Towards Communication-Based Steering of Complex Distributed Systems

Klaus Dräger and Marta Kwiatkowska

Department of Computer Science, University of Oxford, Oxford, UK

Abstract. Quantitative verification is an established automated technique that can ensure predictability and dependability of software systems which exhibit probabilistic behaviour. Since offline usage of quantitative verification is infeasible for large-scale complex systems that continuously adapt to the changing environment, quantitative runtime verification was proposed as an alternative. Using an illustrative case study of communicating, distributed probabilistic processes, we formulate the problem of quantitative steering, a runtime technique that involves system monitoring, prediction of future errors, and enforcement of system's behaviour away from the error states. We consider a communication-based variant of steering where enforcement is achieved by modifying the contents of communication channels. Our approach is based on stochastic games, where one player is the system and the other players assume the role of the controller, and hence steering reduces to finding a controller strategy that meets the given quantitative goal. We discuss the solution to the quantitative steering problem and its extensions inspired by complex real-world scenarios.

1 Introduction

Software systems underpin the vast majority of our activities, from commerce, to manufacturing, transport and healthcare. Typical requirements for such systems are that they run in distributed and de-centralised environments; must be fault-tolerant, since devices may fail and communication media may be unreliable; and are expected to run continuously, adapting to the changes in the environment, for example user demand. Being deployed in business-critical setting, they must also behave in a predictable and dependable manner.

Formal verification techniques such as model checking [10] have proved particularly useful in preventing errors in the deployed software. Formal verification is used mainly in an offline fashion, though there have been recent efforts to integrate it within autonomic systems [9], where adaptive behaviour can be handled by applying concepts from control such as feedback loops. In this context, software is monitored at runtime, its behaviour analysed against given requirements, and, if deviation is detected, instructions are issued to steer its behaviour accordingly. In cases where software systems can exhibit failure and must comply with resource limitations, the modelling frameworks typically allow for probabilistic behaviour and annotation with appropriate quantities to represent the

incurred cost, e.g. energy usage. *Quantitative verification* [19] is a technique which combines formal verification with numerical computation, and is able to automatically answer the questions such as "what is the maximum probability of reaching an error state?", and "what is the expected energy usage in the start up phase?". Quantitative verification techniques have been implemented, e.g., within the PRISM model checker [20]. PRISM has been successfully used to verify a range of quantitative/probabilistic temporal properties, in some cases discovering critical flaws.

The offline application of quantitative verification, however, is usually infeasible in the context of *large-scale complex systems* [24]. The main culprit is state-space explosion in conjunction with the inherent complexity of the analysis methods that are involved. A *quantitative runtime verification* approach was recently proposed [2,16,1] as an alternative, complementary analysis method. We adopt this approach, and focus on the following system characteristics:

- adaptivity in presence of probabilistic choice: we explicitly model failure using probability distributions, and allow for continuous changes as the system evolves, including changes to probability values and system transitions;
- resource limitations: we model resource limitatations, for example finite message queue sizes, by placing quantitative bounds on them;
- partial observability: we assume that, while we have a formal model overapproximating the behaviour of the processes, we know nothing about their current internal state other than what we can infer from the model and the communication history.

In this paper we formulate the problem of *quantitative runtime steering* for large-scale complex systems that exhibit the above characteristics. The (non-quantitative) steering problem has been earlier solved in the context of distributed systems [26], where a model checker has been used to predict and prevent future inconsistencies. As a representative setting, we consider systems comprising a number of distributed probabilistic processes, communicating through message channels. We assume that each process is modelled as a Markov decision process (see the next section for details) and the system is constructed through parallel composition of those. To enable steering, we allow an explicit controller process who can use the channels both as a source of information (to try to determine the actual system state) and as a steering medium (by altering the channel contents). We then take a *stochastic game* view [8] of the system, where, in addition to a randomised player that deals with probabilistic transitions, we have:

- player 1 representing the decisions of a controller, striving to ensure that the required quantitative property holds; and
- player 2 representing the combined decisions of the system components, which in order to cover the worst possible scenario is usually assumed to be malicious.

