Refinement of dynamic systems

Stefan Hallerstede and Michael Butler
{shl,mjb}@ecs.soton.ac.uk

Declarative Systems and Software Engineering Group
Technical Report DSSE-TR-99-8
November 29 1999

www.dsse.ecs.soton.ac.uk/techreports/

Department of Electronics and Computer Science
University of Southampton
Highfield, Southampton SO17 1BJ, United Kingdom

Abstract

Existing refinement frameworks such as B allow a developer to specify a system on an abstract level. Subsequently, this abstract specification is refined into an implementation that performs the specified task. In this paper a conventional refinement approach is extended with a means for performance analysis. This new approach guides a developer towards well-performing implementations throughout the refinement process.

The main achievement of this work is an elaboration of a connection between performance and trace refinement.

1 Introduction

Refinement theory [Abr96a, BvW98] is focused on preserving correctness properties in the stepwise-development approach to reactive systems. The range of correct implementations includes programs that perform well in the intended environment all of which are not distinguished from poorly-performing programs. If performance analysis is required for some reactive system, it is done outside the formal apparatus of refinement. In queueing theory [Mit98, Sen99] performance is a fundamental and important notion. To integrate such a notion of performance into refinement theory of reactive systems, probabilistic behaviour has to be modelled within that theory. The advantage of having an integrated approach to refinement including both correctness and performance is that a system developer can be guided throughout the refinement process towards a well performing reactive system. At each refinement step it can be verified if the performance objective is achieved.

There is a variety of formalisms for performance analysis [BDG94, Hil93, MCB84, Mit98, STP96]. By and large their semantics is based on Markov processes and Markov reward processes. All of these formalisms, however, are applied on either an implementation level or a high level abstraction of the original system, without a formal relationship between the levels. The lack of such a formal relationship makes it difficult to compare different models of some system with respect to their performance. In stochastic process algebras, i.e. [Hil93], systems can be compared using equivalence. This basically means that one always remains on the same abstraction level. The other formalisms use Petri nets, automata and queueing systems. Usually informal arguments are applied to compare alternative models of a system. The majority of these formalisms uses a continuous time representation. Sennott [Sen99] argues that in connection with computers, discrete time is often appropriate even though the underlying time domain is continuous. We follow [Sen99] and use discrete time.

Action systems and similar formalisms [Abr96a, BKS88, CM88, Hoa85] do support a notion of refinement. These formalisms have been used in the design of reactive systems [Abr96b, BSS96, Sim94]. The lack of support for performance analysis has been recognised as a problem in [BSS96]. The main ingredient missing is a means to express randomness in the behaviour of a physical environment of a reactive system. Accordingly, we begin by introducing a probabilistic program model.

In the conventional relational program model, states are mapped to sets of states. This model supports a notion of nondeterminism which is based on the view that for each initial state the program chooses a state from the corresponding set of final states. There are different ways to add probabilistic capabilities to this model while retaining a compatible notion of nondeterminism. We present a probabilistic model which we call the vector model. It maps initial states to sets of state densities. Technically, the vector model is defined similar to the probabilistic model proposed in [JSM97]. The difference is that in [JSM97] nondeterminism is regarded as generalised form of probabilistic choice. In the model presented here both concepts are kept seperate (see section 7).

Having defined a probabilistic program model we are ready to state our model for reactive systems that is capable of expressing performance objectives. We call these systems dynamic systems. They are defined in two stages. First, probabilistic action systems are introduced. They model reactive systems operating in probabilistic environments. The behaviour of a probabilistic action system is described by the traces the system may undergo.

In the second step, probabilistic action systems are amended with cost functions. Cost functions stem from the theory of stochastic dynamic programming [Put94, Sen99]. They are used to encode performance objectives. We call probabilistic action systems with cost functions dynamic systems. The way in which the performance objectives should be encoded depends on the criterion used for performance evaluation. The approach followed here assumes that performance is measured in terms of long-run expected average cost per unit time.

The behaviour of a dynamic system is described by traces of state densities and expectations. Expectations associate each trace with costs that are derived from the cost function of the system.

We have decided to define probabilistic action systems seperately to be able to identify properties concerning performance. This would enable us to reason about different performance criteria without changing the whole theory.

The contribution of this paper is to create a connection between refinement of reactive systems and performance analysis. As a by-product we propose a relational model for probabilistic programs tailored to our needs.

2 Relational programs

Programs can be modelled in a variety of ways. The advantage of the relational model is that programs in it are described operationally. The material in this section is mainly based on [BvW98].

In the relational model a program maps each state of its domain to a set of states. States are elements of countable sets called state spaces Γ which are defined by products of the form

$$\Gamma = \prod_{i=1}^{n} \Delta_i, \quad n \geq 1 \ .$$

The projections $\pi_i : \Gamma \to \Delta_i$, $i = 1, \ldots, n$, are called variables,

$$\mathsf{var}.\Gamma \mathrel{\widehat{=}} \{\pi_1, \pi_2, \ldots, \pi_n\} \ .$$

State spaces are generally denoted by capital greek letters Γ, Σ, Δ . Predicates on Γ, denoted $\mathbb{P}\,\Gamma$, are defined by

$$\mathbb{P}\Gamma \quad \widehat{=} \quad \Gamma \to \mathbb{B} \ .$$

By $\mathbb{B}$ we denote the set of truth values $\{\text{true}, \text{false}\}$. We also use set notation with predicates, for instance $\sigma \in p \mathrel{\widehat{=}} p.\sigma$. Intersection $\cap$ and union $\cup$ of predicates are defined in the usual way by pointwise extension [BvW98].

Relational programs, denoted $\Sigma \leftrightarrow \Gamma$, are defined by

$$\Sigma \leftrightarrow \Gamma \mathrel{\widehat{=}} \Sigma \to \mathbb{P}\,\Gamma \ ,$$

where Σ and Γ are state spaces. The ordering of two relational programs $P, Q : \Sigma \leftrightarrow \Gamma$ is given by

$$P \subseteq Q \quad \widehat{=} \quad \forall \sigma \in \Sigma \bullet P.\sigma \subseteq Q.\sigma \ .$$

The following operations, meet $\cap$, join $\cup$, general meet $\bigcap$, general join $\bigcup$, are defined for relational programs $P, Q : \Sigma \leftrightarrow \Gamma$, and $Z : \mathbb{P}(\Sigma \leftrightarrow \Gamma)$, a set of programs

$$\begin{aligned}
(P \cap Q).\sigma \quad &\widehat{=} \quad P.\sigma \cap Q.\sigma \ , \\
(P \cup Q).\sigma \quad &\widehat{=} \quad P.\sigma \cup Q.\sigma \ , \\
(\bigcap Z).\sigma \quad &\widehat{=} \quad \bigcap\{Q.\sigma \mid Q \in Z\} \ , \\
(\bigcup Z).\sigma \quad &\widehat{=} \quad \bigcup\{Q.\sigma \mid Q \in Z\} \ .
\end{aligned}$$

