A Performance-Oriented Refinement Assistant

Stefan Hallerstede* and Michael Butler
Department of Electronics and Computer Science, University of Southampton
Highfield, Southampton, SO17 1BJ, United Kingdom
{sh1, mjb}@ecs.soton.ac.uk

January 8, 2002

Abstract

Stepwise refinement transforms an abstract specification into a more deterministic concrete specification. Ultimately one arrives at a specification that is implementable. At the various stages in the refinement process decisions are made that determine how the final implementation operates. Different implementations can be compared with respect to their expected performance within their environment. In this sense refinement poses an optimisation problem. We present a B-based language and a tool that can assist in solving this optimisation problem.

1 Introduction

Formal notations like B or action systems support a notion of refinement [1, 2]. Refinement relates an abstract specification **A** to a more concrete specification **C**. Knowing **A** and **C** one proves **C** refines, or implements, specification **A**. In this paper we consider specification **A** as given and concern ourselves with a way to find a candidate for implementation **C**. We exclude concrete systems from the search that do not meet the abstract specification. In our formalism **A** specifies a performance objective with respect to which its possible implementations can be compared. An implementation is considered good if it performs well.

A variety of formalisms is used for performance analysis [4, 9]. These formalisms have in common that they offer only probabilistic choice operators. We regard systems where all choice are resolved probabilistically as deterministic. There are tools for performance evaluation of queueing systems and networks, stochastic process algebras [6], and stochastic Petri nets [8]. To our knowledge there is no such tool for action systems. 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. Nondeterminism admits a notion of abstraction that is made more concrete through refinement. In some tools the lack of nondeterminism in these formalism is partly remedied by the use of parameterised specifications, e.g. experiments in [8]. But there is no means to reason about these parameters from within the formalism.

We present the formalism of probabilistic action systems, a B-like [1] specification language for performance analysis. The semantics of our formalism is relational. Refinement

*Current Address: KeesDA, Parc Equation, 2 avenue de Vignate, 38610 Gières, France of probabilistic action systems resembles B-refinement. We believe the notation employed in performance analysis should be close to the notation used in the development process. Then specifications used in refinement, or parts thereof, could be used in performance analysis and results from the analysis can easily be transferred back. We decided on B as a foundation of probabilistic action systems because 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 nonintrusive as possible. Probabilistic action systems do not make assumptions about the performance measure. The performance measures are defined in terms of Markov decision processes [10, 11]which are closely related to probabilistic action systems [5]. In our present study we use long-run average expected-cost [10] which is explained briefly in section 2. A software tool that has been developed to assist in the computation of performance measures and optimal implementations of probabilistic action systems is also presented.

2 Probabilistic Action Systems

Probabilistic action systems are based on B and the guarded command language of [3] adding probabilistic imperative features as in [7].

Syntax

The program notation used in probabilistic action systems comprises the following:

Cost $|\, r \,|$ leaves the state unchanged and causes a cost of $r \geq 0$.

Guard $|\, p \,|$ leaves the state unchanged and blocks execution in states where predicate p evaluates to false. We define skip $\widehat{=}$ $|$ true $|$.

Assignment $x := e$ changes the value of variable x to the value of expression e.

Sequential Composition $P; Q$ executes first P then Q.

Parallel Composition $P \parallel Q$ executes P and Q in parallel. The variables changed by P and Q must be disjoint. For instance, $x := y \parallel y := x$ swaps the values of x and y.

Nondeterministic Choice $P \sqcup Q$ executes P or Q, and finite nondeterministic choice $\bigsqcup i \in I \bullet P_i$ executes one of the P_i.

Probabilistic Choice $P \;{}_p\!\oplus\; Q$ executes P with probability p and Q with probability $1 - p$. Finite probabilistic choice $\bigoplus i \in I \mid p_i \bullet Q_i$ executes Q_i with probability p_i. We require $\sum_{i \in I} p_i = 1$.

We introduce the syntactic form of probabilistic action systems by way of an example. A syntactic system has the structure pictured in the figure below. In the constants section natural number constants are declared. Usually these represent parameters for a specification. Only finite state spaces are supported by our tool. 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: finite sets, cartesian products, power sets, functions, and relations. 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. Initialisation and actions of a system are atomic. Reoccuring program text can be given a name and declared in the programs section.

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

System $QUEUE$ above depicts a simple queueing system. In the constants section the size of the buffer is declared, and required to be positive in the constraints section. The set QQ defined in the sets section contains the possible values of variable qq. Initially the buffer size qq may assume one of the values $xx \in QQ$. In the programs 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 buffer qq is measured. Afterwards possible arrivals occur with a rate of $\frac{1}{6}$, and departures follow at a rate of $\frac{1}{3}$ or $\frac{1}{4}$.

