Under consideration for publication in Formal Aspects of Computing

Performance Analysis of Probabilistic Action Systems

Stefan Hallerstede and Michael Butler[1]

University of Southampton, UK
[1]email: M.J.Butler@ecs.soton.ac.uk

Abstract. Formal notations like B or action systems support a notion of refinement. Refinement relates an abstract specification **A** to a concrete specification **C** that is as least as deterministic. Knowing **A** and **C** one proves that **C** refines, or implements, specification **A**. In this study we consider specification **A** as given and concern ourselves with a way to find a good candidate for implementation **C**. To this end we classify all implementations of an abstract specification according to their performance. We distinguish performance from correctness. Concrete systems that do not meet the abstract specification correctly are excluded. Only the remaining correct implementations **C** are considered with respect to their performance. A good implementation of a specification is identified by having some optimal behaviour in common with it. In other words, a good refinement corresponds to a reduction of non-optimal behaviour. This also means that the abstract specification sets a boundary for the performance of any implementation.

We introduce the probabilistic action system formalism which combines refinement with performance. In our current study we measure performance in terms of long-run expected average-cost. Performance is expressed by means of probability and expected costs. Probability is needed to express uncertainty present in physical environments. Expected costs express physical or abstract quantities that describe a system. They encode the performance objective. The behaviour of probabilistic action systems is described by traces of expected costs. A corresponding notion of refinement and simulation-based proof rules are introduced. Probabilistic action systems are based on discrete-time Markov decision processes. Numerical methods solving the optimisation problems posed by Markov decision processes are well-known, and used in a software tool that we have developed. The tool computes an optimal behaviour of a specification **A** thus assisting in the search for a good implementation **C**.

Keywords: Markov Decision Process, Simulation, Trace Refinement

1. Introduction

In recent years there has been growing interest in combining formal methods with performance analysis. The resulting developments gave rise to stochastic variants of established event-based and state-based formalisms (see Section 2). We distinguish between event-based and state-based formalisms by way of their behavioural semantics. The B-formalism [Abr96] is essentially state-based, but can also be considered event-based [AM98]. It lacks a means of performance analysis. This article describes our effort to supplement a B-like notation with a suitable notion of performance. Our notation has a relational semantics whereas the B-notation

Correspondence and offprint requests to: S. Hallerstede and M. Butler

uses a predicate transformer semantics. From the outset we only considered extensions that would support automatic calculation of performance measures. Realistically sized systems usually consist of thousands of states leading to thousands of equations to be solved. We consider it infeasible to do this by hand. Our extended notation, probabilistic action systems, is not event-based anymore. Neither events nor states are observable. The behaviour is described in terms of expected costs which capture all features that are relevant for performance. Despite this difference, refinement of probabilistic action systems is reminiscent of B refinement. This makes it easier to learn for someone familiar with B refinement already. In practice, this will also mean that experience in either is useful in the other.
A variety of formalisms is used for performance analysis [HR94, Hil96, MCB84, Mit98, STP96]. These formalisms are similar in that they offer only probabilistic choice operators. We regard systems where all choices are resolved probabilistically as deterministic. We prefer to include nondeterminism as it also admits a notion of abstraction that is made more concrete through refinement. There are tools for performance evaluation of queuing systems and networks, stochastic process algebras (e.g. [HHK$^+$00]), and stochastic Petri nets (e.g. [Lin98]). To our knowledge there is no such tool for B-like notations. For performance evaluation a specification in one of these languages is translated into a Markov process. Subsequently a performance measure for the Markov process is computed.
Syntactically, probabilistic action systems are close to B [Abr96] and probabilistic predicate transformers [MMS96]. The program constructs used also lean on [BvW98]. We decided on B as a foundation of probabilistic action systems because of its relatively widespread use. It is a rich mathematical notation, it has a notion of nondeterminism, and offers good structuring mechanisms. The inclusion of features for performance analysis was made as non-intrusive as possible. We believe the notation used for performance analysis should be close to the notation used in the development process. Then specifications used in B-refinement, or parts thereof, could be used in performance analysis and results from the analysis can easily be transferred back.
In Section 3 probabilistic programs are presented which, in addition to probabilistic choice, introduce the cost statement. The definition of sequential composition is delayed until Appendix A because it is rather technical and may not be immediately interesting. In Appendix B we sketch the correspondence to relational programs. In our earlier work we associated costs with states and actions [HB99]. Using cost statements instead simplifies the formalism as a whole, and increases the expressiveness with respect to possible performance objectives.
Section 4 introduces probabilistic action systems and explains the performance measure used in this study. The behaviour of these systems is described in terms of costs incurred during operation. Refinement is defined correspondingly, and proof rules for refinement and equivalence introduced. Refinement may be used to change the state space representation, or simply to justify that some deterministic system is an implementation. The latter being particularly important for optimal systems that solve the Markov decision process associated with a probabilistic action system. Equivalence is important in its own right because computing the optimal solution requires a small state space. Using equivalence we can prove that the reduced state space model has the same cost behaviour as the original model. In Section 5 a worked out case study is presented. We have implemented a software tool to assist in finding optimal implementations. The performance measures and optimal implementations presented in Section 5 have been calculated using that software tool. The tool takes an ASCII-representation of a probabilistic action system as input. This is translated into a labelled transition system which corresponds to a Markov decision process. A value iteration algorithm is then used to compute an optimal solution of the Markov decision process. Finally, using information kept during the translation, this optimal solution is printed out.

2. Related Work

As mentioned in the introduction we distinguish between state-based and event-based approaches to performance modelling.

State-Based Approaches

The models underlying the formalisms discussed in this and the next section are called Markov processes [KS76] and Markov decision processes [Tij94]. The two models themselves are used to model state-based stochastic systems [Put94, Sen99]. However, they are impractical to use in the performance analysis of complex systems because no structuring mechanisms are available [HT93]. Queuing systems are the traditional

structured formalism used in performance analysis [Mit98]. They have also been applied to computer systems performance modelling [Hav98, Kan92, Nel95, STP96]. Usual performance measures derived from queuing systems are: system throughput, average numbers of waiting customers at stations in a queuing network, and waiting times [Mit98]. The underlying model of a queuing system is a Markov process, a model that can only represent deterministic systems.
Stochastic Petri nets [MCB84, Mol82] have been applied to performance modelling of computer architectures. Their origin are classical Petri nets [Rei85] which are described by a collection of places, transitions and markings. Exponentially distributed firing delays are used to model uncertain behaviour. Consequently, stochastic Petri nets model real-time and probability. There is no notion of nondeterminism though. The operational behaviour of (generalised) stochastic Petri nets is characterised by the interaction of immediate transitions and exponentially delayed transitions. If two exponentially delayed transitions t_1 and t_2 in a stochastic Petri net compete for a token, the conflict is resolved probabilistically. Let μ_1 and μ_2 be the transition rates of t_1 and t_2, i.e. the mean time it takes for t_i to fire is $1/\mu_i$. Then transition t_i fires with probability $p_i = \mu_i/(\mu_1 + \mu_2)$. The use of the transition probabilities p_i corresponds to a shift to discrete time [Mit98]. If an immediate transition is enabled in a marking, that marking "vanishes", i.e. in the semantical model the marking is not visible. If an immediate transition conflicts with an exponentially delayed one, the immediate transition has priority. And if two immediate transitions conflict, the conflict is resolved by explicitly specified priorities. The generalised model [MCB84] added immediate transitions to the original model [Mol82]. The performance measures that can be derived from stochastic Petri nets are expectations of functions of markings, and probabilities of predicates over markings [Lin98]. In [Lin98] the lack of nondeterminism in these formalism is partly remedied by the use of parameterised specifications, e.g. experiments in [Lin98]. But there is no means to reason about these parameters from within the formalism.
The action system formalism [BvW94] has been extended in [ST96] with probabilistic features for reliability analysis. It is based on the probabilistic extension [MMS96] of the guarded command language [Dij76]. The probabilistic guarded command language contains notions of nondeterminism and probabilistic choice but is not compatible with the general performance measures supported by Markov decision processes and used in our approach. In [Tro99] this has been partly rectified by using parameterized refinement similar to [Lin98]. By insisting on a close correspondence to standard probability theory our approach is similar to [Heh]. However, our model is closely based on Markov decision processes so that tool support can be easily achieved.