Sequential composition $P; Q$ of programs $P : \Sigma \leftrightarrow \Delta$ and $Q : \Delta \leftrightarrow \Gamma$ is defined by

$$(P; Q).\sigma \quad \widehat{=} \quad \bigcup\{Q.\epsilon \mid \epsilon \in P.\sigma\} \ .$$

For a predicate $p : \mathbb{P}\,\Gamma$ the guard $|p| : \Gamma \leftrightarrow \Gamma$ is defined by

$$|p|.\sigma \quad \widehat{=} \quad \{\sigma' \mid \sigma' = \sigma \wedge p.\sigma\} \ .$$

For $x \in \mathsf{var}.\Gamma$ and an expression $e : \Gamma \to \mathsf{ran}.x$ [1] the assignment of e to variable x, denoted $x := e : \Gamma \leftrightarrow \Gamma$, is defined implicitly through the projections of Γ

$$\begin{aligned} x.((x := e).\sigma) \quad &\widehat{=} \quad e.\sigma \ , \\ y.((x := e).\sigma) \quad &\widehat{=} \quad y.\sigma \quad \text{for all } y \neq x \ . \end{aligned}$$

If $a \in \mathsf{ran}.x$ is a value we define the pointwise extension $\dot{a} : \Gamma \to \mathsf{ran}.x$ of a by $\dot{a} \mathrel{\widehat{=}} (\lambda\, \tau : \Gamma \bullet a)$. The assignment of a to x is then defined by

$$x := a \quad \widehat{=} \quad x := \dot{a} \ .$$

3 Vector programs

Vector programs preserve all algebraic properties of relational programs. In relational programs nondeterminism is modelled by the multiplicity of final states. In vector programs this is replaced by the multiplicity of state densities. This is where vector programs owe their name. State densities can be regarded as vectors.

State densities model the probabilistic characteristics of vector programs. A *state density* $\mathbb{D}\,\Gamma$ on a state space Γ is defined by[2]

$$\mathbb{D}\,\Gamma \quad \widehat{=} \quad \{f : \Gamma \to \mathbb{R} \mid 0 \leq f \wedge \#\mathsf{car}.f \in \mathbb{N} \wedge \sum_{\tau \in \Gamma} f.\tau = 1\} \ ,$$

where $\mathsf{car}.f$, the *carrier* of a state density f, is defined by

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

As everywhere in this text we use pointwise extension of $\leq$ to interpret $0 \leq f$, i.e. $0 \leq f$ means $\forall\, \tau \bullet 0 \leq f.\tau$. The restriction of densities to functions with finite carrier is required for the existence of the expectations to be defined in subsection 5.2. Another consequence of the use of state densities with finite carrier only is the continuity of sequential composition of vector programs (see subsection 3.1).

The definition of vector programs resembles that of relational programs. A *vector program* maps states to sets of state densities. The set of all vector programs, denoted $\Sigma \Leftrightarrow \Gamma$, is defined by

$$\Sigma \Leftrightarrow \Gamma \quad \widehat{=} \quad \Sigma \to \mathbb{P}\,\mathbb{D}\,\Gamma \ .$$

[1]For a partial function $\phi : S \nrightarrow T$ the domain of ϕ is defined by $\mathsf{dom}.\phi \mathrel{\widehat{=}} \{s \mid (\exists\, t \bullet \phi.s = t)\}$, and the range of ϕ is defined by $\mathsf{ran}.\phi \mathrel{\widehat{=}} \{t \mid (\exists\, s \bullet \phi.s = t)\}$.

[2]Note that $\sum_{\tau \in \Gamma} f.\tau$ is an infinite sum. The requirement to add up to one makes it a probability density.

In section 6 we will be especially interested in programs that use only a finite part of the state spaces. A vector program Q is called *finite* if $\mathsf{dom}.Q$ and all members of $\mathsf{ran}.Q$ are finite. The set of finite vector programs is denoted $\Sigma \Leftrightarrow_F \Gamma$.

Vector programs are ordered similar to relational programs. The ordering of vector programs $Q, R : \Sigma \Leftrightarrow \Gamma$ is defined by

$$Q \subseteq R \quad \widehat{=} \quad \forall \sigma \in \Sigma \bullet Q.\sigma \subseteq R.\sigma \ .$$

A vector program is called *deterministic* if it yields at most one state density for each state in its domain. The set of *deterministic programs in a program* $Q : \Sigma \Leftrightarrow \Gamma$ is defined by

$$\mathsf{det}.Q \quad \widehat{=} \quad \{P \mid P \subseteq Q \wedge (\forall \sigma : \mathsf{dom}.Q \bullet \exists f : \mathbb{D}\Gamma \bullet \{f\} = P.\sigma)\} \ .$$

Note, that all deterministic programs in a program Q have the same domain as Q, and that Q is deterministic iff $\{Q\} = \mathsf{det}.Q$. Also, Q is the join of $\mathsf{det}.Q$.

Proposition 3.1 $\bigcup(\mathsf{det}.Q) = Q$.

Relational programs $Q : \Sigma \leftrightarrow \Gamma$ are embedded into $\Sigma \Leftrightarrow \Gamma$ by the morphism $\mathsf{pr} : (\Sigma \leftrightarrow \Gamma) \to (\Sigma \Leftrightarrow \Gamma)$.

$$\mathsf{pr}.Q.\sigma \quad \widehat{=} \quad \{\chi_{\{\tau\}} \mid \tau \in Q.\sigma\} \ ,$$

where $\chi_{\{\tau\}}$ is the point density with $\chi_{\{\tau\}}.\tau = 1$. In general, we denote the characteristic function of a set S by χ_S. Morphism pr is well-defined, i.e. for all relational programs $Q : \Sigma \leftrightarrow \Gamma$ their embedding $\mathsf{pr}.Q$ is an element of $\Sigma \Leftrightarrow \Gamma$. The well-definedness of the following definitions is easily proven.

It is easy to see that pr is isotonic, i.e.

$$Q \subseteq R \Leftrightarrow \mathsf{pr}.Q \subseteq \mathsf{pr}.R \ .$$

Assignment and guard are defined by the embedding of corresponding relational programs

$$\begin{aligned} x := e \quad &\widehat{=} \quad \mathsf{pr}.(x := e) \ , \\ |p| \quad &\widehat{=} \quad \mathsf{pr}.|p| \ . \end{aligned}$$

Meets and joins are defined analogously to their counterparts in relational programs, e.g. $(P \cap Q).\sigma \mathrel{\widehat{=}} P.\sigma \cap Q.\sigma$.