In [8], a reward-based temporal logic and verification algorithms were proposed for turn-based stochastic games and implemented as an extension of PRISM [23].

The logic can express properties such as "player 1 has a strategy to ensure that the maximum probability of reaching a final state is at least 0.99, regardless of the strategy of any other player". Here, for simplicity, we assume that the controller's goal is to ensure a *safety* objective, namely the avoidance of a given set of *error* states. As a secondary goal, we want the controller to achieve this objective in the least intrusive way possible. We model this requirement by associating a *cost* with every alteration of the channels; this naturally leads to the definition of the steering problem in terms of a generation of a controller strategy that meets the stated *quantitative goal*.

To illustrate our approach, we introduce a motivating case study which forms the basis of our discussions. We then describe how to solve the steering problem outlined above, at first treating a simplified variant which can be solved exactly using existing methods. As mentioned above, due to the adaptivity and partial observability, this exact solution will be inappropriate for very large systems; in the following sections we discuss how to adapt the method to a variety of harder scenarios inspired by real-world complex systems. We conclude the paper by summarising future research in this area.

Related Work. The idea to incorporate the use of formal methods at runtime dates back to the work of Crow and Rushby [11] on fault detection, identification and reconfiguration. Subsequent developments include the framework of [22]. In [26] a model checker executed from the current local state has been used to predict and prevent future inconsistencies in a distributed system. In the quantitative runtime setting, a number of approaches have been proposed for different types of models, to mention the autonomic approach of [2,1] for discrete- and continuous-time Markov chains, parametric techniques of [15,16] for discrete time Markov chains and the incremental approach of [21] for Markov decision processes. Partially observable Markov decision processes are known to be infeasible, but a promising partial approach to adversary generation was recently proposed in [18]. Stochastic games have been a very active research topic, see e.g. [13,7,3]. A survey of results can be found in [5] and an overview of partially observable stochastic games in [4]. The majority of the work has been theoretical, and we are aware of only two implementations, GIST [6] for synthesis and PRISM-games for quantitative verification [8,23]. Our paper is the first to propose stochastic game techniques as a solution to quantitative runtime steering.

1.1 Case Study

Consider the following example, motivated by a cloud computing scenario, for distributing a workload of tasks among a number of processing units. We are given processing units $P_1, \ldots, P_n$ arranged in a network (see Figure 1). For the sake of simplicity, the network topology we use in this example is a ring of processes, each of which can communicate with other processes up to two positions away, and with an *environment* process.

Figure 2 shows the abstract specification for process $P_r, r \in \{1, \ldots, n\}$. The process executes the following main loop:

- In the idle location q_1, it can receive requests to perform some computation, either as a req_r message from the environment, or as a $a_{i,r}$ or $b_{i,r}$ message from another process P_i, representing some subtask which P_i generated to distribute the workload. Note that i serves as a free variable, which parameterizes families of actions ($a_{i,r}$ and $ack_{i,r}$) and locations ($q_{11,i}$) in P_r. The process moves to q_2, storing the client process id in variable m (a value of 0 representing the environment).
- In each of the *busy* locations q_2, q_3, q_4, further requests from another P_i are rejected using a $nack_{r,i}$ message.
- In locations q_2 and q_3, the process makes a probabilistic decision about whether or not to issue a subrequest of type a or b, respectively. If the answer is positive, it then chooses a recipient, by calling the auxiliary function *pick* j (resp. *pick* k), which
  - assigns to j (resp. k) a neighbour id chosen uniformly at random, excluding the current values of j, k, m, and
  - increases the counter *tries*,

  and sending an $a_{r,j}$ (resp. $b_{r,k}$) message.
- In q_4, the process checks and acts on responses:
  - if *tries* exceeds a given bound, give up and send a $nack_{r,m}$ message,
  - if an *ack* arrives from P_j (or P_k), set j (respectively k) to zero,
  - if both j and k are zero, there are no pending subrequests; send an *ack* message,
  - if either P_j or P_k send a *nack*, choose a new recipient for the failed subrequest (again using *pick* j/*pick* k).