Event-Based Approaches

The event-based formalisms for performance analysis are usually based on classical process algebras like CCS [Mil89], CSP [Hoa85] or LOTOS [BB89]. They are generally called stochastic process algebras, e.g. EMPA [BDG98], MPA [Buc94], TIPP [HR94, HRW95], or PEPA [Hil96].
Stochastic process algebras are usually deterministic in the sense that for all choices there are (stochastic) instructions for how to resolve them. Similar to stochastic Petri nets, stochastic process algebras use exponentially distributed delays between events. Their behaviour is usually described by labelled transition systems or traces of actions. Let μ denote a rate and a an action. An activity α is defined by a tuple (a, μ). A choice between two activities $\alpha_1 = (a_1, \mu_1)$ and $\alpha_2 = (a_2, \mu_2)$ is resolved similarly to conflict resolution in stochastic Petri nets: action a_i occurs with probability $\mu_i/(\mu_1 + \mu_2)$. Stochastic process algebras have the usual combinators, like synchronisation or hiding. The definition of synchronised composition varies between the different algebras. Internal actions are denoted by the special symbol τ. Internal actions themselves are not observable, only their effect is. As in classical process algebras notions of bisimulation and equivalence between process terms exist. These form the basis of methods to reduce the size of the semantical model of process terms for numerical analysis [HHK$^+$00, Hil96].
The stochastic process algebras TIPP and EMPA also have notions of nondeterminism. In fact, the stochastic process algebra EMPA has language kernels that correspond to classical process algebra like CCS, stochastic process algebras like MPA, and probabilistic process algebras like probabilistic CSP [Sei95]. However, in EMPA performance analysis is only possible for specifications that do not contain nondeterminism. In TIPP the process term being analysed must be bisimilar to a deterministic process term, effectively saying the original process term does not contain nondeterministic choices.
Typical measures used with stochastic process algebras are probabilities of process states, throughput, and means based on variables present in parametric processes [HHK$^+$00]. In [BCSS98, Ber97] the stochastic

process algebras PEPA and EMPA have been equipped with means to specify more general performance measures based on rewards [Put94]. In EMPA$_r$ [Ber97] activities are triples (a, μ, r) where a and μ are as above and $r \in \mathbb{R}$ is a reward associated with action a. The article [Ber97] gives examples on how to express standard performance measures as mentioned above in EMPA$_r$. This is extended in [BCSS98] to include also rewards associated with states. Tool support for any of these methods is generally considered to be essential for the method to be useful in practice [HN96]. Software tools are available to compute performance measures specified in the process algebras PEPA: PEPA Workbench [GH94], TIPP: TIPPtool [HHK$^+$00], and EMPA$_r$: TwoTowers [BCSS98]. The tool TwoTowers also supports some model checking of functional aspects specified in EMPA$_r$ process terms.

3. Probabilistic Programs

The program semantics we use is based on relational programs as presented in [BvW98]. In this section this model is extended to deal with probability and expected costs. The probabilistic constructs are similar to [JSM97, MMS96]. Technically, probabilistic programs are defined in a similar way to the (probabilistic) relational model proposed in [JSM97]. The difference is that in [JSM97] nondeterminism is regarded as a generalised form of probabilistic choice. Our probabilistic programs keep both concepts separate from each other. They are based on Markov decision processes [Put94]. We have chosen this model over the model in [JSM97] because in the refinement of probabilistic action systems we seek a non-probabilistic control program that is derived by refining an initial nondeterministic system. Its existence is guaranteed by the theory of Markov decision processes. This is discussed further in [Hal01].
We briefly review the discrete-time model of Markov decision processes presented in [Tij94]. Discrete-time means that processes are observed at equidistant points of time $0, 1, 2$, etc. At time n a process M is in some state $\tau \in \Gamma$ and a decision has to be made on the next action to take. The set of all possible states is denoted by Γ. For each state τ there is a set of possible actions $A.\tau$. Both the state space Γ and the sets $A.\tau$, $\tau \in \Gamma$, are assumed to be finite in Markov decision processes. If process M chooses action $a \in A.\tau.$ in state τ, then it incurs a cost $c_\tau.a$, and at time $n+1$ it will be in state τ' with probability $p_{\tau\tau'}.a$ where $\sum_{\tau'} p_{\tau\tau'}.a = 1$. The distinguishing feature of Markov processes is that the incurred cost $c_\tau.a \geq 0$ and transition probabilities $p_{\tau\tau'}.a$ depend on the current state but are independent of the history of process M.

Probabilistic States

The state of probabilistic programs is described by a collection of states and their corresponding probabilities. More precisely, a *probabilistic state* $f \in \Gamma \to \mathbb{R}_{\geq 0}$ is a function that assigns probabilities to states. The set $\mathbb{D}\,\Gamma$ of all probabilistic states over Γ is defined by

$$(\mathbb{D}\,\Gamma).f \;\widehat{=}\; \mathsf{card}.(\mathsf{car}.f) \in \mathbb{N} \wedge \textstyle\sum_{\tau\in\Gamma} f.\tau = 1 \;,$$

where $\mathsf{car}.f \subseteq \Gamma$, the *carrier* of f, is defined by

$$\mathsf{car}.f.\tau \;\widehat{=}\; f.\tau > 0 \;.$$

The set $\mathsf{car}.f$ describes a set of states in which a probabilistic program may be at some instant. The expression $\mathsf{card}.(\mathsf{car}.f) \in \mathbb{N}$ means that this set is finite. The value $f.\tau$ is the probability that the program is in state τ. We use the notation $\tau \mathbin{@} p$ to represent $f.\tau = p$, and the notation

$$\{\tau_1 \mathbin{@} p_1, \tau_2 \mathbin{@} p_2, \ldots, \tau_n \mathbin{@} p_n\} \;,$$

where $\mathsf{car}.f \subseteq \{\tau_1, \tau_2, \ldots, \tau_n\}$ (and $1 = \sum_{i=1}^{n} p_i$), to represent probabilistic state f itself. A similar notation is used in [SMM97]. Probabilistic states are known as *densities*, or *masses*, in probability theory. We have decided to use the term 'probabilistic state' because phrases like 'program P is in probabilistic state f' sound more intuitive than if one of the other terms was used.
We define two operators on probabilistic states. We need addition and scalar product of functions $\Gamma \to \mathbb{R}_{\geq 0}$. They are defined by point-wise extension: $(f+g).\tau \mathrel{\widehat{=}} f.\tau + g.\tau$, $(p*f).\tau \mathrel{\widehat{=}} p*f.\tau$, where $f, g \in \Gamma \to \mathbb{R}_{\geq 0}$ and $p \in (0, 1)$. Probabilistic addition of f and g is defined by

$$f \;{}_p\!\oplus\; g \;\widehat{=}\; p*f + (1-p)*g \;.$$

If a program is in probabilistic state f with probability p and in probabilistic state g with probability $1-p$, then its probabilistic state is $f\ {}_p\!\oplus\ g$. This situation arises when a program branches to probabilistic states f and g with the respective probabilities p and $1-p$. Probability p is required to be in the open interval to avoid complications in the semantics of probabilistic programs. The expressiveness of the language would not change, however, if 0 and 1 were allowed.

Probabilistic Programs

Probabilistic programs relate states with pairs (c, f) of expected costs and probabilistic states. Costs c have no effect on the execution of a program. They record expected costs associated with execution paths. Considered on their own, expected costs yield a more abstract description of the behaviour of a program. If f is chosen as the successor probabilistic state, then expected cost c is incurred by the program. Cost c does characterise the choice made but without details about the probabilistic state chosen. Expected cost is usually not specified directly in a specification. Instead, costs (as opposed to expected costs) are specified at different locations in a program. From these costs, expected costs c are derived based on the probabilistic choices choices in the program. In [SMM97] expectation transformers are introduced where expected costs entirely replace probabilistic state. Probabilistic programs are used to describe the behaviour of probabilistic action systems in section 4.
Let Γ and Γ' be state spaces. A relation P from Γ to $\mathbb{R}_{\geq 0} \times \mathbb{D}\,\Gamma'$ is called a *probabilistic program*. The set of all these is denoted by $\mathcal{P}(\Gamma, \Gamma')$.
A cost is a nonnegative real number. Similar to probabilistic addition and sum above we introduce such operators for costs. They yield expected costs when choices between different program branches occur probabilistically. Let $p \in (0,1)$ and $c_1,\ c_2 \in \mathbb{R}_{\geq 0}$. We define:

$$c_1\ {}_p\!\oplus\ c_2 \ \ \widehat{=}\ \ p * c_1 + (1-p) * c_2\ .$$

The expression on the right hand side calculates the expected cost for a probabilistic choice. If cost c_1 is incurred with probability p and cost c_2 with probability $1-p$, then the expected cost incurred is $c_1\ {}_p\!\oplus\ c_2$. We refer to $c_1\ {}_p\!\oplus\ c_2$ as expected cost. For a probabilistic state $f \in \mathbb{D}\,\Gamma$ and a function $C \in \Gamma \to \mathbb{R}_{\geq 0}$ we define the product $f * C$ by

$$f * C \ \ \widehat{=}\ \ \textstyle\sum_{\tau \in \mathsf{car}.f} f.\tau * C.\tau\ .$$

The sum denotes the expected cost that is incurred when a program is in probabilistic state f and continuation causes costs $C.\tau$ in state τ. We call C a *cost function*.
Arbitrary nonnegative costs are associated with programs by means of cost statements. Upon encountering such a statement a program incurs the specified cost, so that an execution yields an expected cost as well as a final probabilistic state. For a real expression $C \in \Gamma \to \mathbb{R}_{\geq 0}$ the cost statement $\mid C \mid \in \mathcal{P}(\Gamma, \Gamma)$ costs value $C.\tau$ at state τ and behaves like skip on the state. It is defined by

$$\mid C \mid.\tau.(c, f) \ \ \widehat{=}\ \ c = C.\tau \wedge f = \chi.\tau\ ,$$

where for a state space Γ and $\tau \in \Gamma$ the point density $\chi.\tau \in \mathbb{D}\,\Gamma$ is defined by $\chi.\tau.\tau = 1$ and $\chi.\tau.\tau' = 0$ for all $\tau' \neq \tau$. A probabilistic program P is called *cost-free* if $P.\tau.(c, f)$ implies $c = 0$ for all τ.
For a predicate q over Γ the guard $\mid q \mid$ blocks execution whenever $\neg q$ holds. It is defined by

$$\mid q \mid.\tau.(c, f) \ \ \widehat{=}\ \ f = \chi.\tau \wedge q.\tau\ .$$

We define $\mathsf{skip} \ \widehat{=}\ \mid \mathsf{true} \mid$. Assignment $x := e$ for a variable x and an expression e is defined by lifting the relational program $x := e$ to the corresponding probabilistic program (see Appendix A). Relational assignment is defined as usual; variable x is set to the value described by expression e, and all other variables are not changed.
We define nondeterministic choice between probabilistic programs $P \in \mathcal{P}(\Gamma, \Gamma')$ and $Q \in \mathcal{P}(\Gamma, \Gamma')$ by

$$(P \bigsqcup Q).\tau.(c, f) \ \ \widehat{=}\ \ P.\tau.(c, f) \vee Q.\tau.(c, f)\ .$$

Nondeterministic choice takes costs into account that are associated with the alternatives P and Q. Different costs are incurred by a program depending on which alternative an execution follows. Finite nondeterministic choice is defined similarly [Hal01]. It is denoted by the construct $\bigoplus i \in I \mid p_i \bullet P_i$, where each branch P_i is executed with probability p_i. It corresponds to a finite repetition of binary probabilistic choice.