For $p \in [0, 1]$ probabilistic choice $Q \;{}_p\!\oplus R$ is defined arithmetically giving probability p to branch Q and probability $1 - p$ to branch R

$$(Q \;{}_p\!\oplus R).\sigma \quad \widehat{=} \quad \{p * f + (1 - p) * g \mid f \in Q.\sigma \wedge g \in R.\sigma\} \ ,$$

where addition of functions and multiplication of functions with constants are defined by pointwise extension

$$\begin{aligned} (f+g).\sigma &\mathrel{\widehat{=}} f.\sigma + g.\sigma \ , \\ (p * f).\sigma &\mathrel{\widehat{=}} p * f.\sigma \ . \end{aligned}$$

Sequential composition $Q;R$, for programs $Q : \Sigma \Leftrightarrow \Delta$ and $R : \Delta \Leftrightarrow \Gamma$, picks for each initial state σ all densities f from $Q.\sigma$ in turn and builds the weighted average of all families $(G.\tau \mid \tau \in \mathsf{car}.f)$, where $G.\tau \in R.\tau$. If there is an intermediate state $\tau \in \mathsf{car}.f$ which R cannot map anywhere, then f does not lead to a family of final densities and is thrown away. The families G are taken from the set $\mathsf{fun}.R : \mathbb{P}(\Delta \nrightarrow \mathbb{D}\,\Gamma)$, defined by

$$\mathsf{fun}.R \mathrel{\widehat{=}} \{G \mid \mathsf{dom}.G = \mathsf{dom}.R \wedge (\forall \sigma : \mathsf{dom}.R \bullet G.\sigma \in R.\sigma)\} \ .$$

The families G are just another way to describe the deterministic programs in a program. But they are better suited to define the product (1) below.

Proposition 3.2 $G \in \mathsf{fun}.R \Leftrightarrow (\exists P : \mathsf{det}.R \bullet \{G\} = \mathsf{fun}.P)$.

For a state density $f : \mathbb{D}\,\Delta$ and a family $G : \Delta \nrightarrow \mathbb{D}\,\Gamma$ with $\mathsf{car}.f \subseteq \mathsf{dom}.G$ we define the product $f * G$ by

$$f * G \mathrel{\widehat{=}} \textstyle\sum_{\tau \in \mathsf{dom}.G} f.\tau * G.\tau \ . \tag{1}$$

Proposition 3.3 $f * G \in \mathbb{D}\,\Gamma$.

Proof

We have $\mathsf{dom}.(G.\tau) = \Gamma$ for all $\tau \in \mathsf{dom}.G$, and

$$\begin{aligned} \textstyle\sum_{\sigma\in\Gamma} (f * G).\sigma &= \textstyle\sum_{\sigma\in\Gamma} (\sum_{\tau\in\mathsf{dom}.G} f.\tau * G.\tau).\sigma \\ &= \textstyle\sum_{\sigma\in\Gamma} (\sum_{\tau\in\mathsf{dom}.G} f.\tau * G.\tau.\sigma) \\ &= \text{“absolute convergence of } \textstyle\sum_\sigma \sum_\tau f.\tau * G.\tau.\sigma\text{”} \\ &\ \textstyle\sum_{\tau\in\mathsf{dom}.G} (\sum_{\sigma\in\Gamma} f.\tau * G.\tau.\sigma) \\ &= \text{“product of constant with convergent series”} \\ &\ \textstyle\sum_{\tau\in\mathsf{dom}.G} f.\tau * (\sum_{\sigma\in\Gamma} G.\tau.\sigma) \\ &= \textstyle\sum_{\tau\in\mathsf{dom}.G} f.\tau * 1 \\ &= \text{“}\mathsf{car}.f \subseteq \mathsf{dom}.G\text{”} \\ &\ \textstyle\sum_{\tau\in\Gamma} f.\tau \\ &= 1 \ . \end{aligned}$$

Vector program R can map each state from $\mathsf{car}.f$ to some density iff $\mathsf{car}.f \subseteq \mathsf{dom}.R$. This is exactly the case iff there is a $G \in \mathsf{fun}.R$ with $\mathsf{car}.f \subseteq \mathsf{dom}.G$ because $\mathsf{dom}.G = \mathsf{dom}.R$.

$$(Q;R).\sigma \quad \widehat{=} \quad \{f * G \mid f \in Q.\sigma \wedge G \in \mathsf{fun}.R \wedge \mathsf{car}.f \subseteq \mathsf{dom}.G\} \ .$$

Note that $G.\epsilon$ is a state density and $f.\epsilon$ a real number. Also, the sum $\sum_{\epsilon \in \mathsf{dom}.G} f.\epsilon * G.\epsilon$ is defined pointwise. The following propsition shows that $Q;R$ is indeed a vector program.

Proposition 3.4 $(Q;R) \in \Sigma \Leftrightarrow \Gamma$.

Proof

> If $\sigma \notin \mathsf{dom}.Q$, then $(Q;R).\sigma = \varnothing$. Otherwise, let $f \in Q.\sigma$, $G \in \mathsf{fun}.R$ and $\mathsf{car}.f \subseteq \mathsf{dom}.G$. By proposition 3.3 we have $f * Q \in \mathbb{D}\Gamma$, hence, $(Q;R).\sigma \in \mathbb{P}\mathbb{D}\Gamma$.

The morphism pr is a homomorphism for all operations on $\Sigma \leftrightarrow \Gamma$, in particular

$$\begin{aligned} \mathsf{pr}.(P \cup Q) &= \mathsf{pr}.P \cup \mathsf{pr}.Q \ , \\ \mathsf{pr}.(P;Q) &= \mathsf{pr}.P; \mathsf{pr}.Q \ . \end{aligned}$$

3.1 Recursion

Vector Programs $(\Gamma \Leftrightarrow \Gamma, \subseteq)$ form a complete partial order. The bottom element is $\mathbf{O}$, defined by, $\forall \sigma : \Gamma \bullet \mathbf{O}.\sigma = \varnothing$.

An operation $T : (\Gamma \Leftrightarrow \Gamma) \to (\Gamma \Leftrightarrow \Gamma)$ is called *continuous* if for every ascending chain of programs $(R.n \mid n \in \mathbb{N})$ the equality $T.(\bigcup_{n\in\mathbb{N}} R.n) = \bigcup_{n\in\mathbb{N}} T.(R.n)$ holds. Operations with higher arities are called continuous if they are continuous in each component.

Proposition 3.5 *Each of the vector program operations* $\cup$, $\cap$, $;$, ${}_p\oplus$, $\bigcup$ *is continuous.*[3]

Continuity of sequential composition depends on the finiteness of the carriers of all involved state densities. The other operations do not require carriers to be finite. Note that general join $\bigcap$ is not continuous but, for instance, $T.X = \bigcup_{i\in\mathbb{N}}(x := i \cup X)$ is. The least fixed point of T, denoted $\mu X \bullet T.X$, is defined by

$$\mu X \bullet T.X \quad \widehat{=} \quad \bigcap\{Y \mid T.Y \subseteq Y\} \ .$$

[3] Continuity of $\bigcup$ means that if $(R_n \mid n \in \mathbb{N})$ is an ascending chain of programs, then $\bigcup(Z \cup \{\bigcup\{R_n \mid n \in \mathbb{N}\}\}) = \bigcup\{\bigcup(Z \cup \{R_n\}) \mid n \in \mathbb{N}\}$.