Note that the system is driven by user demand and does not offer a guarantee that the workload will be successfully distributed; in fact, it may fail to do so and this can be expressed using probability. This system therefore exhibits two problems. Firstly, in the case when a process fails to distribute the workload despite having tried the specified number of times, it is possible that *ack* or *nack* messages arrive after the *tries* counter has exceeded the bound; these messages are not cleaned up and can confuse the process in subsequent computations. Secondly, the system may enter a configuration in which failure is unacceptably high, for example, if the workload is distributed badly between the units. Suppose the initial request leads to generation of subtasks as in Figure 3, where in the rightmost configuration processes $P_1, \ldots, P_7$ are all busy (note that these are only a subset of the full system, and some of them may still have subrequests to send). Then in that rightmost column:

- P_5 has only busy neighbours, so if it generates any subrequests, it will fail (after 5 attempts). The probability for this case is 0.64.
- If P_5 does fail, P_3 receives the resulting $nack_{5,3}$ and re-sends its subrequest up to 4 times; the recipient is chosen randomly from $\{P_1, P_2, P_5\}$, subject to the condition that the same recipient cannot be chosen in two successive attempts. Since all neighbours other than P_5 are busy, this will lead to a failure of the overall request with probability 0.4672 (see the case $k = 5, tries = 2$ in Figure 4).

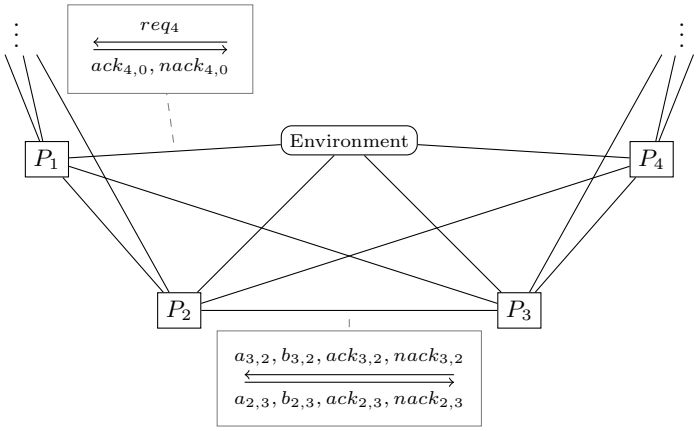

Fig. 1. A simple cluster of computing units. Processes $P_1, \ldots, P_n$ are arranged in a ring, each able to directly communicate with processes up to two positions away, and with the environment, which can generate requests req_i for each process P_i. Messages $a_{i,j}, b_{i,j}$ represent subrequests from P_i to P_j; $ack_{i,j}$ and $nack_{i,j}$ are success and failure notifications (in the latter case, j can be 0, representing a notification to the environment).

Our goal is to prevent the above scenario through using a controller which cancels requests responsible for creating such contiguous overloaded regions by deleting them and injecting a *nack* message. The idea is that the controller is able to predict that congestion is reachable in the near future, and can then select an appropriate strategy to avoid it. In the example scenario, the congestion problem could be addressed by trying to get P_6 to send its second subrequest to a process further down the chain, instead of its direct neighbour. In order to do this, the controller would delete the request ($a_{6,7}$ or $b_{6,7}$) and inject a $nack_{7,6}$ message.

2 Preliminaries

In this section, we formally describe the class of systems we are interested in, together the corresponding quantitative verification and steering problems. The systems comprise a number of distributed, probabilistic processes, each modelled as a Markov decision process, communicating through (bounded) channels.

2.1 Words and word distances

As usual, the sets of finite and infinite *words* over an alphabet Σ are denoted by Σ^* and Σ^ω, respectively. The empty word is ϵ, and the length of a word $w \in \Sigma^*$ is $|w|$.