For $p \in \Gamma \to (0,1)$ probabilistic choice between $P \in \mathcal{P}(\Gamma, \Gamma')$ and $Q \in \mathcal{P}(\Gamma, \Gamma')$ is defined by

$$(P \ {}_p\!\oplus\ Q).\tau.(e,h) \ \widehat{=}\ \exists (c,f) \in P.\tau,\ (d,g) \in Q.\tau \bullet \\ e = c \ {}_{p.\tau}\!\oplus\ d \wedge h = f \ {}_{p.\tau}\!\oplus\ g \ .$$

In words, (e,h) consists of an expected cost e and a corresponding expected probabilistic state h resulting from probabilistic choice $P \ {}_p\!\oplus\ Q$ in state τ. Finite probabilistic choice between probabilistic programs is defined similarly [Hal01].
We also define a lifted form of parallel composition of programs $P \in \mathcal{P}(\Gamma, \Gamma_1')$ and $Q \in \mathcal{P}(\Gamma, \Gamma_2')$. By adding the costs of the component programs P and Q we keep the correspondence $P \parallel Q = P; Q$ between certain programs P and Q in the presence of costs. Sequential and parallel composition both accumulate expected costs. We define

$$(P \parallel Q).\tau.(e,h) \ \widehat{=}\ \exists (c,f) \in P.\tau,\ (d,g) \in Q.\tau \bullet \\ e = c + d \wedge h = f \parallel g \ ,$$

where $f \parallel g$ denotes the point-wise product of f and g:

$$(f \parallel g).(\tau_1, \tau_2) \ \widehat{=}\ f.\tau_1 * g.\tau_2 \ .$$

The type of $P \parallel Q$ is $\mathcal{P}(\Gamma, \Gamma_1' \times \Gamma_2')$.
Sequential composition $P; Q$ of probabilistic programs P and Q is defined in Appendix A. We also show that probabilistic sequential composition behaves as relational sequential composition on non-probabilistic programs. Similar results hold for other relational operators [Hal01].
As an alternative to our approach to treating expected costs, they could also be calculated by introducing an explicit state component c, say, to represent costs. However, our approach is more convenient as it frees a user from the burden to specify how expected costs from different program components are to be combined. This is especially true in the presence of $\parallel$ where different cost variables would have to be used on either side.

4. Probabilistic Action Systems

In this section probabilistic action systems are introduced. They are based on a programming notation that combines probabilistic and nondeterministic choice. This makes it possible to reason about optimality in a single framework. Furthermore, the semantical similarity to Markov decision processes enable efficient tool support. Their behaviour is described by traces of costs. A cost is a positive real number. A probabilistic action system models a closed system which includes the environment. The traces of a probabilistic action system are manifestations of its cost structure. The cost structure is specified by way of the cost statement. Refinement in this context means subsumption of the cost structure of a refined probabilistic action system by that of the initial probabilistic action system. A cost trace of a probabilistic action system conceals the actions. One only knows that a cost trace corresponds to a history of state changes. It is assumed that each state change takes exactly one unit of time. A unit of time is a time interval of some length which depends on the application context. We also refer to the time interval as *time slot* or *transition period*. The state change occurring during a transition period is termed *transition*.
A probabilistic action system $\mathbf{A}$, or *system* for short, is defined by a tuple (Γ, I, P) where

Γ is a finite state space,
$I \subseteq \mathbb{D}\,\Gamma$ is a set of probabilistic states, the *initialisation* of $\mathbf{A}$, and
$P \in \mathcal{P}(\Gamma, \Gamma)$ is a program, the *action* of $\mathbf{A}$.

Finiteness of the state space is required to guarantee the existence of an optimal solution of the associated Markov decision process [Put94].
The behaviour of probabilistic action system $\mathbf{A}$ is described by its traces and impasses. A trace t is a sequence of non-negative real numbers ($t \in \operatorname{seq} \mathbb{R}_{\geq 0}$). The real numbers correspond to expected costs system $\mathbf{A}$ may incur during single transitions periods in its evolution. An impasse is a trace after which the system may not be able to continue. The state of a probabilistic action system is not observable directly. However, the traces of a system are the observable consequence of state changes the system undergoes.

The evolution of $\mathbf{A}$ is described via sequences of these expected costs. The sequences of probabilistic states are implicitly contained in the definition of $\mathsf{path}.\mathbf{A}$. If $\mathsf{path}.\mathbf{A}.t.f$ is true, action system $\mathbf{A}$ may undergo cost trace t leading to probabilistic state f.

$$\mathsf{path}.\mathbf{A}.\langle\,\rangle.f \;\widehat{=}\; f \in I$$
$$\mathsf{path}.\mathbf{A}.t^\frown\langle c\rangle.f \;\widehat{=}\; \exists\, g \bullet \mathsf{path}.\mathbf{A}.t.g \wedge \widehat{P}.g.(c,f)\ .$$

The term $\widehat{P}$ denotes a lifted form of P that is based on sequential composition. Its definition can be found in Appendix A. It describes the effect of P having a probabilistic state g as a start state instead of a state τ. The set $\widehat{P}.g$ is empty if $\mathsf{car}.g \subseteq \mathsf{dom}.P$ does not hold. In the definition of path this means, having reached probabilistic state g program P is executed. So t contains the expected costs incurred during each transition until probabilistic state f is reached. The semantics of $\mathbf{A}$ is defined by $\mathsf{beh}.\mathbf{A} \mathrel{\widehat{=}} (\mathsf{tr}.\mathbf{A}, \mathsf{im}.\mathbf{A})$ where $\mathsf{tr}.\mathbf{A}$ are called the *traces* of $\mathbf{A}$,

$$\mathsf{tr}.\mathbf{A}.t \;\widehat{=}\; \exists f \bullet \mathsf{path}.\mathbf{A}.t.f\ ,$$

and $\mathsf{im}.\mathbf{A}$ are called the *impasses* of $\mathbf{A}$

$$\mathsf{im}.\mathbf{A}.t \;\widehat{=}\; \exists f \bullet \mathsf{path}.\mathbf{A}.t.f \wedge \widehat{P}.f = \varnothing\ .$$

A probabilistic action system $\mathbf{A}$ is called *live* if it does not have impasses, i.e. $\mathsf{im}.\mathbf{A} = \varnothing$. Finite traces that can be continued indefinitely are called infinite traces. The infinite traces $\mathsf{itr}.\mathbf{A}$ of a system $\mathbf{A}$ are defined by:

$$\mathsf{itr}.\mathbf{A}.v \;\widehat{=}\; \forall\, t \in \mathsf{seq}\,\mathbb{R}_{\geq 0} \bullet t \leq v \Rightarrow \mathsf{tr}.\mathbf{A}.t\ ,$$

where $t \leq v$ means that trace t is a prefix of infinite trace v. The behaviour of a live system $\mathbf{A}$ is entirely described by the infinite traces it can engage in because system $\mathbf{A}$ has no impasses $\mathsf{im}.\mathbf{A} = \varnothing$, and:

Theorem 4.1 Let $\mathbf{A}$ be a live system, then

$$\mathsf{tr}.\mathbf{A}.t \;\Leftrightarrow\; \exists\, v \in \mathsf{itr}.\mathbf{A} \bullet t \leq v$$

for all $t \in \mathsf{seq}\,\mathbb{R}_{\geq 0}$.

In a later section we define a performance measure for live systems, expected average-cost. The measure is not defined for systems with impasses. Refinement, on the other hand, is defined more generally in terms of traces and impasses, not requiring liveness. It is possible to define other performance measures on probabilistic action systems, for instance, expected discounted cost or finite-horizon measures (see [Put94]) as seems appropriate for the system being modelled.

Syntax

We introduce the syntactic form of probabilistic action systems by way of an example. The syntactic representation of probabilistic action systems is based on a subset of B and the guarded command language of [BvW98] adding probabilistic imperative features as in [MMS96]. A syntactic system has the structure pictured in Figure 1. In the constants section natural number constants are declared. Usually these represent parameters for a specification which constrain the state space to be finite. Possible values of the constants are constrained by the predicate of the constraints section. The sets section contains constant sets that are used elsewhere in the specification. The state is declared as a collection of variables in the variables section. The data types that can be used are similar to those of B: sets, Cartesian products, power sets, functions, and relations. In our approach sets can be finite ranges of integer numbers, or any other finite set with its symbolic values being enumerated. Initial values for variables are given in the initialisation section. The operational behaviour is further specified in the actions section containing a number of actions. Multiple actions are a specification convenience. They are implicitly combined using nondeterministic choice to form a single action. Initialisation and actions of a system are atomic. Reoccurring program text can be given a name and declared in the procedures section.