From proposition 3.5 it follows that least fixed points of vector programs can be finitely approximated (see [DP90], CPO Fixpoint Theorem 1).

$$\mu X \bullet T.X \quad = \quad \bigcup_{n \in \mathbb{N}} T^n.\mathbf{O} \ ,$$

where the *n-th approximation* T^n is defined by

$$\begin{array}{rcl} T^0.X & \mathrel{\widehat{=}} & X \ , \\ T^{n+1} & \mathrel{\widehat{=}} & T.(T^n.X) \ . \end{array}$$

Iterates serve to describe vector programs that are run repeatedly for an indefinitely long period of time. In section 4 they are employed to characterise reactive behaviour. For $n \in \mathbb{N}$ the *n-th iterate* of a vector program $Q : \Gamma \Leftrightarrow \Gamma$, denoted Q^n, is defined by

$$\begin{array}{rcl} Q^0 & \mathrel{\widehat{=}} & \mathbf{id} \ , \\ Q^{n+1} & \mathrel{\widehat{=}} & Q^n ; Q \ . \end{array}$$

3.2 Initialisation

In the following sections we use a short notation to initialise program variables before running a program. Any vector program $I : \Gamma \Leftrightarrow \Gamma$ whose final densities do not depend on the initial states, $I.\sigma = I.\tau$ for all $\tau, \sigma \in \Gamma$, can be used to initialise program variables. This is the case if constant values are assigned to all program variables. We will identify I with the set of densities that comprises its range.

From $K : \mathbb{P}\,\mathbb{D}\,\Gamma$, a set of densities, we derive a vector program $I : \Gamma \Leftrightarrow \Gamma$ as stated above,

$$I.\sigma \quad \mathrel{\widehat{=}} \quad K \ .$$

Conversely, this definition of initialisation K can be read to interpret certain programs I as initialisations by identifying them with a set of densities K. We use this to define a shorthand notation for program initialisation. If a program is preceded by a set of densities the outcome is again a set of densities.

$$K ; Q \quad \mathrel{\widehat{=}} \quad \bigcup_{\sigma \in \Gamma} (I ; Q).\sigma \ .$$

We say $K ; Q$ is an *initialised program*. Note that $K ; Q$ equals $(I ; Q).\sigma$ for any $\sigma \in \Gamma$.

4 Probabilistic action systems

A *probabilistic action system* $\mathbf{A}$ is defined by a tuple (I, Q) where

- $I : \mathbb{P}\mathbb{D}\Gamma$ is a set of state densities, the *initialisation* of $\mathbf{A}$,
- $Q : \Gamma \Leftrightarrow \Gamma$ is a vector program, the *action* of $\mathbf{A}$.

The behaviour of a probabilistic action system is described by its traces and impasses. Its traces are sequences of state densities the system may undergo during its evolution. Impasses are those traces of the system it cannot continue. The semantics of a probabilistic action system $\mathbf{A}$ 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 $\mathbf{A}$

$$\begin{aligned} \mathsf{tr}.\mathbf{A}.\langle\,\rangle &\mathrel{\widehat{=}} \text{true} \;, && (2)\\ \mathsf{tr}.\mathbf{A}.\langle f\rangle &\mathrel{\widehat{=}} f \in I \;, && (3)\\ \mathsf{tr}.\mathbf{A}.t^\frown\langle f\rangle^\frown\langle g\rangle &\mathrel{\widehat{=}} \mathsf{tr}.\mathbf{A}.t^\frown\langle f\rangle \wedge g \in \{f\} * Q \;. && (4) \end{aligned}$$

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

$$\begin{aligned} \mathsf{im}.\mathbf{A}.\langle\,\rangle &\mathrel{\widehat{=}} I = \varnothing \;, && (5)\\ \mathsf{im}.\mathbf{A}.t^\frown\langle f\rangle &\mathrel{\widehat{=}} \mathsf{tr}.\mathbf{A}.t^\frown\langle f\rangle \wedge \{f\} * Q = \varnothing \;. && (6) \end{aligned}$$

Traces of a probabilistic action system $\mathbf{A} = (I, Q)$ correspond to initialised iterates of Q in the following way:

Proposition 4.1 $\mathsf{tr}.\mathbf{A}.t \Rightarrow (\forall\, n : \mathbb{N} \mid 0 \le n < \#t \bullet t.n \in I; Q^n)$.

We also note that all prefixes of a trace of a system are traces of the same system.

Proposition 4.2 $\mathsf{tr}.\mathbf{A}.t \Rightarrow (\forall\, s \bullet s \le t \Rightarrow \mathsf{tr}.\mathbf{A}.s)$.

4.1 Refinement

Reactive system $\mathbf{B}$ implements another reactive system $\mathbf{A}$ if all behaviour $\mathbf{B}$ exhibits is possible for system $\mathbf{A}$ too. In other words, the behaviour of system $\mathbf{A}$ is *refined*. We call system $\mathbf{A}$ *abstract* and system $\mathbf{B}$ *concrete*. Usually, we speak of abstract systems and concrete systems implying that the first is implemented by the latter.

We say, a probabilistic action system $\mathbf{B}$ *refines* probabilistic action system $\mathbf{A}$, denoted $\mathbf{A} \sqsubseteq \mathbf{B}$, if the two conditions

$$\begin{aligned} \mathsf{tr}.\mathbf{B} &\subseteq \mathsf{tr}.\mathbf{A} \;, && (7)\\ \mathsf{im}.\mathbf{B} &\subseteq \mathsf{im}.\mathbf{A} \;. && (8) \end{aligned}$$

are satisfied. Thus, as a consequence of (7), a concrete system may be more deterministic than an abstract system. Condition (8) ensures that the concrete system $\mathbf{B}$ can always continue to run when it was possible for the abstract system $\mathbf{A}$.

Proposition 4.3 *Under assumption (7) condition (8) is equivalent to*
$\mathsf{tr}.\mathbf{B}.t \wedge \mathsf{tr}.\mathbf{A}.t^\frown\langle f\rangle \Rightarrow (\exists\, g \bullet \mathsf{tr}.\mathbf{B}.t^\frown\langle g\rangle)$.

The following refinement rule is easily proved.

Theorem 4.4 *Let* $\mathbf{A} = (I, Q)$ *and* $\mathbf{B} = (J, R)$ *be two probabilistic action systems. If*

$$\begin{aligned} J &\subseteq I & (9)\\ I \neq \varnothing &\Rightarrow J \neq \varnothing & (10)\\ R &\subseteq Q & (11)\\ \mathsf{dom}.Q &\subseteq \mathsf{dom}.R & (12) \end{aligned}$$

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

It is notable that refinement implies the first two conditions of the above rule. The simple reason for this is that the traces of length one just contain the initial densities.