In order to represent the intrusiveness of a steering strategy, we use a *distance* between words, based on the number of steps needed to transform one into the other, where a step consists of deleting or inserting a symbol (corresponding

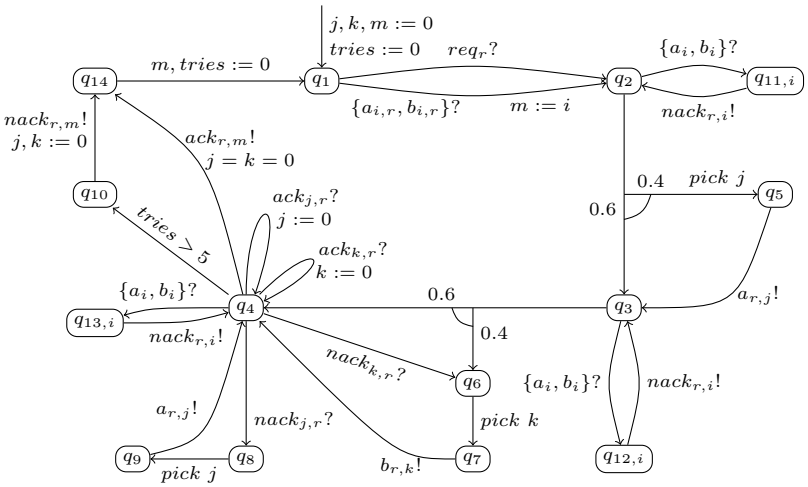

Fig. 2. One process P_r in the cluster. Variables m, j, and k are used to store the ids of neighbours whose subrequest P_r is currently processing (m) and to which subrequests have been sent (j, k). The auxiliary actions *pick* j and *pick* k, called before sending a subrequest, assign a neighbour index other than the current values of j, k, m to j or k, respectively, and increment the *tries* counter.

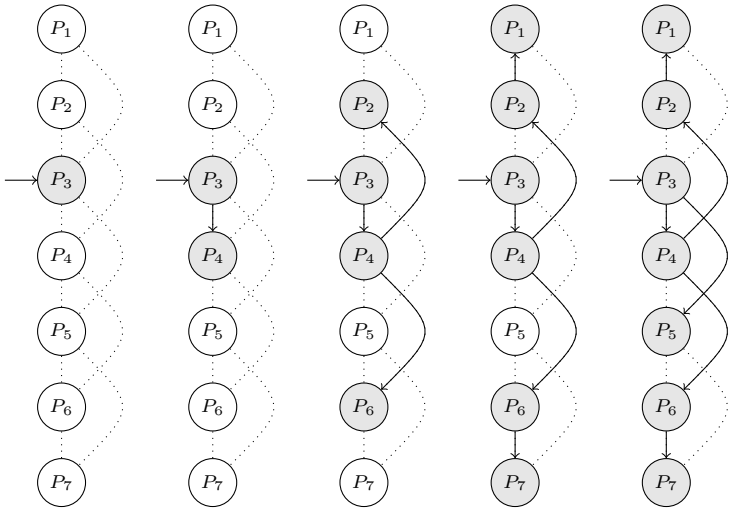

Fig. 3. Buildup of congestion in part of the network. Arrival of a request leads to a cascade of subrequests, eventually rendering all processes in the segment $P_1, \ldots, P_7$ busy (indicated by shading). In particular, P_5 is effectively isolated in the final state (rightmost column), since it has only busy neighbours. This leads to a failure probability for the overall task which is unnecessarily high.