Example 4.2 Figure 1 depicts an abstraction of a simple queuing system. The waiting queue is represented by its size. In the constants section the maximal size of the queue is declared, and required to be positive

system $QUEUE$
constants
  $SIZE$;
constraints
  $SIZE > 0$;
sets
  $QQ \;=\; 0\,..\,SIZE$;
variables
  $qq \in QQ$;
initialisation
  $\bigsqcup xx \in QQ \bullet qq := xx$;
procedures
  $aa \;=\; (\mid qq < SIZE \mid; qq := qq + 1) \sqcup \mid qq = SIZE \mid$;
  $dd \;=\; (\mid qq > 0 \mid; qq := qq - 1) \sqcup \mid qq = 0 \mid$;
actions
  $queue \;=\; \mid qq \mid; (aa \;{}_{\frac{1}{6}}\!\oplus\, \mathsf{skip}); ((dd \;{}_{\frac{1}{3}}\!\oplus\, \mathsf{skip}) \sqcup (dd \;{}_{\frac{1}{4}}\!\oplus\, \mathsf{skip}))$;
end

Fig. 1. A simple queuing system

in the constraints section. The set QQ defined in the sets section contains the possible values of variable qq. Initially the queue size qq may assume one of the values $xx \in QQ$. In the procedures section arrivals aa and departures dd are specified. These are used in the actions section. It consists of a single action $queue$. At the start of each transition period the size of the queue qq is measured to yield a cost. Then possible arrivals occur with a rate of $\frac{1}{6}$, and departures may occur with a rate of $\frac{1}{3}$ or $\frac{1}{4}$.
Assume $SIZE$ equals 2. If system $QUEUE$ is in state $qq = 1$, action $queue$ may increase qq, decrease it, or leave it unchanged. The cost of the transition is 1 in any case. Assume the server was fast and the departure rate equals $\frac{1}{3}$ for the moment. Then the successor state would be

$$\left\{0 \mathbin{@} \tfrac{5}{18}, 1 \mathbin{@} \tfrac{11}{18}, 2 \mathbin{@} \tfrac{2}{18}\right\}$$

The expected cost of the next transition from this probabilistic state is $\frac{10}{12} = 0 * \frac{5}{18} + 1 * \frac{11}{18} + 2 * \frac{2}{18}$. The sequence $\langle 1, \frac{10}{12}\rangle$ is a trace of system $QUEUE$. If the departure rate equalled $\frac{1}{4}$ instead of $\frac{1}{3}$ in state $qq = 1$, the expected cost of the first transition would be 1, followed by $\frac{11}{12}$. So both $\langle 1, \frac{10}{12}\rangle$ and $\langle 1, \frac{11}{12}\rangle$ are traces of $QUEUE$. In system $QUEUE$ all traces may continue indefinitely.

Refinement

Refinement of probabilistic action systems may reduce the number of traces and the number of possible impasses. The cost structure of a system is the basis for the performance measures introduced below. The performance measures are only defined for live systems, i.e. systems that have no impasses. Since the number of impasses cannot increase, a measure defined on a live system is also defined on any of its refinements. Hence the performance of a system can be compared to that of any of its refinements.
Probabilistic action system $\mathbf{C}$ refines probabilistic action system $\mathbf{A}$, denoted by $\mathbf{A} \sqsubseteq \mathbf{C}$, if all behaviour possible for $\mathbf{C}$ is also possible for $\mathbf{A}$. That is system $\mathbf{C}$ has less traces and less impasses than system $\mathbf{A}$. We define:

$$\mathbf{A} \sqsubseteq \mathbf{C} \;\;\widehat{=}\;\; \mathsf{tr}.\mathbf{C} \subseteq \mathsf{tr}.\mathbf{A} \;\wedge\; \mathsf{im}.\mathbf{C} \subseteq \mathsf{im}.\mathbf{A}\ .$$

Probabilistic action systems $\mathbf{A}$ and $\mathbf{C}$ are called equivalent, denoted by $\mathbf{A} \equiv \mathbf{C}$, if $\mathbf{A} \sqsubseteq \mathbf{C}$ and $\mathbf{C} \sqsubseteq \mathbf{A}$.
When proving cost refinements $\mathbf{A} \sqsubseteq \mathbf{C}$ we do not use the definition directly. Instead of checking entire traces we compare the step by step behaviour of systems $\mathbf{A}$ and $\mathbf{C}$. Simulation is a proof technique to do this: System $\mathbf{C}$ refines system $\mathbf{A}$ if system $\mathbf{A}$ can simulate the behaviour of system $\mathbf{C}$ step by step. This method is widely used in formalisms that have trace-based behaviour [BvW94, Jos88] and in data refinement [Abr96, WD96]. Thus, someone who is able to carry out conventional refinement proofs should be able to carry out probabilistic refinement proofs without much difficulty.

Although the behaviour of probabilistic action systems is modelled by traces of costs instead of traces of actions, simulation of probabilistic action systems looks very similar to conventional simulation (e.g. [Jos88]). Theorem 4.3 presents a simulation method that establishes refinement.

Theorem 4.3 (Refinement Proof Rule) Let $\mathbf{A} = (\Gamma_A, I, P)$ and $\mathbf{C} = (\Gamma_C, J, Q)$ be two probabilistic action systems. If there is a deterministic cost-free probabilistic program $M \in \mathcal{P}(\Gamma_A, \Gamma_C)$, such that

$$J \subseteq I; M \tag{1}$$
$$\mathsf{dom}.P \subseteq \mathsf{dom}.(M; Q) \tag{2}$$
$$M; Q \subseteq P; M \tag{3}$$

then $\mathbf{A} \sqsubseteq \mathbf{C}$.

We call the probabilistic state function M in Theorem 4.3 *probabilistic simulation*. It creates a link between the states of the two systems. If M was allowed to be nondeterministic in Theorem 4.3, new impasses could be introduced [Hal01]. Our proof rule differs from corresponding conventional proof rules in (2). This would make it possible for a nondeterministic simulation to introduce new states in the concrete system that lead to impasses. The appearance of our proof rule is dictated by the way probabilistic programs are defined as relations from states to probabilistic states.
Condition (1) ensures that the initialisation of the concrete system $\mathbf{C}$ can be matched by the initialisation of the abstract system $\mathbf{A}$. Condition (2) ensures that the abstract system can refuse to continue whenever the concrete system can do so. Hence any impasse of the concrete system must also be an impasse of the abstract system. Condition (3) ensures that the effect of the concrete action is matched by that of the abstract action.
Refinement may remove traces and impasses. Sometimes it is desirable to keep the entire cost behaviour intact by proving equivalence between the systems. Equivalence proves especially useful as a tool for the numerical solution of associated performance measures. A major problem in computing these measures is the size of the state space of typical systems. To make the analytical solution feasible, the original system is replaced with an equivalent system of a much smaller size. In principle we could apply Theorem 4.3 twice to prove equivalence of two systems. Instead we prefer to use a stronger Theorem 4.4 to prove equivalence directly. States can be identified which are behaviourally indistinguishable. This technique is called *aggregation* in [Hil96]. The approach to aggregation taken in [Buc94, Hil96] is based on bisimulation between the used stochastic process algebras. It is referred to as "lumpability". In their approach the way states are "lumped" together is fixed by the definition of a bisimulation between processes. Since our approach allows for the specification of more general performance measures we need more freedom when "lumping" states. The concept of lumpability originates in the theory of Markov chains [KS76]. Theorem 4.4 treats simulation between equivalent probabilistic action systems. It strengthens Theorem 4.3.

Theorem 4.4 (Equivalence Proof Rule) Let $\mathbf{A} = (\Gamma_A, I, P)$ and $\mathbf{C} = (\Gamma_C, J, Q)$ be two probabilistic action systems. If there is a deterministic cost-free probabilistic program $M \in \mathcal{P}(\Gamma_A, \Gamma_C)$, $\mathsf{dom}.M = \Gamma_A$, such that

$$J = I; M \tag{4}$$
$$M; Q = P; M \tag{5}$$

then $\mathbf{A} \equiv \mathbf{C}$.

Optimal Systems

In system *QUEUE* (Figure 1) execution can continue from any state. For such systems average-cost optimality is considered an appropriate performance measure [Tij94]. It measures long-run expected average-cost per unit time. In our model each transition takes one unit of time, and there is no time between two transitions.
In the remainder of this section we use system *QUEUE* to explain the concept of average-cost optimality. Each of the infinite traces $v \in \mathsf{itr}.QUEUE$ corresponds to choices of actions that have been made during the

evolution of the system as suggested above. The quantity[1]

$$\mathsf{avg}.v \;\widehat{=}\; \mathsf{lim}_{n\to\infty}\, \tfrac{1}{n} \textstyle\sum_{i=1}^{n} v.i \tag{6}$$

gives the average value of the infinite trace v. The value $\mathsf{avg}.v$ is called the long-run expected average cost of v. The infinite trace w minimising $\mathsf{avg}.w$ is called average-cost optimal. Such an optimal sequence exists for all finite state systems. There is always a refinement that is an optimal implementation [Hal01]. In the case of system *QUEUE* the deterministic system with departure rate $\frac{1}{3}$ is optimal among all possible implementations. There are other performance measures that can be used depending on the problem at hand. This includes measures for systems in continual operation as well as measures for systems that stop at some point [Put94].
The average-cost optimal value of a live system $\mathbf{A}$ is defined in terms of infinite traces of costs by

$$\mathsf{val}.\mathbf{A} \;\widehat{=}\; \mathsf{min}_{v\in \mathsf{itr}.\mathbf{A}}\, \mathsf{avg}.v \ .$$

The optimal value $\mathsf{val}.\mathbf{A}$ corresponds to the best possible behaviour of $\mathbf{A}$ as opposed to best guaranteed. We note that val is a monotonic performance measure, as stated in the following theorem.

Theorem 4.5 $\mathbf{A} \sqsubseteq \mathbf{C} \;\Rightarrow\; \mathsf{val}.\mathbf{A} \leq \mathsf{val}.\mathbf{C}$.

