed by the sequential component of the involved lifeline. Performance annotation of execution specifications is not mandatory (line 1).

Algorithm 3 ExecutionSpecification

```
1: if self.rate = null then
2:     return targets
3: end if
4: $LocalState \Leftarrow$ createConstant(self)
5: $action \Leftarrow$ createAction(self)
6: $LocalState \overset{def}{=} (action, \text{self.rate}).targets[self.lifeline]$
7: return update(targets, self.lifeline, $LocalState$)
```

5.3.4 Combined Fragments

The pseudo-code for the translation of a **par** fragment is shown in Algorithm 4. The visit of each operand will return the initial local states of the lifelines when that parallel branch is executed (line 15). Such states are prefixed with a *start* activity for them to synchronise when the parallel fragment is entered (line 18). According to the semantics of SDs, all the executions within a parallel fragment must be finished before the fragment's successor can start its execution. Here, a shared *stop* activity serves as a synchronisation point for all the parallel branches (lines 8,20). Here we assume symmetric synchronisation delays, although alternative situations can be captured through proper stereotype annotations.

Algorithm 4 Parallel Fragment (**par**)

```
1: parTgt $\Leftarrow$ targets
2: for operand in self.operands do
3:     opTgt $\Leftarrow$ new Constant[$L$]
4:     for $i = 1$ to $L$ do
5:         if isCovering(i, self) then
6:             opTgt[i] = new $Constant_{i,self}$
7:             if first operand then
8:                 opTgt[i] $\overset{def}{=} (stop_{self}, \text{self.rate})$.targets[i]
9:                 parTgt[i] = opTgt[i]
10:            end if
11:        else
12:            opTgt[i] = targets[i]
13:        end if
14:    end for
15:    opTgt $\Leftarrow$ visitList(operand.fragments, opTgt)
16:    for $j = 1$ to $L$ do
17:        if isCovering(j, self) then
18:            $Starter_{j,self} \overset{def}{=} (start_{self}, \text{self.rate})$.opTgt[j]
19:            if not first operand then
20:                opTgt[j] $\overset{def}{=} (stop_{self}, \text{self.rate}).Starter_{j,self}$
21:                $N_C \Leftarrow$ lifeline$_j$.poolSize
22:                addComponent($Starter_{j,self}[N_C]$)
23:            end if
24:            opTgt[j] $\Leftarrow Starter_{j,self}$
25:        end if
26:    end for
27: end for
28: return parTgt
```

A **loop** fragment is translated into PEPA as shown in Algorithm 5.

Algorithm 6 shows the pseudocode to translate an **alt** fragment into a nondeterministic choice between the alternative operands. The visit of one operand will return the initial local states of the PEPA components when that path is taken (*opTgt*, line 4). Hence, after all the operands are visited, for each lifeline there is a list of all its alternative behaviours. The elements of such a list are composed through the PEPA choice operator, for each lifeline (*choices*, line 2). To model the decision-taking process at the **alt** fragment, each element is prefixed with an activity (line 14) whose duration reflects the probability with which that path is chosen.

Algorithm 5 Loop fragment (**loop**)

```
loopTgt ⇐ **new** Constant[$L$]
**for** $i = 1$ to $L$ **do**
    **if** isCovering(i, self) **then**
        loopTgt[i] ⇐ $Loop_{self,i}$
    **else**
        loopTgt[i] ⇐ targets[i]
    **end if**
**end for**
opTgt ⇐ visitList(self.operand.fragments, loopTgt)
firstLifeline ⇐ **true**
rep ⇐ self.operand.rep
**for** $i = 1$ to $L$ **do**
    **if** isCovering(i, self) **then**
        **if** firstLifeline **then**
            loopRate = $\frac{\text{rep}}{\text{rep}+1} \cdot$ self.rate
            exitRate = $\frac{1}{\text{rep}+1} \cdot$ self.rate
            firstLifeline ⇐ **false**
        **else**
            loopRate = exitRate = $\top$
        **end if**
        $Loop_{self,i} \overset{def}{=} (loop_{self}, loopRate)$.opTgt[i] + $(exit_{self}, exitRate)$.targets[i]
    **end if**
**end for**
**return** loopTgt
```