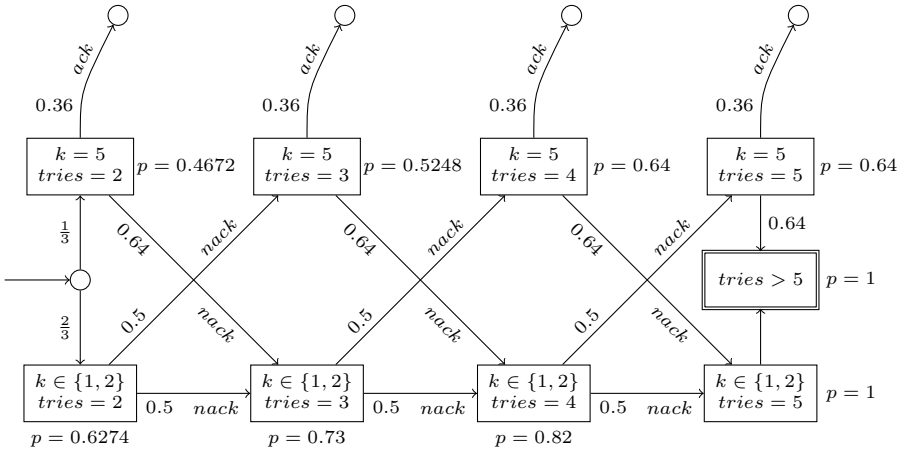

Fig. 4. Probabilities of overall failure in the situation shown in Figure 3, starting in the next-to-last column and abstracting away all details except the values of $k \in \{1, 2, 5\}$ and the *tries* counter in P_3. The subrequest can only succeed (with probability 0.36) when $k = 5$; the value of k must change in each iteration.

to interception or injection of a message). Specifically, we have the following definition.

Definition 1. *Let Σ be an alphabet, and let $\gamma = (\gamma_-, \gamma_+)$ consist of the* cost *functions $\gamma_-, \gamma_+ : \Sigma \to \mathbb{N} \cup \{\infty\}$. This gives rise to a weighted directed graph $G_{\Sigma,\gamma} = (\Sigma^*, E, w)$, where E consists of all pairs of the form $(xay, xy), (xy, xay)$ for $a \in \Sigma$ and $x, y \in \Sigma^*$, and the edge weight w is given by $w(xay, xy) = \gamma_-(a)$ and $w(xy, xay) = \gamma_+(a)$.*

The (weighted edit) distance $d_\gamma(u, v)$ between words $u, v \in \Sigma^*$ is the minimal weight of a path from u to v in $G_{\Sigma,\gamma}$.

2.2 Markov Decision Processes

Definition 2. *Given a set S, a finitely supported* probability distribution *on S is a function $\Delta : S \to \mathbb{R}_{\geq 0}$ such that $\Delta(s) = 0$ for all but finitely many $s \in S$ and $\sum_{s \in S} \Delta(s) = 1$. We denote the set of all such distributions on S by $\mathcal{D}(S)$.*

Definition 3. *A* Markov decision process (MDP) *is a tuple $A = (Q, \Sigma, q^0, T)$, where*

- *Q is a set of* states*, including the* initial state $q_0 \in Q$*,*
- *Σ is an* alphabet *of actions, and*
- *$T : Q \times \Sigma \to \mathcal{D}(Q)$ is a partial* transition function.

The action alphabet represents the possibility of nondeterministic choices in A, which can be resolved by an *adversary*. In its most general form, this adversary

makes a probabilistic choice between actions based on the history, i.e. it is given as a function $\sigma : Q^* \to \mathcal{D}(\Sigma)$. For any such adversary and any temporal property φ defining a measurable set $[\![\varphi]\!] \subseteq Q^\omega$ of paths, we get a probability $p_A^\sigma(\varphi)$ of the behaviour of A satisfying φ in standard fashion [19]. The supremum $\sup_\sigma p_A^\sigma(\varphi)$ and infimum $\inf_\sigma p_A^\sigma(\varphi)$ are denoted by $p_A^{\max}(\varphi)$ and $p_A^{\min}(\varphi)$, respectively.

For simplicity, we will restrict ourselves to *pure memoryless* adversaries, given by functions $\sigma : Q \to \Sigma$, which suffice for the case of safety/reachability properties.

2.3 The Simple Steering Problem

We first introduce a version of the steering problem which, at least in principle, can be solved exactly using existing methods. In the next section, we will discuss some complications which are typical for real-life scenarios.