From this we see that it is possible to have refinements that can only perform worse. A queuing system that works with a departure rate of $\frac{1}{4}$ is also a refinement of system *QUEUE* though its optimal value is worse than $\mathsf{val}.\mathit{QUEUE}$. Ideally we like to choose a refinement that maintains the optimal cost, but this is not guaranteed by refinement itself and must be checked separately.

A Software Tool to Compute Optimal Implementations

We have implemented a software tool that computes an optimal implementation and the optimal value of a probabilistic action system with respect to average-cost performance. It uses algorithms from dynamic programming [Put94] to do this. The tool consists of four parts: a parser that creates an abstract syntax tree, an interpreter that computes the Markov decision process corresponding to the system, a solver that computes the optimal solution, and a printer that prints out the solution in human-readable form.
The software tool automatically checks for liveness of a system, and other properties [Put94] useful in the context of dynamic programming. We have experimented with different dynamic programming algorithms and are now using value iteration with a preceding aperiodicity transformation [Tij94]. It produced the best results in terms of runtime and space requirements. To be able to apply dynamic programming state spaces of probabilistic action systems must be finite. Optimal solutions may not exist otherwise. Probabilistic programs have been defined so that there is a close correspondence to Markov decision processes. In particular, sequential composition is similar to matrix multiplication (see Appendix A).
The interpreter dominates the runtime of the software. For the smallest system shown the overall runtime is 2 minutes and for the largest it takes several hours where almost the entire time is consumed by the interpreter. The result is printed out in form of a collection of guarded actions the nondeterministic composition of which yields the (deterministic) optimal implementation. See [Hal01] for details on the tool, its implementation, and performance.

5. Example

The cyclic polling system treated in this section is similar to those investigated in [STP96, Sen99]. However, they use continuous time whereas we use a discrete time model. The system is specified as shown in Figure 2. It consists of a number of *STATIONS* arranged in a ring. Each station is equipped with a buffer of some maximal *CAPACITY*. The stations are numbered from 1. The successor of station i is station $i+1$ unless i is the last station whose successor is station 1. Function *NEXT* defines the successor relation. A server travels around the ring from one station to its successor. Being at station i the server can either remain

[1] See [Hal01] for a justification that the limit is well-defined for finite state systems.

system $POLLING$
constants
$STATIONS$; $CAPACITY$;
constraints
$STATIONS > 0 \wedge CAPACITY > 0$;
sets
$STATION = 1\,..\,STATIONS$;
$NEXT = (\lambda\, s \in 1\,..\,(STATIONS - 1) \bullet s + 1) \cup \{STATIONS \mapsto 1\}$;
variables
$station \in STATION$;
$buffer \in STATION \to 0\,..\,CAPACITY$;
$moving \in \mathbb{B}$;
programs
$arrival$ = // arrival of packets at stations
$\bigoplus S \in \mathbb{P}(buffer^{-1}[0\,..\,CAPACITY - 1])$
$\quad | \left(\frac{1}{10}\right)^{\mathsf{card}(S)} * \left(\frac{9}{10}\right)^{\mathsf{card}(buffer^{-1}[0..CAPACITY-1]-S}$ •
$\quad\quad buffer := buffer \lhd (\lambda\, s \in S \bullet buffer.s + 1)$;
$departure$ = // departure of processed packets from $station$
$\quad | \; buffer.station > 0 \; |;$
$\quad\quad (buffer := buffer \lhd \{station \mapsto buffer.station - 1\} \;_{0.25}\!\oplus\, \mathsf{skip})$
$\quad \sqcup$
$\quad | \; buffer.station = 0 \; |;$
initialisation
$station := 1 \parallel moving := \mathsf{false} \parallel buffer := STATION \times \{0\}$;
actions
$serve$ =
$\quad | \sum s \in STATION \bullet buffer.s \; |;$
$\quad | \neg\, moving \; |; arrival; departure;$
$walk$ =
$\quad | \; 1.0 \; |;$
$\quad | \sum s \in STATION \bullet buffer.s \; |;$
$\quad arrival;$
$\quad (moving := \mathsf{true} \;_{0.5}\!\oplus\, (moving := \mathsf{false} \parallel station := NEXT.station))$;
end

Fig. 2. A cyclic polling system

there or move to the next station, where it must reside for at least one unit of time. If the server decides to stay at some location i it serves packets from $buffer.i$ with an average rate of $\frac{1}{4}$ packets per unit of time, or it idles if the buffer is empty. Packets arrive at the beginning of each time slot with a rate of $\frac{1}{10}$. On arrival at station i a packet is added to $buffer.i$ if that buffer is not full. The mean time it takes for the server to get from one station to the next is 2 units of time. This means the probability of arriving at the next station after one unit of time is $\frac{1}{2}$, since the arrival process is geometric.
The state of system $POLLING$ is described by three variables. Variable $station$ contains the location of the server. If the boolean variable $moving$ has value true the server is travelling between two stations. The total function $buffer$ holds the number of packets waiting at each station. Initially, the server is at station 1 and stationary. Also, all buffers are empty.
In the procedures section the joint arrivals and the departures are defined. The joint arrivals are the product of the arrivals of all non-full buffers. Let $W = buffer^{-1}[0..CAPACITY - 1]$ be the set of stations having space available in their buffer, and $S \subseteq W$ a set of stations at which packets arrive. Then the probability of arrivals at exactly all stations in S equals

$$\left(\frac{1}{10}\right)^{\mathsf{card}.S} * \left(\frac{9}{10}\right)^{\mathsf{card}.(W-S)} .$$

This leads to the definition of $arrivals$ in Figure 2. The expression $r \lhd s$ denotes relational overwriting of relation r by relation s. A departure, i.e. service completion, takes place with probability $\frac{1}{4}$ if the buffer at

station *station* (the location of the server) *buffer.station* is non-empty. Otherwise the server idles. Departure is simpler than arrival since only one station is served at a time.
There are two actions *serve* and *walk* which represent the two tasks of the server. It can either service a station or move to the next station, the choice between the two actions being nondeterministic. Nondeterminism is convenient as it is not yet clear what is the right strategy. At the beginning of each time slot arrivals are dealt with. Remember that execution of an action represents what may happen in one unit of time. The actual servicing of packets in action *serve* is described by program *departure*. If the server is moving it arrives with probability $\frac{1}{2}$ at the next station, and *moving* is set to false. With probability $\frac{1}{2}$ it continues moving towards that station.
Costs are specified as real numbers. For each packet waiting in any queue a cost of 1.0 is incurred, in total,

$$\sum s \in STATION \bullet buffer.s \ .$$

The system also incurs a cost of 1.0 per unit of time for moving between stations. System *POLLING* is live which can be shown as outlined at the end of Section 4. Liveness can also be deduced by observing that action *walk* may occur in any possible state in both systems.
The cost structure of system *POLLING* is set up so that small queues are preferred as well as stationarity of the server. Each packet waiting in some buffer causes a cost of one. A cost of one is also incurred each time period during which the which the server is walking. Serving itself causes no cost.
In [Hal01] a larger case study modelling lift system is presented which makes more use of the nondeterministic features of probabilistic action systems in order to derive an optimal control program.

A Smaller System

System *POLLING* has a large state space. In this section we use state aggregation to reduce its size. In the model *POLLING* of Figure 2 a server walks from station to station serving packets. In the following we present an alternative model where the stations walk and the server remains at a fixed location. Using Theorem 4.4 we show that both systems are equivalent, and subsequently compare their dimensions.
Figure 3 describes the system with the fixed server. It always resides at location 1. Consequently no variable is needed to record its position. Arrivals of new packets take place at all locations that have space available in their buffer. Departures only occur from station 1, the location of the server. Program *rotate* models the walking stations. Each station is replaced with its successor within the cyclic arrangement.
Variable *station* in system *POLLING* represents the location of the server. In system *REDPOLL* the location of the server is 1 and does not change. So, relative to the location of the server in both systems, station 1 in system *REDPOLL* corresponds to the station identified by variable *station* in system *POLLING*. In general, station s in system *REDPOLL* corresponds to station $\langle\!\langle s + station \rangle\!\rangle$, where

$$\langle\!\langle x \rangle\!\rangle \ \widehat{=} \ ((x-2) \bmod STATIONS) + 1 \ .$$

We note that the function

$$(\lambda\, s \in STATION \bullet \langle\!\langle s + station \rangle\!\rangle)$$

is bijective for a fixed $station \in STATION$. It relocates stations from its range to corresponding stations in its domain. Based on the function $\langle\!\langle \cdot \rangle\!\rangle$ we define a simulation $sim \in \mathcal{M}(\Gamma_{POLLING}, \Gamma_{REDPOLL})$ by

$$\begin{array}{lcl} sim & = & mov := moving \parallel \\ & & loc := (\lambda\, s \in STATION \bullet buffer.\langle\!\langle s + station \rangle\!\rangle) \ . \end{array}$$