Lemma 4.5 *Let* $\mathbf{A} = (I, Q)$ *and* $\mathbf{B} = (J, R)$.
$\mathbf{A} \sqsubseteq \mathbf{B} \Rightarrow J \subseteq I \wedge (I \neq \varnothing \Rightarrow J \neq \varnothing)$.

5 Dynamic systems

Probabilistic action systems are the basic structure to describe the behaviour of systems we are interested in. They are amended with performance objectives by giving preference to certain states and actions. Costs are assigned to states and actions for this purpose. The goal is to remain in cheap states while executing cheap actions.

The specified cost structure is reflected in the behavioural semantics which gives a refinement notion that is compatible (in the sense of theorem 6.4) with the performance measure to be defined in section 6.

5.1 Cost structures

In practice the action $Q : \Gamma \Leftrightarrow \Gamma$ of a probabilistic action system $\mathbf{A} = (I, Q)$ is written as a collection of joined vector programs Q_i,

$$Q \quad = \quad Q_1 \cup Q_2 \cup \ldots \cup Q_n \ .$$

Performance objectives can be associated with $\mathbf{A}$ by assigning costs to the components Q_i and states $\sigma \in \Gamma$. Being in state σ system $\mathbf{A}$ incurs a cost y_σ. Executing component Q_i causes a cost x_i. The objective is to keep the operating costs of $\mathbf{A}$ as low as possible. That is cheaper states and component actions are to be preferred during operation.

A *syntactical dynamic system* is defined by overlaying a probabilistic action system with a cost structure. States with equal costs are grouped together by predicates $p_j : \mathbb{P}\,\Gamma$.

$$
\begin{array}{l}
\textbf{system } Name \\
\textbf{init} \\
\quad I \\
\textbf{action} \\
\quad x_1 \leftarrow Q_1 \parallel x_2 \leftarrow Q_2 \parallel \ldots \parallel x_n \leftarrow Q_n \\
\textbf{static} \\
\quad y_1 \leftarrow p_1 \parallel y_2 \leftarrow p_2 \parallel \ldots \parallel y_m \leftarrow p_m \\
\textbf{end}
\end{array}
$$

where

- $I : \Gamma \Leftrightarrow \Gamma$ is a vector program that initialises all variables with constant values;
- $x_1, x_2, \ldots, x_n,\ y_1, y_2, \ldots, y_m : \mathbb{R}$ are real numbers, the costs associated with actions and states respectively;
- $Q_1, Q_2, \ldots, Q_n : \Gamma \Leftrightarrow \Gamma$ are vector programs;
- $p_1, p_2, \ldots, p_m : \mathbb{P}\,\Gamma$ are disjoint predicates.

We call the Q_i *component actions*, the y_j *static costs*, and the x_i *action costs.*

We illustrate the syntactical notation by means of the queueing system depicted in figure 1. It is taken from [Sen99]. The waiting room has space

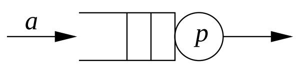

Figure 1: Single-server queue with finite waiting room

for two customers. If it is full, newly arriving customers are turned away. Customers arrive with rate $a \in [0, 1]$ and are served with rate $p \in [0, 1]$. The time model underlying this system is discrete. After one unit time has elapsed a customer has arrived during that time with probability a, and has been served with probability p.

Suppose the system provides a choice between two service rates p_1 and p_2 with $p_1 < p_2$. Further assume that it would be more costly to use service rate p_2 than service rate p_1, and that turning away customers is bad. We assign costs to the features of the system. The system can assume three different states $\{0, 1, 2\}$, the number of customers in the waiting room. The service costs are C_1 and C_2 for p_1 and p_2 respectively. And being in state i a holding cost H_i is incurred.

The whole specification as a dynamic system is given in example 5.1.

Example 5.1 *We define interleaving $Q \mid\mid\mid R$ of two vector programs Q and R by means of sequential composition and join,*

$$Q \mid\mid\mid R \quad \widehat{=} \quad Q; (R \cup |\neg \mathsf{dom}.R|\,) \cup R; (Q \cup |\neg \mathsf{dom}.Q|\,) \ .$$

We also define $\mathbf{id} \ \widehat{=}\ \mathsf{pr}.(\lambda\, \sigma \bullet \sigma)$. *This is used in the specification of the single-server queue:*

system SQueue
init
$s := 0 \cup s := 1 \cup s := 2$
action
$C_1 \leftarrow (\,|s < 2|; (s := s+1 \ {}_a\!\oplus\ \mathbf{id})) \mid\mid\mid (\,|s > 0|; (s := s-1 \ {}_{p_1}\!\oplus\ \mathbf{id}))$
$C_2 \leftarrow (\,|s < 2|; (s := s+1 \ {}_a\!\oplus\ \mathbf{id})) \mid\mid\mid (\,|s > 0|; (s := s-1 \ {}_{p_2}\!\oplus\ \mathbf{id}))$
static
$H_0 \leftarrow s = 0 \ [\!]\ H_1 \leftarrow s = 1 \ [\!]\ H_2 \leftarrow s = 2$
end

Not all syntactical systems are useful. This can be seen if a system *Name* is transformed into an equivalent system in which $Q_i = \{\tau_i \mapsto \{f_i\}\}$ for all i. It could now be that two different action costs x_i and x_j are associated with the same action Q_i. Executing Q_i would be ambiguous because it is unclear which action cost it causes.

A syntactical dynamic system is called *consistent* if

$$(Q_i \cap Q_j).\tau \neq \varnothing \Rightarrow x_i = x_j \ .$$

On the semantical level costs are represented by a cost functions c. Pairs consisting of a state σ and a state density f are associated with a cost $c.\sigma.f$. The underlying idea is that being in state σ, a system can choose between different state densities f at given costs. Each pair corresponds to a component action of the form $\{\sigma \mapsto \{f\}\}$.

A *dynamic system* is defined by a tuple $\mathbf{A} = (I, Q, c)$ where (I, Q) is a probabilistic action system, and $c : \Gamma \to \mathbb{D}\,\Gamma \to \mathbb{R}$ is a *cost function.*

A consistent syntactical dynamic system *Name* is translated into a dynamic system

$$\mathbf{Name} \quad = \quad (\mathcal{I}.Name, \mathcal{V}.Name, \mathcal{C}.Name) \ .$$

Its initialisation $\mathcal{I}.Name$ is defined by $\mathcal{I}.Name \ \widehat{=}\ I.\sigma$ for any $\sigma \in \Gamma$ (see subsection 3.2). Vector program $\mathcal{V}.Name$ is the join of all Q_i

$$\mathcal{V}.Name \quad \widehat{=} \quad Q_1 \cup Q_2 \cup \ldots \cup Q_n \ .$$

Finally, the cost function is compiled from the action costs and the static costs. If a static cost is not defined for a state it is assumed to be zero.