Algorithm 6 Alternative Fragment (**alt**)

```
choiceTgt ⇐ **new** Constant[$L$]
choices ⇐ **new** Choice[$L$]
**for** operand in self.operands **do**
    opTgt ⇐ visitList(operand.fragments, targets)
    firstLifeline ⇐ **true**
    **for** $i = 1$ to $L$ **do**
        **if** isCovering($i$, self) **then**
            **if** firstLifeline **then**
                rate = operand.prob $\cdot$ self.rate
                firstLifeline ⇐ **false**
            **else**
                rate ⇐ $\top$
            **end if**
            prefix ⇐ $(path_{self,operand}$, rate).opTgt[i]
            choices[i].addOperand(prefix)
        **end if**
    **end for**
**end for**
**for** $i = 1$ to $N$ **do**
    **if** isCovering(i, self) **then**
        $Choice_{i,self} \overset{def}{=} choices[i]$
        choiceTgt[i] ⇐ $Choice_{i,self}$
    **else**
        choiceTgt[i] ⇐ targets[i]
    **end if**
**end for**
**return** choiceTgt
```

5.3.5 Post-visit

Algorithm 7 is applied to update the cooperation sets of the system equation. For each visited node, it adds to the cooperation set all the actions that are enabled by both children. Here, $\overrightarrow{\mathcal{A}}(C)$ the set of all the actions enabled by C and its derivatives [6]. The algorithm is defined recursively and is started by $update(System)$. The recursion terminates with the visit of an array of sequential components in $\mathcal{P}$.

5.4. Walkthrough Example

As a practical application, let us consider the construction of the performance model of the SD in Fig. 1. Let $[A_0, B_0]$ the initial array of targets as computed upon initialisation. Let *System* be the constant that defines the system equation.

$$System \overset{def}{=} A_0[N_A] \parallel B_0[N_B] \qquad (3)$$

The order of the traversal of the interaction is: 1) m'_6@B; 2) m'_6@A; 3) beh_{m_6}; 4) m_6@A; 5) m_6@B; 6) **par** fragment; 5) **loop** fragment; 6) **alt** fragment. Each top-level fragment is examined in a separate paragraph. For ease of reference, the values of the array of *targets* are also shown.

$m'_6@B$, **targets** $= [A_0, B_0]$ By Algorithm 2 the new process definition $B_1 \overset{def}{=} (m'_6, \top).B_0$ is added.

$m'_6@A$, **targets** $= [A_0, B_1]$ We create $A_1 \overset{def}{=} (m'_6, rep_{m_6}).A_0$. The rate is active because A is sending the message.

beh_{m_6}, **targets** $= [A_1, B_1]$ This behaviour specification is not stereotyped. Hence, the visit simply returns the targets that were supplied as input.

m_6, **targets** $= [A_1, B_1]$ With similar arguments to m'_6, we add $A_2 \overset{def}{=} (m_6, \top).A_1$ and $B_2 \overset{def}{=} (m_6, req_{m_6}).B_1$.

par fragment, targets $= [A_2, B_2]$ Let $start_1$, $stop_1$ be the synchronisation actions of the behaviour, both performed with rates r_{sync}. Before visiting the operands, the targets to be

Algorithm 7 Function $update(n)$

```
**if** $n \notin \mathcal{P}$ **then**
    $\overrightarrow{\mathcal{A}}_l \Leftarrow update(n.\text{left})$
    $\overrightarrow{\mathcal{A}}_r \Leftarrow update(n.\text{right})$
    $n.\text{set} \Leftarrow \overrightarrow{\mathcal{A}}_l \cap \overrightarrow{\mathcal{A}}_r$
**end if**
**return** $\overrightarrow{\mathcal{A}}(n)$
```

passed are prepared (lines 2–14). If a lifeline does not cover the fragment, there is no manipulation of targets. Else, new constants are created. For the first operand, the targets are $[A_3, B_3]$, $A_3 \overset{def}{=} (stop_1, r_{sync}).A_2$, $B_3 \overset{def}{=} (stop_1, r_{synch}).B_2$. The visit of the first operand generates one process definition for lifeline A, $A_4 \overset{def}{=} (m_4, req_{m_4}).A_3$. Two definitions are generated for lifeline B: $B_4 \overset{def}{=} (beh_{m_4}, r_{beh_{m_4}}).B_3$ and $B_5 \overset{def}{=} (m_4, \top).B_4$. (Here, the behaviour specification is translated into an independent activity because it is stereotyped.) The targets returned by the first operand are $[A_4, B_5]$. The *start* activity is now prefixed to those targets (line 21): $Starter_{A,1} \overset{def}{=} (start_1, r_{sync}).A_4$ and $Starter_{B,1} \overset{def}{=} (start_1, r_{sync}).B_5$.