Program *sim* copies the contents buffered at the stations of system *POLLING* to the corresponding stations of system *REDPOLL*.
The two probabilistic action systems *POLLING* and *REDPOLL* are both live. Based on equivalence Theorem 4.4 using state relation *sim* system *POLLING* can easily be related to system *REDPOLL*. The only difference is that instead of the walking server we deal with walking stations.
We seek an optimal implementation for system *POLLING*. Because the systems are equivalent we only need to analyse system *REDPOLL* which has a smaller state space.
We can easily calculate the sizes of the state spaces $\Gamma_{POLLING}$ and $\Gamma_{REDPOLL}$:

$$\mathsf{card}.\Gamma_{POLLING}$$

```
system REDPOLL
constants
  STATIONS; CAPACITY;
constraints
  STATIONS > 0 ∧ CAPACITY > 0;
sets
  STATION = 1 .. STATIONS;
  NEXT = (λ s ∈ 1 .. (STATIONS − 1) • s + 1) ∪ {STATIONS ↦ 1};
variables
  loc ∈ STATION → 0 .. CAPACITY;
  mov ∈ 𝔹;
programs
  arrival = // arrival of packets at locations
    ⨁ S ∈ ℙ(loc⁻¹[0 .. CAPACITY − 1])
      | (1/10)^card(S) * (9/10)^card(loc⁻¹[0..CAPACITY−1]−S) •
        loc := loc ⩤ (λ s ∈ S • loc.s + 1);
  departure = // departure of processed packets from location 1
    | loc.1 > 0 |;
      (loc := loc ⩤ {1 ↦ loc.1 − 1} 0.25⊕ skip)
    ⊔
    | loc.1 = 0 |;
  rotate = // rotate locations
    loc := (λ s ∈ STATION • loc.(NEXT.s));
initialisation
  mov := false ∥ loc := STATION × {0};
actions
  serve =
    | ∑ s ∈ STATION • loc.s |;
    | ¬ mov |; arrival; departure;
  walk =
    | 1.0 |;
    | ∑ s ∈ STATION • loc.s |;
    arrival;
    (mov := true 0.5⊕ (mov := false ∥ rotate));
end
```

Fig. 3. Another cyclic polling system

$$
\begin{aligned}
&= \mathsf{card}.STATION \\
&\quad * (\mathsf{card}.(0 \,..\, CAPACITY))^{\mathsf{card}.STATION} \\
&\quad * \mathsf{card}.\mathbb{B} \\
&= STATIONS * (CAPACITY + 1)^{STATIONS} * 2
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{card}.\Gamma_{REDPOLL} \\
&= (\mathsf{card}.(0 \,..\, CAPACITY))^{\mathsf{card}.STATION} * \mathsf{card}.\mathbb{B} \\
&= (CAPACITY + 1)^{STATIONS} * 2
\end{aligned}
$$

It is not difficult to convince oneself that the set of reachable states of **POLLING** equals the whole state space $\Gamma_{POLLING}$, and that of **REDPOLL** equals $\Gamma_{REDPOLL}$. Obviously the simple relationship

$$\mathsf{card}.(\Gamma_{POLLING}) \;=\; STATIONS * \mathsf{card}.(\Gamma_{REDPOLL})$$

holds between the two systems. In other words, the state space of system *REDPOLL* is linearly smaller in the number of stations than that of system *POLLING*. The probabilistic choice in both arrivals is exponential.

STATIONS	*CAPACITY*	val.**REDPOLL**
5	2	7.7746
6	2	9.7468
7	2	11.7312
8	2	13.7222
9	2	15.7170

Table 1. Values of some instances of system *REDPOLL*

There are about $2^{STATIONS}$ alternatives to consider. So in analysing system *REDPOLL* instead of system *POLLING* there is about $STATIONS * 2^{STATIONS}$ less effort.

An Optimal Implementation

We calculate optimal implementations for systems with $CAPACITY = 2$ and a variety of values of *STATIONS* (see table 1) to increase our confidence in the resulting optimal system. All optimal average-cost values and optimal implementations have been calculated using the software tool mentioned in Section 4. The optimal guards of *serve* and *walk* have been constructed by our tool. The tool accepts system **REDPOLL** with the corresponding values for *CAPACITY* and *STATIONS* inserted as input, and calculates the optimal average-cost value and all states for which the guards of the optimal implementation must be true. The tool can deal with more complicated systems. However, the answer produced by the tool can be difficult to interpret.
We believe that it would be difficult to construct the optimal guards by hand so that the tool is adding real value.
We present an optimal implementation for the reduced system *REDPOLL* with parameters $STATIONS = 5$ and $CAPACITY = 2$ as calculated by our tool. All we need is a guard $\mid gd_{serve} \mid$ for action *serve*. We can use its negation as the guard in action *walk*. We have:

$$gd_{serve} \;\widehat{=}\; loc.1 = 0 \Rightarrow gd_0 \wedge gd_1 \wedge gd_2 \ .$$

The predicates gd_0, gd_1, and gd_2 are defined and explained below. From the precedent $loc.1 = 0$ of implication gd_{serve} we gather that the optimal server always serves when the buffer at station one is not empty. Otherwise one of the three conditions in its antecedent must hold. They treat the three possible values of $loc.2$.

$$\begin{aligned} gd_0 \;\widehat{=}\; & loc.2 = 0 \Rightarrow \\ & \quad loc.3 > 0 \Rightarrow \\ & \quad\quad (loc.3 = 1 \Rightarrow loc.4 = 0 \vee loc.5 = 0) \wedge \\ & \quad\quad (loc.3 = 2 \Rightarrow loc.4 = 0) \end{aligned}$$

If $loc.2$ equals 0, then the server serves if the buffer at station three, $loc.3$, is also empty. Otherwise, if $loc.3 > 0$, we distinguish the two remaining cases for $loc.3$. Observe that the server is more inclined to serve than walk when $loc.3$ is full. This is because rejecting packets does not cause any cost. Remember that the system incurs a cost of 1 for walking from station to station per unit of time.

$$gd_1 \;\widehat{=}\; loc.2 = 1 \Rightarrow loc.3 = 0 \wedge loc.4 = 0$$

If one packet waits in the buffer of station two, the server serves if no packets are waiting at the two immediately following stations.

$$gd_2 \;\widehat{=}\; loc.2 = 2 \Rightarrow loc.3 = 0 \wedge loc.4 = 0 \wedge loc.5 = 0$$

If the buffer of station two is full, all three buffers not mentioned in the guard must be empty. Otherwise the serve walks. This differs from the way station three is treated. The reason is that it is much cheaper to get to station two than to station three. We now present the actions *serve* and *walk* of the optimal system *OPTPOLL*. The rest of the specification is identical to system *REDPOLL*. Action *OPTPOLL.serve* simply adjoins gd_{serve} to the existing guard:

$$\begin{aligned} & OPTPOLL.serve \;= \\ & \quad \mid \textstyle\sum s \in STATION \bullet loc.s \mid; \\ & \quad \mid \neg\, mov \wedge gd_{serve} \mid; arrival; departure; \end{aligned}$$

The guard of action *OPTPOLL.walk* is the negation of $\neg\, mov \wedge gd_{serve}$,

$$
\begin{array}{l}
OPTPOLL.walk \;\; = \\
\quad |\; 1.0 \;|; \\
\quad |\; \sum s \in STATION \bullet loc.s \;|; \\
\quad |\; mov \vee \neg\, gd_{serve} \;|; arrival; \\
\quad (mov := \mathsf{true} \;{}_{0.5}\!\oplus\; (mov := \mathsf{false} \parallel rotate));
\end{array}
$$

System *OPTPOLL* is also an optimal implementation of the pictured systems with six, seven, eight and nine stations. This means in systems with more than five stations packages waiting at stations not mentioned in the guard have no influence on the behaviour of the optimal server. The reason is that the server expects a packet to arrive at a near station. Compared to the cost of waiting, travelling to a far away station would appear expensive. The optimal implementation *OPTPOLL* is a refinement of *REDPOLL*, and hence a refinement of *POLLING*. This is a consequence of the following theorem:

Theorem 5.1 Strengthening guards is a refinement.
Proof This a special case of the refinement proof rule 4.3 with $M = \mathsf{skip}$.

System *POLLING* contains explicit cost statements to specify the cost associated with packets waiting in some buffer, and the cost associated with moving between stations. In fact, there is an implicit cost objective specifying that rejection of packets is free. Although the original model *POLLING* has all operational features that we would expect of a finite-buffer system, it might not express our performance objectives properly. If we wanted to associate a cost of, say, one with each rejection we would need unbounded counters to represent the number of rejections. This cannot be analysed in our approach. In [Sen99] algorithms are presented to solve optimisation problems of this kind for system with countable state spaces. The algorithms work by solving a sequence of finite systems. Dealing with infinite state is an extension of our work which we leave for further research.

6. Conclusion

We have proposed a formalism that supports the two notions of refinement and performance. This is achieved by the combined use of probabilistic and nondeterministic statements in the underlying relational probabilistic language. The formalism is suitable for the modelling of typical dynamic programming problems which employ Markov decision processes. We have used refinement to prove equivalence between such systems. Conventionally, this is done by informal argumentation. Refinement is a convenient framework for such proofs. In addition, it adds to the certainty that the optimization problem solved is the intended one.
Having a language at our disposal that allows us to mix freely probabilistic and nondeterministic languages also helps in expressing more complex system and analysing them [Hal01]. In conjunction with our software tool we can explore optimal implementations more efficiently than if in a parameterized approach was used.
The probabilistic action system notation is close to the B notation. This makes it is easier to use for developers who use the B notation than, for instance, process algebras or Petri nets. Like B our formalism has a rich data language including sets, relations, etc. All sets used must be finite though. This restriction ensures the existence of optimal implementations. These implementations can be computed by dynamic programming algorithms and we have developed a prototype tool to achieve this. Hereby the mathematical properties of probabilistic programs are exploited, e.g. the correspondence between sequential composition and matrix multiplication. Probabilistic programs P, Q, and R have the following algebraic property [Hal01]:

$$P \;{}_{p}\!\oplus\; (Q \bigsqcup R) \;\; = \;\; (P \;{}_{p}\!\oplus\; Q) \bigsqcup (P \;{}_{p}\!\oplus\; R)\;.$$

This allows us to compute an optimal solution of any program before it is executed, hence, we can use dynamic programming. This property is not shared by probabilistic predicate transformers [MMS96].
We intend to use probabilistic action systems when applying conventional refinement. Their use can contribute to finding good implementations of a system, and thus contribute to enhance the conventional notion of state-based refinement. The precise relationship between the two approaches remains to be investigated.