$$\mathcal{C}.Name.\tau.f \quad \widehat{=} \quad \begin{cases} x_i + y_j & , \mathit{if} \ \ f \in Q_i.\tau \wedge p_j.\tau \\ x_i & , \mathit{if} \ \ f \in Q_i.\tau \wedge (\forall\, j \bullet \neg p_j.\tau) \\ 0 & , \mathit{if} \ \ (\forall\, i \bullet f \notin Q_i.\tau) \end{cases}$$

5.2 Behaviour

The semantics of a dynamic system $\mathbf{A} = (I, Q, c)$ is defined by a tuple $\mathsf{beh}.\mathbf{A} = (\mathsf{tr}.\mathbf{A}, \mathsf{im}.\mathbf{A}, \mathsf{ex}.\mathbf{A})$. The *expectations* $\mathsf{ex}.\mathbf{A}$ associate expected average costs e with final densities and transitions of a trace t. In $\mathsf{ex}.\mathbf{A}.(t, g, e)$, the value e is a cost dynamic system $\mathbf{A}$ may incur when it engages in a transition to a state density g after having undergone trace t.

$$\begin{aligned} \mathsf{ex}.\mathbf{A}.(\langle\,\rangle, g, 0) \;&\hat{=}\; \mathsf{tr}.\mathbf{A}.\langle g\rangle \ , && (13)\\ \mathsf{ex}.\mathbf{A}.(t^\frown\langle f\rangle, g, e) \;&\hat{=}\; \mathsf{tr}.\mathbf{A}.t^\frown\langle f\rangle \,\wedge && (14)\\ &\qquad \exists\, H : \mathsf{fun}.Q \mid \mathsf{car}.f \subseteq \mathsf{dom}.H \bullet \\ &\qquad\quad g = \textstyle\sum_{\tau\in\mathsf{dom}.H} f.\tau * H.\tau \,\wedge \\ &\qquad\quad e = \textstyle\sum_{\tau\in\mathsf{dom}.H} f.\tau * c.\tau.(H.\tau) \ . \end{aligned}$$

Note that the value $e \in \mathbb{R}$ is defined because all involved state densities have a finite carrier. The empty trace $\langle\,\rangle$ really describes a system before it is in operation. Since we are only interested in costs that are incurred during operation the empty trace is assigned cost zero. If the system has engaged in a trace $t^\frown\langle f\rangle$ then it is currently in state σ with probability $f.\sigma$. Recall from the definition of sequential composition

$$\begin{aligned} &g \in \{f\} * Q \Leftrightarrow \\ &\quad \exists\, H : \mathsf{fun}.Q \mid \mathsf{car}.f \subseteq \mathsf{dom}.H \bullet \\ &\qquad g = \sum_{\tau\in\mathsf{dom}.H} f.\tau * H.\tau \ . \end{aligned}$$

Furthermore, by (4),

$$\mathsf{tr}.\mathbf{A}.t^\frown\langle f\rangle \wedge g \in \{f\} * Q \quad\Leftrightarrow\quad \mathsf{tr}.\mathbf{A}.t^\frown\langle f\rangle^\frown\langle g\rangle \ .$$

The significance of the family of densities H is thus twofold. It guarantees that $t^\frown\langle f\rangle^\frown\langle g\rangle$ is a trace of $\mathbf{A}$, and it records possible choices of densities $H.\tau$ from $Q.\tau$ to determine the costs of transitions to density g. The sum $\sum_{\tau\in\mathsf{dom}.H} f.\tau * c.\tau.(H.\tau)$ weighs each cost $c.\tau.(H.\tau)$ of using $H.\tau$ in state τ with the probability $f.\tau$ of presently being in state τ. This is the expected average cost of choosing H to continue after trace $t^\frown\langle f\rangle$.

From the above argumentation it follows that traces can be derived fom expectations which gives yet another characterisation of them.

Proposition 5.2 $\mathsf{tr}.\mathbf{A}.t^\frown\langle g\rangle \Leftrightarrow (\exists\, e : \mathbb{R} \bullet \mathsf{ex}.\mathbf{A}.(t, g, e))$.

5.3 Refinement

We say, dynamic system $\mathbf{A} = (I, Q, c)$ is *refined by* dynamic system $\mathbf{B} = (J, R, d)$, denoted $\mathbf{A} \sqsubseteq \mathbf{B}$, if $(I, Q) \sqsubseteq (J, R)$ and

$$\mathsf{ex}.\mathbf{B} \subseteq \mathsf{ex}.\mathbf{A} \ . \qquad (15)$$

Implication (15) demands that the expected costs incurred by the concrete system at any transition be the same as the corresponding costs in the abstract system.

Refinement of dynamic systems can be proved by a rule similar to the one stated in theorem 4.4. The single condition (16) added accounts for the preservation of the performance objectives.

Theorem 5.3 *Let* $\mathbf{A} = (I, Q, c)$ *and* $\mathbf{B} = (J, R, d)$ *be two dynamic systems. If conditions (9), (10), (11), (12) and*

$$\forall \sigma : \mathsf{dom}.R \bullet \forall f : R.\sigma \bullet c.\sigma.f = d.\sigma.f \tag{16}$$

hold then $\mathbf{A} \sqsubseteq \mathbf{B}$ *.*

6 Finite live dynamic systems

Throughout this section we often identify probabilistic action systems and dynamic systems by leaving out cost functions when they are not needed. The term dynamic system, or shorter system, is used generically to refer to both kinds of system.

We call dynamic systems without impasses *live*. A performance measure is defined for finite live dynamic systems, where a dynamic system $\mathbf{A} = (I, Q)$ is called *finite* if both I and Q are finite. The performance measure is not applicable to dynamic systems that have impasses. But it can be used to compare the performance of dynamic systems that are derived from finite live dynamic systems because no impass can be introduced by refinement. In other words, refinement preserves liveness.

Proposition 6.1 $\mathbf{A}$ *is live and* $\mathbf{A} \sqsubseteq \mathbf{B}$ *implies* $\mathbf{B}$ *is live.*

6.1 Infinite behaviour

The behaviour of a dynamic system that goes on indefinitely gives rise to the infinite behaviour of the system. An infinite sequence[4] of state densities v is called an *infinite trace* of a dynamic system $\mathbf{A} = (I, Q)$ if $\mathsf{itr}.\mathbf{A}.v$ holds, where

$$\mathsf{itr}.\mathbf{A}.v \quad \widehat{=} \quad v.0 \in I \wedge (\forall i \bullet v.(i+1) \in \{v.i\} * Q) \ .$$

Infinite traces of dynamic systems can be approximated by finite traces. More precisely, v is an infinite trace of $\mathbf{A}$ iff all its finite prefixes are traces of $\mathbf{A}$.

Proposition 6.2 $\mathsf{itr}.\mathbf{A}.v \Leftrightarrow (\forall t \mid t \leq v \bullet \mathsf{tr}.\mathbf{A}.t)$.

[4]Infinite sequences of elements of a set S are denoted by $\mathsf{seq}_\infty\, S$.

Any finite trace of a live system is a prefix of an infinite trace of the same system.