The second operand is passed *placeholder* targets $[A_5, B_6]$, which will be defined after the operand is visited. Similarly to the first operand, the visit of the fragments will result in following two new definitions for lifeline B and one for lifeline A: $B_7 \overset{def}{=} (beh_{m_5}, r_{beh_{m_5}}).B_6$, $B_8 \overset{def}{=} (m_5, \top).B_7$; $A_6 \overset{def}{=} (m_5, req_{m_5}).A_5$. Thus, the array returned by the second operand is $[A_6, B_8]$. Now, the start activity is prefixed to such targets: $Starter_{A,2} \overset{def}{=} (start_1, r_{sync}).A_6$, $Starter_{B,2} \overset{def}{=} (start_1, r_{sync}).B_8$. Then, the placeholder targets are defined: $A_5 \overset{def}{=} (stop_1, r_{sync}).Starter_{A,2}$, $B_6 \overset{def}{=} (stop_1, r_{sync}).Starter_{B,2}$. The system is updated thus:

$$System \overset{def}{=} \Big(\big(A_0[N_A] \underset{\mathcal{L}}{\bowtie} B_0[N_B]\big) \underset{\mathcal{M}}{\bowtie} Starter_{A,2}[N_A] \Big) \underset{\mathcal{K}}{\bowtie} Starter_{B,2} \tag{4}$$

The visit returns the targets of the first operand $[Starter_{A,1}, Starter_{B,1}]$.

loop fragment, targets = $[Starter_{A,1}, Starter_{B,1}]$ Both lifelines are involved in the loop, hence lt $= [Loop_1, Loop_2]$ (line 1–8). Line 9 adds two definitions for each lifeline: $A_7 \overset{def}{=} (m'_3, \top).Loop_1$, $A_8 \overset{def}{=} (m_3, req_{m_3}).A_7$; $B_9 \overset{def}{=} (m'_3, rep_{m_3}).Loop_2$, $B_{10} \overset{def}{=} (m_3, \top).B_9$. The targets returned are $[A_8, B_{10}]$. Lines 12–22 define the *Loop* processes: $Loop_1 \overset{def}{=} (loop, \frac{i}{i+1} r_{loop}).A_8 + (exit, \frac{1}{i+1} r_{loop}).Starter_{A,1}$; for the second lifeline, rates are passive, i.e. $Loop_2 \overset{def}{=} (loop, \top).B_{10} + (exit, \top).Starter_{B,1}$.

alt fragment, targets = $[Loop_1, Loop_2]$ The visit of the first operand generates: $A_9 \overset{def}{=} (m'_1, \top).Loop_1$, $A_{10} \overset{def}{=} (m_1, req_{m_1}).A_9$; $B_{11} \overset{def}{=} (m'_1, rep_{m_1}).Loop_2$, $B_{12} \overset{def}{=} (m_1, \top).B_{11}$; The array returned is $[A_{10}, B_{12}]$. Let $Choice_A$ be the choice operator for lifeline A. Its temporary definition is: $Choice_A \overset{def}{=} (path_1, p_1 r_{alt}).A_{10}$. Similarly, $Choice_B \overset{def}{=} (path_1, \top).B_{12}$. The visit of the first operand generates: $A_{11} \overset{def}{=} (m'_2, \top).Loop_1$, $A_{12} \overset{def}{=} (m_2, req_{m_2}).A_{11}$; $B_{13} \overset{def}{=} (m'_2, rep_{m_2}).Loop_2$, $B_{14} \overset{def}{=} (m_2, \top).B_{13}$. The array returned is $[A_{12}, B_{14}]$. $Choice_A$ is added a new operand for the second path, i.e. $(path_2, (1 - p_1) r_{alt}).A_{12}$. Similarly, $(path_2, \top).B_{14}$ is added to $Choice_B$. This fragment returns $[Choice_A, Choice_B]$. Finally, the visit of the SD is completed with $A_0 \overset{def}{=} Choice_A$ and $B_0 \overset{def}{=} Choice_B$.