We consider systems modelled as a parallel composition of a number of processes, each given as an MDP, communicating through (bounded) message channels. In particular, the alphabet of each process consists of three subsets: *internal* actions, as well as *send* and *receive* actions along the channels. Formally, we have the following.

Definition 4. *A* Probabilistic Bounded Channel System (PBCS) *is given by* $S = (A, C, \beta, \gamma)$*, where*

- $A = \{A_1, \ldots, A_k\}$ *is a finite set of MDPs* $A_i = (Q_i, \Sigma_i, q_i^0, T_i)$*,*
- $C = \{\Gamma_{i,j} \mid i, j \in \{1, \ldots, k\}, i \neq j\}$ *is a set of* message alphabets*, whose disjoint union we denote by* Γ*,*
- $\beta : C \to \mathbb{N}$ *is a* channel bound *function, and*
- $\gamma = (\gamma_-, \gamma_+)$ *is a pair of cost functions on* Γ*,*

such that, for $i = 1, \ldots, k$*, the alphabet* Σ_i *in* A_i *is the union of a set* Λ_i *of* local *actions,* $\{a? \mid a \in \Gamma_{j,i}$ *for some* $j\}$*, and* $\{a! \mid a \in \Gamma_{i,j}$ *for some* $j\}$*.*

A (global) state $s = (l, c)$ *of* S *consists of*

- *a tuple* $l = (l_i)$ *of* local states $l_i \in Q_i$ *for each* i*, and*
- *a tuple* $c = (c_{i,j})$ *of* channel contents $c_{i,j} \in \Gamma_{i,j}^*$ *for all* $i \neq j$*.*

We denote the set of global states by Q_S*. The* initial *global state* s^0 *is given by* $l_i = q_i^0$ *for all* i *and* $c_{i,j} = \epsilon$ *for all* $i \neq j$*.*

A transition of this system corresponds to a transition in one of its processes, which, as a side effect, may add a new message to an outgoing channel (if the action was a *send*) or consume a message from an incoming channel (if it was a *receive*). These two types of non-local transitions may only happen if the channel in question is not full or empty, respectively. Formally, we define the transition relation of the composed system S.

Definition 5. *Let* $S = (A, C, \beta, \gamma)$ *be a PBCS and* $s = (l, c)$ *its global state.*

- *For $q \in Q_i$, $s[q] = (l', c)$ is obtained by replacing the location l_i of A_i in l with q, i.e. $l'_i = q$ and $l'_j = l_j$ for $j \neq i$.*
- *For $\Delta \in \mathcal{D}(Q_i)$, $s[\Delta] \in \mathcal{D}(Q_S)$ is the distribution defined by $s[\Delta](s[q]) = \Delta(q)$ for $q \in Q_i$ and $s[\Delta](s') = 0$ otherwise.*
- *For $a \in \Gamma_{i,j}$, $a.s = (l, c')$ and $s.a = (l, c'')$ are obtained by appending a to the front or back, respectively, of the channel contents $c_{i,j}$:*
  - *$c'_{i,j} = a.c_{i,j}$ and $c''_{i,j} = c_{i,j}.a$;*
  - *$c'_{r,s} = c''_{r,s} = c_{r,s}$ otherwise.*

Then the transitions in A_i translate into the following system transitions *for all $s = (l, c)$:*

- *if $a \in \Lambda_i$ and $T_i(q_i, a) = \Delta$, then $T(s, a) = s[\Delta]$;*
- *if $a \in \Gamma_{i,j}$, $|c_{i,j}| < \beta(\Gamma_{i,j})$ and $T_i(q_i, a!) = \Delta$, then $T(s, a!) = (s.a)[\Delta]$;*
- *if $a \in \Gamma_{j,i}$ and $T_i(q_i, a?) = \Delta$, then $T(a.s, a?) = s[\Delta]$.*

Thus the composed system induces an MDP (Q_S, Σ, s^0, T) over the global states, and our goal is to ensure that this MDP satisfies a given safety property, expressed as a subset $E \subseteq Q_S$ of error states to be avoided.