Proposition 6.3 *Let* $\mathbf{A}$ *be a live dynamic system.*
$\mathsf{tr}.\mathbf{A}.t \Rightarrow (\exists\, v : \mathsf{seq}_\infty \mathbb{D}\Gamma \bullet \mathsf{itr}.\mathbf{A}.v \wedge t \leq v)$.

In fact, the behaviour of a live system is entirely described by its infinite traces. Proposition 6.1 then says that if the initial specification is live, then all its refinements are live too. Thus, for a finite live dynamic system we can define a performance measure that considers only its infinite behaviour. The same measure is then valid for all refinements of a that dynamic system.

6.2 Performance

In this subsection a performance measure for finite live dynamic systems is defined. It measures the best possible long-run expected average cost a live system can achieve. That is the minimum over all behaviours of the system. The performance measure is only defined for finite live systems. If the requirement of finiteness is dropped, the minimum need not exist. One could, for instance, specify a system that has a behaviour with long-run average costs $\frac{1}{n}$ for all $n \leq 1$ but no behaviour with average cost 0 [Sen99]. Obviously there is no minimum, and for each behaviour there are infinitely many others that perform better.

Associated with the behavioural semantics of a dynamic system $\mathbf{A}$ is a model of discrete time. Initialisation takes place at time $n = 0$. And a transition θ, say, from trace s to $s ^\frown \langle f \rangle$ takes place at time $n = \# s$. Each time n a transition θ occurs a cost $e_n \in \mathbb{R}$ is incurred. The value e_n originates from the expectation $\mathsf{ex}.\mathbf{A}.(s, f, e_n)$ corresponding to θ.

From proposition 6.3 we know that all behaviour of a live dynamic system $\mathbf{A}$ is infinite. Hence we need only consider infinite traces v of $\mathbf{A}$. In the same way as outlined in the preceding paragraph they induce infinite sequences of expected costs $\mathsf{tc}.\mathbf{A}.v : \mathbb{P}(\mathsf{seq}_\infty \mathbb{R})$, defined by,

$$\mathsf{tc}.\mathbf{A}.v \quad \widehat{=} \quad \{E \mid (\forall\, i : \mathbb{N} \bullet \mathsf{ex}.\mathbf{A}.(\langle v.0, v.1, \ldots, v.(i-1)\rangle, v.i, E.i))\} \ .$$

Based on the infinite sequences $\mathsf{tc}.\mathbf{A}.v$ we define our performance measure $\mathsf{perf}.\mathbf{A}$ for finite live dynamic systems $\mathbf{A}$,

$$\mathsf{perf}.\mathbf{A} \quad \widehat{=} \quad \mathsf{min}_{v \in \mathsf{itr}.\mathbf{A}}\, \mathsf{min}_{E \in \mathsf{tc}.\mathbf{A}.v}\, \mathsf{avg}.E \ ,$$

where

$$\mathsf{avg}.E \quad \widehat{=} \quad \lim_{n \to \infty} \frac{1}{n}(\sum_{k=1}^{n} E.k)$$

is the long-run expected average cost associated with E. More than one infinite sequence of costs E may belong to one infinite trace v. The term

$\mathsf{min}_{E \in \mathsf{tc}.\mathbf{A}.v}\, \mathsf{avg}.E$ gives the best average cost associated with infinite trace v. Finally, taking the minimum over $v \in \mathsf{itr}.\mathbf{A}$ yields the best average cost dynamic system $\mathbf{A}$ can produce.

The performance measure interacts with refinement in the following way.

Theorem 6.4 *Let* $\mathbf{A}$ *and* $\mathbf{B}$ *be finite live dynamic systems. Then*
$\mathbf{A} \sqsubseteq \mathbf{B} \Rightarrow \mathsf{perf}.\mathbf{A} \leq \mathsf{perf}.\mathbf{B}$.

Proof

By definition of $\sqsubseteq$ for dynamic systems we have $\mathsf{tc}.\mathbf{B}.v \subseteq \mathsf{tc}.\mathbf{A}.v$ for all infinite traces v, and also $\mathsf{itr}.\mathbf{B} \subseteq \mathsf{itr}.\mathbf{A}$. Hence, by definition of the performance measure, $\mathsf{perf}.\mathbf{A} \leq \mathsf{perf}.\mathbf{B}$.

Hence an abstract system $\mathbf{A}$ sets a lower limit for the possible performance of any implementation $\mathbf{B}$. On reexamining the performance measure perf we see that this relationship with refinement is reasonable. The abstract specification $\mathbf{A}$ exhibits a collection of behaviours. There is no guarantee that $\mathbf{A}$ acts according to its best possible behaviour. An implementation $\mathbf{B}$ of $\mathbf{A}$ may now either keep some of the best behaviours or drop all of them. In the latter case the implementation can under no circumstances perform as well as $\mathbf{A}$ might. In the other case we would call $\mathbf{B}$ a good implementation because only behaviours with bad performance have been lost in the refinement. That is, $\mathbf{B}$ is a good implementation if $\mathsf{perf}.\mathbf{A} = \mathsf{perf}.\mathbf{B}$.

7 Related Work

In our probabilistic program model, nondeterministic choice $\cup$ and probabilistic choice ${}_p\oplus$ are incomparable. The order of vector programs only concerns nondeterministic choice. This is in contrast to the approaches of [MMS96, JSM97] where nondeterministic choice $\oplus$ is defined as a kind of generalised probabilistic choice,

$$Q \oplus R \;\; \widehat{=} \;\; \bigcup_{p \in [0,1]} Q \;{}_p\oplus\; R \ .$$

Obviously nondeterministic choice $\oplus$ and probabilistic choice ${}_p\oplus$ are related to each other, $Q \oplus R \supseteq Q \;{}_p\oplus\; R$. In the corresponding refinement notion, $\oplus$ can be refined by ${}_p\oplus$ with the understanding that probabilistic choice is more deterministic. The model in [JSM97] is operational like ours but does include an improper state "$\perp$" to treat divergence. It can be embedded into the probabilistic predicate transformer logic of [MMS96]. Residual probability, i.e. the probability of the occurence of "$\perp$", in those models leads to divergence where in our model it leads to impasses. Our performance measure requires liveness (as defined in section 6), which makes it important to be able to express impasses. In fact, the role impasses play in our semantical model is similar to the role divergence plays in other models. In the extreme case using trace refinement without impasses would allow us to refine any dynamic system by a system that can only engage in the empty trace. That is it can not even start. Because we are especially interested in dynamic systems that are in continual operation impasses are considered ill-behaviour. Hence new impasses should never be introduced in a refinement.