Behaviour

In system $QUEUE$ operation can continue from any state. For such systems average-cost optimality is considered an appropriate performance measure [11]. 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. 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 simple calculation shows that the successor state would be 0 with probability $\frac{5}{18}$, 1 with probability $\frac{11}{18}$, 2 with probability $\frac{2}{18}$. We say that the system is in a *probabilistic state*, written as follows:

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

Independent of the departure rate the expected cost of continuing 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 in state $qq = 1$, the expected cost of the first transition would be 1, followed by $\frac{11}{12}$, so $\langle 1, \frac{10}{12} \rangle$ is also a trace of $QUEUE$. In system $QUEUE$ all traces may continue indefinitely.

Refinement

To prove a refinement we compare the step by step behaviour of systems **A** and **C**. Simulation is a proof technique to do this: System **C** refines system **A** if system **A** can simulate the behaviour of system **C** step by step. This method is widely used in formalisms that have trace-based behaviour [2] and in data refinement [1].

Let **A** and **C** be two probabilistic action systems. If there is a deterministic cost-free probabilistic state relation M, such that

$$J \subseteq I; M \tag{S1}$$

$$\mathsf{dom}.(\mathbf{A}.a) \subseteq \mathsf{dom}.(M; \mathbf{C}.a) \quad \text{for all actions } a \tag{S2}$$

$$M; \mathbf{C}.a \subseteq \mathbf{A}.a; M \quad \text{for all actions } a \tag{S3}$$

then **A** is refined by **C**. A program is called *cost-free* if it does not contain a cost statement.

Condition (S1) ensures that the initialisation of the concrete system **C** can be matched by the initialisation of the abstract system **A**. Condition (S2) 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 (S3) ensures that the effect of the concrete action is matched by that of that of the abstract action.

Average Cost Optimality

Each of the infinite traces v of system *QUEUE* corresponds to choices of actions that have been made during the evolution of the system as suggested above. The quantity

$$\mathsf{avg}.v \;\widehat{=}\; \limsup_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} v.i$$

gives the average value of the infinite trace v. The value $\mathsf{avg}.v$ is called long-run expected average cost of v. The limit superior in definition of avg is used because the limit need not exist. The limit superior corresponds to the highest cost associated with v, i.e. the worst case. 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 [5]. In the case of system *QUEUE* the deterministic system with departure rate $\frac{1}{3}$ is optimal among all possible implementations. Other applications of refinement of probabilistic action systems can be found in [5]. 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 [10].

The average-cost optimal value of a live system **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 **A** as opposed to best guaranteed [5].

The Software Tool

The tool consists of four components which have been implemented in C, Java and SQL:

Compiler generates an abstract syntax tree (AST) from an input specification and stores it in a relational database. The input specification is an ordinary text file.

Expander takes the AST produced by the compiler and generates a probabilistic labelled transition system (LTS) from it. States and transitions of the LTS are assigned unique integer numbers. Each transition carries a history label and a cost label. A history label contains information about the action a transition stems from and the nondeterministic choices made. A cost label states the cost associated with a transition.

Solver computes the optimal implementation of the optimisation problem given by the LTS. The LTS constitutes a Markov decision process (MDP). Any optimisation algorithm solving an MDP (with respect to the desired optimisation criterion) can be employed to compute an optimal solution. The present solver is realised in C using value iteration [10]. The result of its computation, the optimal gain and optimal policy, are stored in the database.

Printer reads the optimal policy and prints it in readable form so that one can directly defer an optimal specification in text form from it. The necessary information is contained in the history labels of the transitions.

We have used the tool to compute optimal implementations of probabilistic action systems with up to 100000 states and 500000 transitions [5].

3 Conclusion

We have presented a B-like formalism for performance analysis of probabilistic action systems. Implementations of probabilistic action systems can be compared with respect to their performance. The probabilistic action systems have a notion of refinement and an accompanying proof rule that resembles those of B and other similar notations. So experience gained in those is applicable in the use of probabilistic action systems too. We have also presented a software tool that computes performance measures and optimal implementations of probabilistic action systems.

References

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

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

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

[4] N. Götz, U. Herzog, and M. Rettelbach. Tipp – a stochastic process algebra. In *Proc. of the Workshop on Process Algebra and Performance Modelling*, pages 427–430, 1993.

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

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

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

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

[9] 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.

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

[11] Henk C. Tijms. *Stochastic Models: An Algorithmic Approach*. Wiley Series in Probability and Mathematical Statistics. Wiley, 1994.