References

[Abr96] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[AM98] Jean-Raymond Abrial and Louis Mussat. Introducing dynamic constraints in B. In Didier Bert, editor, *B'98 : The 2nd International B Conference*, volume 1393 of *Lecture Notes in Computer Science*, pages 83–128. LIRRM Laboratoire d'Informatique, de Robotique et de Micro-électronique de Montpellier, Springer Verlag, 1998.

[BB89] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. In P. H. J. van Eijk, C. A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 23–73. Elsevier Science Publishers North-Holland, 1989.

[BCSS98] M. Bernardo, W.R. Cleaveland, S.T. Sims, and W.J. Stewart. TwoTowers: A tool integrating functional and performance analysis of concurrent systems. In S. Budkowski, A. Cavalli, and E. Najm, editors, *Proc. of the IFIP Joint Int. Conf. on Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing, and Verification (FORTE/PSTV 1998)*, pages 457–467, Paris, France, 1998. Kluwer.

[BDG98] Marco Bernardo, Lorenzo Donatiello, and Roberto Gorrieri. A formal approach to the integration of performance aspects in the modeling and analysis of concurrent systems. *Information and Computation*, 144(2):83–154, 1998.

[Ber97] Marco Bernardo. An algebra-based method to associate rewards with EMPA terms. In Pierpaolo Degano, Robert Gorrieri, and Alberto Marchetti-Spaccamela, editors, *Automata, Languages and Programming, 24th International Colloquium*, volume 1256 of *Lecture Notes in Computer Science*, pages 358–368, Bologna, Italy, 1997. Springer-Verlag.

[Buc94] Peter Buchholz. On a markovian process algebra. Technical Report 500, Universität Dortmund, Fachbereich Informatik, 1994.

[BvW94] Ralph-Johan Back and Joakim von Wright. Trace refinement of action systems. Reports on Mathematics and Computer Science 153, Åbo Akademi, 1994.

[BvW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[GH94] S. Gilmore and J. Hillston. The PEPA workbench: A tool to support a process algebra-based approach to performance modelling. In G. Haring and G. Kotsis, editors, *Proceedings of the Seventh International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, volume 794 of *LNCS*, pages 353–368. Springer-Verlag, 1994.

[Hal01] Stefan Hallerstede. *Performance-Oriented Refinement*. PhD thesis, University of Southampton, 2001.

[Hav98] Boudewijn R. Haverkort. *Performance of Computer Communication Systems: A Model-based Approach*. John Wiley & Sons, 1998.

[HB99] Stefan Hallerstede and Michael Butler. Refinement of dynamic systems. Technical Report DSSE-TR-99-8, University of Southampton, 1999.

[Heh] Eric Hehner. Probabilistic predicative programming. Private Communication.

[HHK+00] H. Hermanns, U. Herzog, U. Klehmet, V. Mertsiotakis, and M. Siegle. Compositional performance modelling with the TIPPtool. *Performance Evaluation*, 39:5–35, 2000.

[Hil96] Jane Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

[HN96] Boudewijn R. Haverkort and Ignas C. Niemegeers. Performability modelling tools and techniques. *Performance Evaluation*, 25:17–40, 1996.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[HR94] Holger Hermanns and Michael Rettelbach. Syntax, semantics, equivalences, and axioms for MTIPP. In U. Herzog and M. Rettelbach, editors, *Proceedings of the Second Workshop on Process Algebra and Performance Modelling*, pages 71–88, 1994.

[HRW95] Holger Hermanns, Michael Rettelbach, and Thorsten Weiss. Formal characterisation of immediate actions in SPA with nondeterministic branching. *The Computer Journal*, 38(7):530–541, 1995.

[HT93] Boudewijn R. Haverkort and Kishor S. Trivedi. Specification techniques for markov reward models. *Discrete Event Dynamic Systems: Theory and Applications*, 3:219–247, 1993.

[Jos88] Mark B. Josephs. A state-based approach to communicating processes. *Distributed Computing*, 3:9–18, 1988.

[JSM97] He Jifeng, K. Seidel, and A. McIver. Probabilistic models for the guarded command language. *Science of Computer Programming*, 28(2–3):171–192, 1997.

[Kan92] Krishna Kant. *Introduction to Computer System Performance Evaluation*. McGraw Hill, New York, 1992.

[KS76] John G. Kemeny and J. Laurie Snell. *Finite Markov Chains*. Undergraduate Texts in Mathematics. Springer-Verlag, 1976.

[Lin98] Christoph Lindemann. *Performance Modelling with Deterministic and Stochastic Petri Nets*. Wiley, 1998.

[MCB84] Marco Ajmone Marsan, Gianni Conte, and Gianfranco Balbo. A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. *ACM transactions on computer systems*, 2:93–122, 1984.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mit98] Isi Mitrani. *Probabilistic Modelling*. Cambridge University Press, 1998.

[MMS96] Carroll Morgan, Annabelle McIver, and Karen Seidel. Probabilistic predicate transformers. *ACM Transactions on Programming Languages and Systems*, 18(3):325–353, 1996.

[Mol82] Michael K. Molloy. Performance analysis using stochastic petri nets. *IEEE Transactions on Computers*, C-31(9):913–917, September 1982.

[Nel95] Randolph Nelson. *Probability, Stochastic Processes, and Queueing Theory: The Mathematics of Computer Performance Modelling*. Springer, 1995.

[Put94] Martin L. Puterman. *Markov Decision Processes – Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Mathematical Statistics. Wiley, 1994.
[Rei85] Wolfgang Reisig. *Petri Nets. An Introduction*, volume 3 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.
[Sei95] Karen Seidel. Probabilistic communicating processes. *Theoretical Computer Science*, 152:219–249, 1995.
[Sen99] Linn I. Sennott. *Stochastic Dynamic Programming and the Control of Queueing Systems*. Wiley Series in Probability and Statistics. Wiley, 1999.
[SMM97] Karen Seidel, Carroll Morgan, and Annabelle McIver. Probabilistic imperative programming: a rigorous approach. Technical report, University of Oxford, PRG, 1997.
[ST96] Kaisa Sere and Elena Troubitsyna. Probabilities in action systems. In *Proc. of the 8th Nordic Workshop on Programming Theory*, 1996.
[STP96] Robin Sahner, Kishor S. Trivedi, and Antonio Puliafito. *Performance and reliability analysis of computer systems: an example-based approach using the SHARPE software package*. Kluwer Academic Publishers, 1996.
[Tij94] Henk C. Tijms. *Stochastic Models: An Algorithmic Approach*. Wiley Series in Probability and Mathematical Statistics. Wiley, 1994.
[Tro99] Elena Troubitsyna. Enhancing Dependability via Parameterized refinement. In *Proc. of Pacific Rim International Symposium on Dependable Computing*, 1999.
[WD96] Jim Woodcock and Jim Davies. *Using Z*. Prentice Hall International, 1996.

A. Sequential Composition of Probabilistic Programs

We use deterministic implementations of a program to define sequential composition. Let $P \in \mathcal{P}(\Gamma, \Gamma')$. An implementation consists of a function $M \in \Gamma \to \mathbb{D}\,\Gamma'$ and a cost function C.
For a probabilistic state $f \in \mathbb{D}\,\Gamma$ and a function $M \in \Gamma \to \mathbb{D}\,\Gamma'$ their product $f * M \in \mathbb{D}\,\Gamma'$ is defined by

$$(f * M).\tau' \;\widehat{=}\; \textstyle\sum_{\tau \in \mathsf{car}.f} f.\tau * M.\tau.\tau' \ .$$

The product $f * M$ is a probabilistic state:

Theorem A.1 $f * M \in \mathbb{D}\,\Gamma$.
Proof The carrier of $f * M$ is finite because the carrier of f is finite. Furthermore,

$$\begin{array}{ll} & \sum_{\tau' \in \Gamma'} (f * M).\tau' \\ = & \sum_{\tau' \in \Gamma'} \sum_{\tau \in \Gamma} f.\tau * M.\tau.\tau' \\ = & \sum_{\tau \in \Gamma} f.\tau * \sum_{\tau' \in \Gamma'} M.\tau.\tau' \\ = & \sum_{\tau \in \Gamma} f.\tau * 1 \\ = & 1 \ . \end{array}$$

Relation fun which maps P to its deterministic implementations (C, M) is defined by

$$\mathsf{fun}.P.(C, M) \;\widehat{=}\; \forall\, \tau \in \mathsf{dom}.P \bullet (C.\tau, M.\tau) \in P.\tau \ .$$

Hence the set $\mathsf{fun}.P$ is a subset of $(\mathsf{dom}.P \to \mathbb{R}_{\geq 0}) \times (\mathsf{dom}.P \to \mathbb{D}\,\Gamma')$. We define sequential composition of $P \in \mathcal{P}(\Gamma, \Gamma')$ and $Q \in \mathcal{P}(\Gamma', \Gamma'')$ by

$$\begin{array}{lll} (P; Q).\tau.(d, g) & \widehat{=} & \exists (c, f) \in P.\tau,\ (C, M) \in \mathsf{fun}.Q \bullet \mathsf{car}.f \subseteq \mathsf{dom}.Q \ \wedge \\ & & \quad d = c + f * C \ \wedge\ g = f * M \ . \end{array}$$

It is executed by picking all probabilistic states f from $P.\tau$ in turn for each initial state τ. Then the weighted average of the probabilistic states $M.\tau' \in Q.\tau'$ for all $\tau' \in \mathsf{car}.f$ is taken with the weights being the probabilities $f.\tau'$. The weighted average $f * M$ can be interpreted as the expected state that is reached if each probabilistic state $M.\tau$ is chosen with probability $f.\tau$. If there is an intermediate state $\tau' \in \mathsf{car}.f$ which Q cannot map anywhere, execution of probabilistic state f is blocked. We note that Q can map each state from $\mathsf{car}.f$ to some probabilistic state if and only if $\mathsf{car}.f \subseteq \mathsf{dom}.Q$. The value d is the expected cost associated with a sequential execution of P and Q. Cost c is incurred when f is chosen as the successor probabilistic state of τ. The expected cost incurred afterwards depends on the intermediate probabilistic state f. It equals $f * C$, where the costs $C.\tau'$ associated with $M.\tau'$ are weighed according to the probability $f.\tau'$, that τ' occurs as an intermediate state.