We have chosen not to introduce a special blocking state "$\top$", because we are not interested in the probability of reaching an impasse but only the possibility of that event. If f is a probability density on $\Gamma \cup \{\top\}$, then we call the restriction of f to Γ a subprobability density, or short subdensity [Koz81]. By allowing subdensities instead of densities everywhere in our vector model, and modifying the definition of sequential composition by removing the condition $\mathsf{car}.f \subseteq \mathsf{dom}.G$, we would arrive at a model where the probability of the occurence of $\top$ equals $1 - \sum_{\tau\in\Gamma} f.\tau$. Assume further that also our trace model would use subdensities. Then some infinite trace v of a system $\mathbf{A}$ might have the property $\mathsf{lim}_{n\to\infty} \sum_{\tau\in\Gamma} v.n.\tau = 0$. As an immediate consequence $\mathsf{lim}_{n\to\infty} \sum_{\tau\in\Gamma} E.n.\tau = 0$ would also hold, where $E \in \mathsf{tc}.\mathbf{A}.v$. Hence, $\mathsf{perf}.\mathbf{A} = 0$. This is not what we intended since now, in general, behaviour that leads to impasses is favoured over continual behaviour. Our model does not use subdensities explicitly but identifies all of them by entirely blocking progress (through $\mathsf{car}.f \subseteq \mathsf{dom}.G$) if one would occur in the alternative model.

We have chosen $\cup$ as nondeterministic choice because in the refinement of dynamic systems (see subsection 5) we seek a nonprobabilistic control program that is derived by refining an initial nondeterministic specification. Furthermore, the nondeterministic choice in [MMS96] will not find the minimum sought in the definition of the performance measure perf . This is due to the fact that proposition 7.1 does not hold in [MMS96].

Proposition 7.1 $Q; R = \bigcup\{ Q; P \mid P \in \mathsf{det}.R\}$.

Proof

Let $h \in \mathbb{D}\,\Gamma$ be a state density,

$$\begin{aligned}
& h \in (\textstyle\bigcup\{ Q; P \mid P \in \mathsf{det}.R\}).\sigma \\
\Leftrightarrow\quad & \exists\, P \mid P \in \mathsf{det}.R \bullet h \in (Q;P).\sigma \\
\Leftrightarrow\quad & \exists\, P, G \mid P \in \mathsf{det}.R \wedge \mathsf{fun}.P = \{G\} \bullet h \in (Q;P).\sigma \\
\Leftrightarrow\quad & \exists\, P, G, f \mid P \in \mathsf{det}.R \wedge \mathsf{fun}.P = \{G\} \wedge f \in Q.\sigma \bullet h = f * G \\
\Leftrightarrow\quad & \text{“proposition 3.2”} \\
& \exists\, G, f \mid G \in \mathsf{fun}.R \wedge f \in Q.\sigma \bullet h = f * G \\
\Leftrightarrow\quad & h \in (Q;R).\sigma \ .
\end{aligned}$$

The property of proposition 7.1 is said to characterize construction time nondeterminism [JSM97] as opposed to runtime nondeterminism in [MMS96]. In this respect our model is closer to the kind of nondeterminism proposed in [Low93], or the second model presented in [JSM97], which is derived from [Jon90].

The behavioural model for probabilistic action systems and dynamic systems has been chosen for two reasons. First, it allows a straightforward definition of refinement. Second, it relates in a simple fashion to stochastic processes [Tij94] and stochastic dynamic programming [Sen99]. The performance measure defined in section 6 originates in the theory of stochastic dynamic programming. In addition, dynamic programming algorithms [Put94, Sen99, Tij94] can be employed to check the quality of a refinement, i.e. that $\mathsf{perf}.\mathbf{A} = \mathsf{perf}.\mathbf{B}$. This is discussed in [HB99].

8 Conclusion

We have shown how reactive systems can be equiped with a performance measure. Our intention is that this measure could be used to analyse the quality of a refinement.

Probabilistic action systems were introduced as a model for reactive systems. Their behaviour was defined by traces of state densities that record a system's evolution in time. We introduced dynamic systems by extending probabilistic action systems with a means to specify performance objectives. The behaviour of a dynamic system was amended with expectations which represent performance objectives on the behavioural level. Based on expectations we defined a performance measure for dynamic systems that describes the best possible performance a system can achieve. Only the best possible behaviour is considered but not the best guaranteed behaviour.

Finiteness plays the key role for the existence of the performance measure in section 6 [Sen99]. The condition that all state densities must have a finite carrier is responsible for both the continuity of vector programs and the existence of the expectations.

The formalism of dynamic systems lacks an operator for parallel compositions. Usually a dynamic system is considered dual. One component of the system represents the controller under development and the other the environment of that controller [BSS96]. The controller specification is refined into a fitting implementation. The environment is left unchanged since it can not be modified to meet a controller's needs. In our approach we would have a probabilistic specification of the environment and a nondeterministic specification of the controller. This is an issue for further study.

Other models of reactive systems make use of data refinement to change the type of the state space of the system. Data refinement is an important tool to achieve legible abstract specifications. Data refinement is also necessary to compare different designs of a system with respect to performance, since changing the state space of a system often reflects an architectural change. The complete subject of data refinement has not been dealt with yet but it is planned as a future extension.

References

[Abr96a] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[Abr96b] Jean-Raymond Abrial. A steam-boiler control specification problem. *Lecture Notes in Computer Science*, 1165:500–509, 1996.

[BDG94] M. Bernardo, L. Donatiello, and R. Gorrieri. Operational GSPN semantics of MPA. Technical Report BOLOGNA#UBLCS-94-12, University of Bologna (Italy). Department of Computer Science., 1994.

[BKS88] R. J. R. Back and R. Kurki-Suonio. Distributed cooperation with action systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513–554, 1988.

[BSS96] M. Butler, E. Sekerinski, and K. Sere. BSS: An action system approach to the steam boiler problem. *Lecture Notes in Computer Science*, 1165:129–148, 1996.

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

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design : a Foundation.* Addison-Wesley, Reading, Mass., 1988.

[DP90] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 1990.

[HB99] Stefan Hallerstede and Michael Butler. Practical evaluation of the quality of refinements. Draft, 1999.

[Hil93] Jane Hillston. PEPA: Performance enhanced process algebra. Technical Report CSR-24-93, University of Edinburgh, Edinburgh, Scotland, 1993.

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

[Jon90] Claire Jones. *Probabilistic Non-Determinism.* PhD thesis, LFCS, Edinburgh University, 1990.

[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.

[Koz81] Dexter Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22, 1981.

[Low93] Gavin Lowe. Representing nondeterminism and probabilistic behaviour in reactive processes. Technical Report PRG-TR-11-93, Programming Research Group, Oxford University, 1993.

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

[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.

[Put94] Martin L. Puterman. *Markov Decision Processes – Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Mathematical Statistics. Wiley, 1994.

[Sen99] Linn I. Sennott. *Stochastic Dynamic Programming and the Control of Queueing Systems*. Wiley Series in Probability and Statistics. Wiley, 1999.

[Sim94] Andrew Simpson. A formal specification of an automatic train protection system. In M. Bertran M. Naftalin, T. Denvir, editor, *FME'94: Industrial Benefit of Formal Methods*, Lecture Notes in Computer Science, pages 602–617. Springer-Verlag, 1994.

[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.