In order to do this, we assume a *controller* whose task is to steer the system away from the bad states. This controller cannot directly influence the decisions of the system processes, but has access to the communication channels, and can remove or insert messages. The set of *controller transitions* is thus given by $(l, c) \Rightarrow (l, c')$ for all l, c, c'. The *cost* of such a transition is the sum of the distances between the channel contents in c and c', i.e. $d((l, c), (l, c')) = \sum_{i \neq j} d_\gamma(c_{i,j}, c'_{i,j})$.

This gives rise to a *stochastic game* structure between the system and the controller. One round of this game consists of the system executing an enabled system transition, followed by a probabilistic choice according to the resulting distribution, and the controller executing one of its transitions, i.e. altering the channel contents.

Definition 6. *The* steering game $G_S = (N, I, M, c)$ *for a PBCS $S = (A, C, \beta, \gamma)$ consists of:*

- *the set $N = N_S \cup N_C$ of* nodes*, where*
  - *$N_S = Q_S \times \{0\}$ is the set of* system nodes*, containing the* initial node *$I = (s^0, 0)$,*
  - *$N_C = Q_S \times \{1\}$ is the set of* controller nodes*,*
- *the set $M = M_S \cup M_C$ of* moves*, where*
  - *M_S is the set of system moves $\{((s, 0), \Delta') \mid s \to \Delta, \Delta'(s, 1) = \Delta(s)$ for all $s\}$,*
  - *M_C is the set of controller moves $\{((s, 1), (s', 0)) \mid s \Rightarrow s'\}$,*

  and
- *$\pi : M \to \mathbb{N}$ gives the* cost *of a move, where $\pi(m) = 0$ for $m \in M_S$ and $\pi((s, 1), (s', 0)) = d_\gamma(s, s')$ represents the total cost of steering operations to obtain s' from s.*

A play *of G_S is an infinite sequence of nodes $p = n_0, n_1, \ldots$ such that $n_0 = I$ and, for all i, $(n_i, n_{i+1}) \in M$.*

We are interested in the long-run cost of such plays. Since a play is in general infinite, we cannot simply add up all the costs of its moves, because this sum would diverge. A standard solution to this problem is to use the *discounted payoff* of p, which is defined in terms of a suitably chosen *discount factor* $\lambda \in (0, 1)$, as the infinite sum $\sum_{k=1}^{\infty} \lambda^k c(n_{k-1}, n_k)$.

The behaviour of the system and controller players is given in terms of strategies $\sigma_S : N_S \to \mathcal{D}(N_C)$ and $\sigma_C : N_C \to N_S$ such that $(n, \sigma_S(n)) \in M_S$ for all $n \in N_S$ and $(n, \sigma_C(n)) \in M_C$ for all $n \in N_C$. Any choice $\sigma = (\sigma_S, \sigma_C)$ of strategies turns the game into a Markov chain $M_\sigma = (N, n^0, T)$ in the standard fashion [8], with the probability distribution $T(n)$ given by $T(n) = \sigma_S(n)$ for $n \in N_S$ and $T(n)(\sigma_C(n)) = 1$ for $n \in N_C$.

The *expected* discounted payoff for σ_S, σ_C and discount factor λ is then given by the limit $\eta_\lambda(\sigma_S, \sigma_C) = \lim_{k\to\infty} p_k$ of the sums

$$p_k = \sum_{n_1,\ldots,n_k \in N} T(n_0, n_1) \cdots T(n_{k-1}, n_k)(\lambda^1 c(n_0, n_1) + \cdots + \lambda^k c(n_{k-1}, n_k)).$$

In order to solve the steering problem described above, we need to find a controller strategy σ_C with two properties, as defined below.