Post-visit and workload The cooperation sets (cfr. Eq. 4) are modified as follows: $\mathcal{L} = \{path_1, path_2, loop, exit, start_1, stop_1, m_4\} \cup \{m_i, m'_i : i \in \{1, 2, 3, 6\}\}$; $\mathcal{M} = \mathcal{K} = \{start_1, m_5, stop_1\}$. *System* is composed with the workload over $\mathcal{W} = \{path_1, path_2\}$ (cfr. Eq. 1). The complete model is presented in Appendix A.

6. Numerical Results

In this section we illustrate a typical workflow to which a PEPA model of a SD may be subjected to carry out quantitative analysis. We consider the model from Figure 1 with the parameter set below.

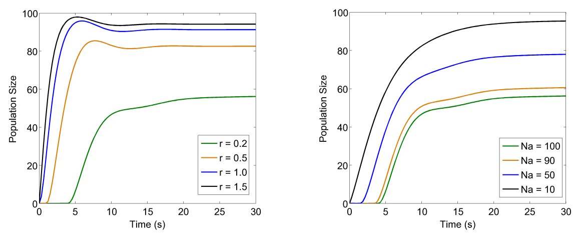
Rates and probabilities		Population sizes	
p_1	0.9	N_W	100
i	3	N_A	100
r	0.2	N_B	100
req_{m_1}	10.0		
req_{m_2}	2.0		

The parameters not shown in the table were set to 1.0. The table below shows the exponential growth of the state space size of the aggregated Markov chain with different settings for the number of copies of the system's processes. Clearly, Markovian analysis would not be viable with the parameter set shown above.

N_W	N_A	N_B	*Size*
1	1	1	40
4	1	1	100
4	2	2	1005
5	4	4	43656

Conversely, the interpretation against the fluid-flow semantics produces a system of 37 ODEs regardless of the population sizes. Such a system can be integrated numerically, resulting in a time series of a (continuous) population level for each model component. This has a clear interpretation with regard to the original UML model, as a PEPA component represents a particular occurrence on a sequence diagram's lifeline. Thus, at each point, the time series shows the mean number of elements of the thread pool that are witnessing a particular event.

ODEs provide a scalable means to sensitivity analysis, i.e. the study of the impact of a parameter on the overall system performance. For instance, the local state *Requesting* of the workload component is interpreted as the state in which



(a) (b)

Figure 2. Sensitivity analysis.

users are waiting for service. Thus, the higher this number, the less responsive the system is perceived to be.

Figure 2(a) shows how the performance is affected by changes in the users' *thinking time*. This analysis gives insights into the transient as well as steady-state behaviour of the system. For instance, the curve for $r = 0.2$ starts to rise at $t \sim 5\,s$. A slow start may also be noticed in the other curves, although less noticeably. This is due to how the workload is modelled: At $t = 0$, all the users start off by executing the *think* action, hence the system is idle. This is followed by a sharp increase, when the users trigger the execution of the sequence. In the steady-steady, the utilisation of the system clearly increases with faster thinking times.

Another class of parameters of interest is the thread pool size of the system's processes. In real-world situations it is usually necessary to make trade-offs between the cost of deploying concurrent resources (in terms of CPU or memory, for instance) and the relative gain from doing so. The diagram in this paper exhibits a high degree of cooperation between the two participating lifelines. Thus, the system cannot behave satisfactorily if the thread pool sizes of the two components are not well balanced. Figure 2(b) shows sensitivity analysis performed by varying the thread pool size of lifeline A. As expected the performance deteriorates as N_A decreases.

7. Conclusion

In this paper we presented an automatic procedure to map UML sequence diagrams into PEPA. We exploit the capability of the calculus to be interpreted against both a Markovian semantics and a continuous-state semantics which leads to a system of ODEs. The procedure generates PEPA models which can take full advantage of the latter, though symmetries can also be exploited by aggregation techniques to produce a smaller underlying Markov process. The algorithm, here illustrated by means of a running example, has a proof-of-concept implementation available at `http://homepages.inf.ed.ac.uk/mtribast/uml`.