Based on the sequential composition of probabilistic programs we define a lifted version of probabilistic program whose domain a probablilistic states:

$$\widehat{P}.g.(c,f) \;\widehat{=}\; \mathsf{car}.g \subseteq \mathsf{dom}.P \wedge \exists (C,M) \in \mathsf{fun}.P \bullet c = g * C \wedge f = g * M \ .$$

Program $\widehat{P}$ yields a probabilistic successor state of a probabilistic state g. Formally, g corresponds to an intermediate probabilistic state in the definition of sequential composition.

B. Relational Programs

A relation R from Γ to Γ' is called a *relational program*. We denote the set of all relational program from Γ to Γ' by $\mathcal{R}(\Gamma,\Gamma')$. Sequential composition of relational program $R \in \mathcal{R}(\Gamma,\Gamma')$ and relational program $S \in \mathcal{R}(\Gamma',\Gamma'')$ is defined by

$$(R;S).\tau.\tau'' \;\widehat{=}\; \exists \tau' \in \Gamma' \bullet R.\tau.\tau' \wedge S.\tau'.\tau'' \ .$$

Relational programs are embedded into probabilistic programs by a function $\Uparrow \in \mathcal{R}(\Gamma,\Gamma') \to \mathcal{P}(\Gamma,\Gamma')$, defined by:

$$(\Uparrow R).\tau.(c,f) \;\widehat{=}\; c = 0 \wedge (\exists \tau' \in R.\tau \bullet f = \chi.\tau') \ ,$$

where for a state space Γ and $\tau \in \Gamma$ the point density $\chi.\tau \in \mathbb{D}\,\Gamma$ is defined by $\chi.\tau.\tau = 1$ and $\chi.\tau.\tau' = 0$ for all $\tau' \neq \tau$.
The function $\Uparrow$ is a homomorphism with respect to sequential composition. Consequently, for non-probabilistic programs it behaves as in the relational semantics. This is important for standard nondeterministic programs to keep their usual semantics in the extended formalism.

Theorem B.1 $\Uparrow(R;S) \;=\; (\Uparrow R);(\Uparrow S)$.
Proof

$$\begin{array}{ll}
& (0,f) \in (\Uparrow(R;S)).\tau \\
\Leftrightarrow & \exists \tau'' \bullet (R;S).\tau.\tau'' \wedge f = \chi.\tau'' \\
\Leftrightarrow & \exists \tau'', \tau' \bullet R.\tau.\tau' \wedge S.\tau'.\tau'' \wedge f = \chi.\tau'' \\
\Leftrightarrow & \exists \tau'', \tau', g \bullet (0,g) \in (\Uparrow R).\tau \wedge g = \chi.\tau' \wedge S.\tau'.\tau'' \wedge f = \chi.\tau'' \\
\Leftrightarrow & \exists \tau'', \tau', g, M \bullet (0,g) \in (\Uparrow R).\tau \wedge g = \chi.\tau' \wedge (\mathbf{0},M) \in \mathsf{fun}.(\Uparrow S) \wedge M.\tau' = \chi.\tau'' \wedge f = \chi.\tau'' \\
\Leftrightarrow & \exists \tau', g, M \bullet (0,g) \in (\Uparrow R).\tau \wedge g = \chi.\tau' \wedge (\mathbf{0},M) \in \mathsf{fun}.(\Uparrow S) \wedge \tau' \in \mathsf{dom}.(\Uparrow S) \wedge M.\tau' = f \\
\Leftrightarrow & \exists g, M \bullet (0,g) \in (\Uparrow R).\tau \wedge (\mathbf{0},M) \in \mathsf{fun}.(\Uparrow S) \wedge \mathsf{car}.g \subseteq \mathsf{dom}.(\Uparrow S) \wedge 0 = 0 + f * \mathbf{0} \wedge f = g * M \\
\Leftrightarrow & (0,f) \in ((\Uparrow R);(\Uparrow S)).\tau \ .
\end{array}$$

C. Proof of theorem 4.3

Let $\mathbf{A} = (\Gamma_A, I, P)$ and $\mathbf{C} = (\Gamma_C, J, Q)$ be probabilistic action systems, and let $M \in \mathcal{P}(\Gamma_A,\Gamma_C)$ be a probabilistic state relation satisfying (1) to (3). Lemma C.1 establishes the simulation relationship for traces:

Lemma C.1 For all $t \in \mathrm{seq}(\mathbb{R}_{\geq 0})$ and $f \in \mathbb{D}\,\Gamma_C$:

$$\mathsf{path}.\mathbf{C}.t.f \;\Rightarrow\; (\exists\, g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.t.g \wedge \{(0,f)\} = \widehat{M}.g) \ .$$

To prove theorem 4.3 we have to show $\mathbf{A} \sqsubseteq \mathbf{B}$, i.e. the inclusion of the traces and the impasses.
Trace inclusion. Let t be a trace.

$$\begin{array}{ll}
& \mathsf{tr}.\mathbf{C}.t \\
\Leftrightarrow & \exists f : \mathbb{D}\,\Gamma_C \bullet \mathsf{path}.\mathbf{C}.t.f \\
\Rightarrow & \exists f : \mathbb{D}\,\Gamma_C,\ g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.t.g \wedge \{(0,f)\} = \widehat{M}.g \\
\Leftrightarrow & \exists g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.t.g \wedge (\exists f : \mathbb{D}\,\Gamma_C \bullet \{(0,f)\} = \widehat{M}.g) \\
\Rightarrow & \mathsf{tr}.\mathbf{A}.t \ .
\end{array}$$

Impasse inclusion. Let t be a trace.

$$
\begin{aligned}
&\mathsf{im}.\mathbf{C}.t \\
&\quad\Leftrightarrow\ \exists f : \mathbb{D}\,\Gamma_C \bullet \mathsf{path}.\mathbf{C}.t.f \wedge \widehat{Q}.f = \varnothing \\
&\quad\Rightarrow\ \exists f : \mathbb{D}\,\Gamma_C,\ g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.t.g \wedge \{(0,f)\} = \widehat{M}.g \wedge \widehat{Q}.f = \varnothing \\
&\quad\Rightarrow\ \exists g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.t.g \wedge (\widehat{M}.g); Q = \varnothing \\
&\quad\Leftrightarrow\ \exists g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.t.g \wedge (\widehat{M; Q}).g = \varnothing \\
&\quad\Rightarrow\ \exists g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.t.g \wedge \widehat{P}.g = \varnothing \qquad (2) \\
&\quad\Leftrightarrow\ \mathsf{im}.\mathbf{A}.t\ .
\end{aligned}
$$

Proof of lemma C.1

Let $t \in \mathrm{seq}(\mathbb{R}_{\geq 0})$ and $f \in \mathbb{D}\,\Gamma_C$. The proof is by induction on the length of trace t.

$$
\begin{aligned}
&\mathsf{path}.\mathbf{C}.\langle\,\rangle.f \\
&\quad\Leftrightarrow f \in J \\
&\quad\Rightarrow \{(0,f)\} \in I; M \qquad (1) \\
&\quad\Leftrightarrow \exists g : \mathbb{D}\,\Gamma_A \bullet g \in I \wedge \{(0,f)\} = \widehat{M}.g \\
&\quad\Leftrightarrow \exists g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.\langle\,\rangle.g \wedge \{(0,f)\} = \widehat{M}.g\ .
\end{aligned}
$$

Let $t = s^\frown\langle c\rangle$:

$$
\begin{aligned}
&\mathsf{path}.\mathbf{C}.t.f \\
&\quad\Leftrightarrow\ \exists h : \mathbb{D}\,\Gamma_C \bullet \mathsf{path}.\mathbf{C}.s.h \wedge (c,f) \in \widehat{Q}.h \\
&\quad\Rightarrow\ \exists h : \mathbb{D}\,\Gamma_C,\ g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.s.g \wedge \{(0,h)\} = \widehat{M}.g \wedge (c,f) \in \widehat{Q}.h \\
&\quad\Leftrightarrow\ \exists g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.s.g \wedge (c,f) \in (\widehat{M}.g); Q \\
&\quad\Leftrightarrow\ \exists g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.s.g \wedge (c,f) \in (\widehat{M; Q}).g \\
&\quad\Rightarrow\ \exists g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.s.g \wedge (c,f) \in (\widehat{P; M}).g \qquad (3) \\
&\quad\Leftrightarrow\ \exists g : \mathbb{D}\,\Gamma_A,\ h : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.s.g \wedge (c,h) \in \widehat{P}.g \wedge \{(0,f)\} = \widehat{M}.h \\
&\quad\Leftrightarrow\ \exists h : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.t.h \wedge \{(0,f)\} = \widehat{M}.h\ .
\end{aligned}
$$