References

[1] S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Trans. Software Eng.*, 30(5):295–310, 2004.

[2] C. Canevet, S. Gilmore, J. Hillston, L. Kloul, and P. Stevens. Analysing UML 2.0 Activity Diagrams in the Software Performance Engineering Process. In Dujmovic et al. [4], pages 74–78.

[3] C. Canevet, S. Gilmore, J. Hillston, M. Prowse, and P. Stevens. Performance Modelling with UML and Stochastic Process Algebras. *IEE Proceedings: Computers and Digital Techniques*, 150(2):107–120, Mar. 2003.

[4] J. J. Dujmovic, V. A. F. Almeida, and D. Lea, editors. *Proceedings of the Fourth International Workshop on Software and Performance, WOSP 2004, Redwood Shores, California, USA, January 14-16, 2004*. ACM, 2004.

[5] S. Gilmore, J. Hillston, and M. Ribaudo. An efficient algorithm for aggregating PEPA models. *IEEE Transactions on Software Engineering*, 27(5):449–464, May 2001.

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

[7] 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, Sept. 2005. IEEE Computer Society Press.

[8] J. López-Grao, J. Merseguer, and J. Campos. From UML activity diagrams to Stochastic Petri nets: application to software performance engineering. In Dujmovic et al. [4], pages 25–36.

[9] Object Management Group. *UML 2.2.1 Superstructure Specification*. OMG, 2007. OMG document number formal/05-07-04.

[10] Object Management Group. *UML Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE). Beta 1*. OMG, 2007. OMG document number ptc/07-08-04.

[11] D. Petriu and H. Shen. Applying the UML performance profile: Graph grammar-based derivation of LQN models from UML specifications. In *TOOLS'02*, volume 2324 of *LNCS*, pages 159–177. Springer, 2002.

[12] D. Petriu and C. Woodside. An intermediate metamodel with scenarios and resources for generating performance models from UML designs. *Softw. Syst. Model.*, 6:163–184, 2007.

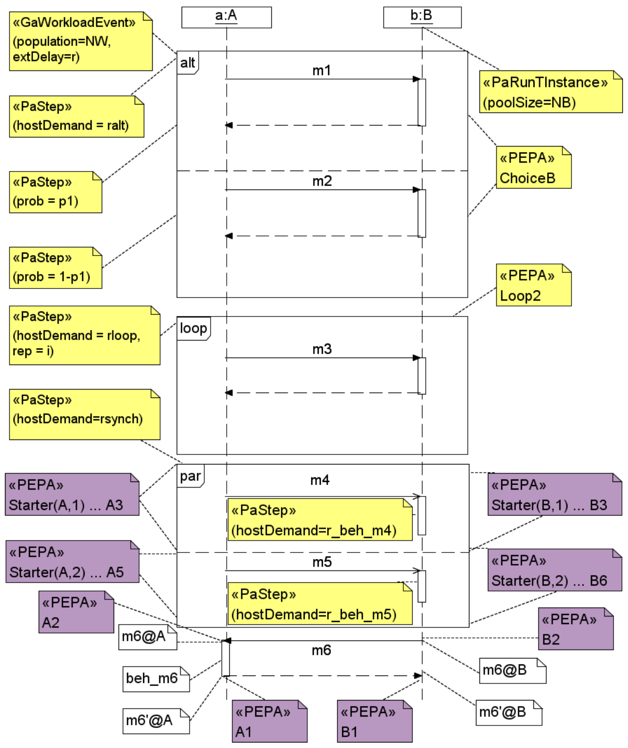
[13] D. Petriu, C. M. Woodside, D. Petriu, J. Xu, T. Israr, G. Georg, R. France, J. Bieman, S. Houmb, and J. Jurens. Performance Analysis of Security Aspects in UML Models. In V. Cortellessa, S. Uchitel, and D. Yankelevich, editors, *WOSP*, pages 78–89. ACM, 2007.

[14] M. Tribastone and S. Gilmore. Automatic Extraction of PEPA Performance Models from UML Activity Diagrams Annotated with the MARTE Profile. 2008. To appear in WOSP'08.

[15] C. Woodside, D. Petriu, D. Petriu, H. Shen, T. Israr, and J. Merseguer. Performance by unified model analysis (PUMA). In *WOSP*, pages 1–12. ACM, 2005.

A. The Complete PEPA Model

This appendix is included for the benefit of the reviewers and will be removed if the paper is accepted. We show here the complete PEPA model derived from this example sequence diagram, shown earlier.



a:A
b:B
«GaWorkloadEvent» (population=NW, extDelay=r)
alt
m1
«PaRunTInstance» (poolSize=NB)
«PaStep» (hostDemand = ralt)
«PaStep» (prob = p1)
m2
«PEPA» ChoiceB
«PaStep» (prob = 1-p1)
«PEPA» Loop2
«PaStep» (hostDemand = rloop, rep = i)
loop
m3
«PaStep» (hostDemand=rsynch)
par
m4
«PEPA» Starter(A,1) ... A3
«PaStep» (hostDemand=r_beh_m4)
«PEPA» Starter(B,1) ... B3
«PEPA» Starter(A,2) ... A5
m5
«PEPA» Starter(B,2) ... B6
«PaStep» (hostDemand=r_beh_m5)
«PEPA» A2
«PEPA» B2
m6@A
m6
beh_m6
m6@B
m6'@A
«PEPA» A1
«PEPA» B1
m6'@B

A.1. Sequential Components

A.1.1. Sequential Component A_0

$$
\begin{array}{rcl}
A_0 & \stackrel{def}{=} & Choice_A \\
Choice_A & \stackrel{def}{=} & (path_1, p_1 r_{alt}).A_{10} \\
 & + & (path_2, (1-p_1) r_{alt}).A_{12} \\
A_{10} & \stackrel{def}{=} & (m_1, req_{m_1}).A_9 \\
A_9 & \stackrel{def}{=} & (m_1', \top).Loop_1 \\
A_{12} & \stackrel{def}{=} & (m_2, req_{m_2}).A_{11} \\
A_{11} & \stackrel{def}{=} & (m_2', \top).Loop_1 \\
Loop_1 & \stackrel{def}{=} & (loop, \frac{i}{i+1} r_{loop}).A_8 \\
 & + & (exit, \frac{1}{i+1} r_{loop}).Starter_{A,1} \\
A_8 & \stackrel{def}{=} & (m_3, req_{m_3}).A_7 \\
A_7 & \stackrel{def}{=} & (m_3', \top).Loop_1 \\
Starter_{A,1} & \stackrel{def}{=} & (start_1, r_{sync}).A_4 \\
A_4 & \stackrel{def}{=} & (m_4, req_{m_4}).A_3 \\
A_3 & \stackrel{def}{=} & (stop_1, r_{sync}).A_2 \\
A_2 & \stackrel{def}{=} & (m_6, \top).A_1 \\
A_1 & \stackrel{def}{=} & (m_6', rep_{m_6}).A_0
\end{array}
$$

This component follows A's lifeline through the **alt**, **loop** and **par** fragments to the final message $m6$. It follows A's lifeline through the upper part of the **par** fragment, incorporating message m_4.

A.1.2. Sequential Component $Starter_{A,2}$

This component follows A's lifeline through the lower half of the **par** fragment, incorporating message m_5.

$$
\begin{array}{rcl}
Starter_{A,2} & \stackrel{def}{=} & (start_1, r_{sync}).A_6 \\
A_6 & \stackrel{def}{=} & (m_5, req_{m_5}).A_5 \\
A_5 & \stackrel{def}{=} & (stop_1, r_{sync}).Starter_{A,2}
\end{array}
$$

The $start_1$ and $stop_1$ activities ensure that the m_5 activity cannot be executed too early.

A.1.3. Sequential Component B_0

Analogously to the component A_0 shown above, this component follows B's lifeline through the **alt**, **loop** and **par** fragments to the final message $m6$. It follows B's lifeline through the upper part of the **par** fragment, incorporating message m_4.

$$
\begin{array}{rcl}
B_0 & \stackrel{def}{=} & Choice_B \\
Choice_B & \stackrel{def}{=} & (path_1, \top).B_{12} \\
 & + & (path_2, \top).B_{14} \\
B_{12} & \stackrel{def}{=} & (m_1, \top).B_{11} \\
B_{11} & \stackrel{def}{=} & (m_1', rep_{m_1}).Loop_2 \\
B_{14} & \stackrel{def}{=} & (m_2, \top).B_{13} \\
B_{13} & \stackrel{def}{=} & (m_2', rep_{m_2}).Loop_2 \\
Loop_2 & \stackrel{def}{=} & (loop, \top).B_{10} \\
 & + & (exit, \top).Starter_{B,1} \\
B_{10} & \stackrel{def}{=} & (m_3, \top).B_9 \\
B_9 & \stackrel{def}{=} & (m_3', rep_{m_3}).Loop_2 \\
Starter_{B,1} & \stackrel{def}{=} & (start_1, r_{sync}).B_5 \\
B_5 & \stackrel{def}{=} & (m_4, \top).B_4 \\
B_4 & \stackrel{def}{=} & (beh_{m_4}, r_{beh_{m_4}}).B_3 \\
B_3 & \stackrel{def}{=} & (stop_1, r_{synch}).B_2 \\
B_2 & \stackrel{def}{=} & (m_6, req_{m_6}).B_1 \\
B_1 & \stackrel{def}{=} & (m_6', \top).B_0
\end{array}
$$

A.1.4. Sequential Component $Starter_{B,2}$

This component follows B's lifeline through the lower half of the **par** fragment, incorporating message m_5.

$$
\begin{array}{rcl}
Starter_{B,2} & \stackrel{def}{=} & (start_1, r_{sync}).B_8 \\
B_8 & \stackrel{def}{=} & (m_5, \top).B_7 \\
B_7 & \stackrel{def}{=} & (beh_{m_5}, r_{beh_{m_5}}).B_6 \\
B_6 & \stackrel{def}{=} & (stop_1, r_{sync}).Starter_{B,2}
\end{array}
$$

As before, the $start_1$ and $stop_1$ activities introduce synchronisation points which ensure that the m_5 activity cannot be executed too early.

A.1.5. Sequential Component *Thinking* This component models the workload and requests either *path1* or *path2* in the **alt** fragment.

$$
\begin{array}{rcl}
\mathit{Thinking} & \overset{\mathit{def}}{=} & (\mathit{think}, r).\mathit{Requesting} \\
\mathit{Requesting} & \overset{\mathit{def}}{=} & (\mathit{path}_1, \top).\mathit{Thinking} \\
 & + & (\mathit{path}_2, \top).\mathit{Thinking}
\end{array}
$$

A.2. System Equation

A PEPA model is formed by composing sequential components. The system equation defines the number of replications of each sequential components (e.g. N_W copies of *Thinking* are requested using $\mathit{Thinking}[N_W]$). In addition it configures the sequential components by defining the cooperation sets which they must operate under. For example, below N_A copies of the sequential component initiated in state A_0 are required to cooperate with N_B copies of the sequential component initiated in state B_0 over the activities in the set $\mathcal{L}$.

$$
\begin{array}{rcl}
\mathit{System} & \overset{\mathit{def}}{=} & \mathit{Thinking}[N_W] \underset{\mathcal{W}}{\bowtie} \\
 & & \Bigg(\Big(\big(A_0[N_A] \underset{\mathcal{L}}{\bowtie} B_0[N_B]\big) \underset{\mathcal{M}}{\bowtie} \\
 & & \quad \mathit{Starter}_{A,2}[N_A]\Big) \underset{\mathcal{K}}{\bowtie} \mathit{Starter}_{B,2}[N_B]\Bigg) \\
\mathcal{W} & = & \{\mathit{path}_1, \mathit{path}_2\} \\
\mathcal{L} & = & \{\mathit{path}_1, \mathit{path}_2, \mathit{loop}, \mathit{exit}, \mathit{start}_1, \mathit{stop}_1, m_4\} \cup \\
 & & \big\{m_i, m'_i : i \in \{1, 2, 3, 6\}\big\} \\
\mathcal{M} & = & \{\mathit{start}_1, \mathit{stop}_1, m_5\} \\
\mathcal{K} & = & \mathcal{M}
\end{array}
$$