Definition 7. *A* λ-optimal *strategy for a steering game G and a set E of error states is a controller strategy σ_C which*

1. *avoids the error states, i.e. reaches E with probability 0 in $M_{(\sigma_S,\sigma_C)}$ for all system strategies σ_S, and*
2. *among the strategies satisfying the first property, minimizes the worst-case expected discounted payoff, i.e. the supremum $sup_{\sigma_S} \eta_\lambda(\sigma_S, \sigma_C)$.*

In the next section we discuss ways of finding a λ-optimal strategy, if it exists (which it may not, if the safety condition cannot be guaranteed).

3 Attacking the steering problem

3.1 The simple version

We will first consider the simplest version of the steering problem as presented in the previous section, assuming a fixed system and full observability, by which we mean that the controller is aware of the internal state of the system processes. In this case, the problem has a straightforward solution using existing methods:

1. Compute the full game graph $G_S = (N, I, M, c)$.
2. Determine the unsafe region U, starting from the set E of error nodes and iteratively adding the following sets until a fixpoint is reached:
   - all system nodes $n \in N_S$ for which there is $\Delta \in \mathcal{D}(N)$ and $n' \in U$ with $(n, \Delta) \in M_S$ and $\Delta(n') > 0$;

   - all controller nodes $n \in N_C$ such that $n' \in U$ for all n' with $(n, n') \in M_C$.
3. If U contains I, give up: we cannot avoid the error. Otherwise, remove U and all incident transitions from G_S, and use the existing algorithms [12] to find an optimal strategy for the resulting discounted payoff game on $N \setminus U$.

Unfortunately, the case of large-scale complex systems is much more complicated, and this simple solution is no longer appropriate. Some prominent difficulties are:

1. The state space, even if finite, (for example, because the processes are actually finite-state abstractions), is huge, making the explicit construction of the full game graph impractical.
2. If we allow for continuous system adaptation, this would, in our setting, manifest itself as changes to parameters such as transition probabilities, channel sizes, or cost functions. This calls for incremental quantitative verification techniques [21], which can be executed at runtime, reacting to system changes.
3. Full observability is unrealistic: all we can really expect the controller to see is the communication behaviour of the processes, i.e. the channel contents. In particular, we have to assume that processes could perform arbitrarily many unobservable internal transitions between communications, i.e. a system move would consist of a sequence of local transitions followed by a *send* or *receive* transition.

Note that the latter two points also imply that we can no longer hope to guarantee the safety property. Instead, we aim at a best-effort approach, which, in this context, we take to mean an attempt to avoid (or, in the partial observability scenario, minimize the probability of reaching) the error states within some number of steps.

We will now describe an approach which addresses the first two points; partial observability introduces some rather fundamental issues and will be discussed in Section 4.

3.2 Runtime verification

In order to tackle the more general case, we use a runtime approach, which only explores a bounded part of the state space in any given step. Specifically, we assume given a suitable *lookahead* $L \in \mathbb{N}$ such that the goal in each step is to find a minimum-cost controller strategy to avoid the error states for *at least* L steps, starting from the current state s. Note that, in this case, we can simply add up the costs of a play with no need for a discount factor.

This problem can be formalized using the techniques developed in [8]. For the original game $G_S = (N, I, M, c)$ and the error states E, define the set of error nodes $N_e := \{(s, i) \mid s \in E, i \in \{0, 1\}\}$, and consider the modified game $G'_{S,E,L,s} = (N', I', M', c')$ starting in s, where

- $N' = N \times \{0, \ldots, L\}$,

- $I' = (s, 0, 0)$,
- $M' = M'_S \cup M'_C$ with
  - the system moves M'_S given by $\{((n,i)\ ,\ \Delta') \mid n \notin N_e, (n,\Delta) \in M_S, \Delta'((n',i)) = \Delta(n') \text{ for all } n'\}$, and
  - the controller moves M'_C given by $\{((n,i)\ ,\ (n',i+1)) \mid n \notin N_e, (n,n') \in M_C, i < L\}$,
- $c'((n,i),(n',i')) = c(n,n')$ for all n, n', i, i'.

Intuitively, we have augmented the original game with a counter which is incremented on each controller move. The error states become sinks from which no escape is possible. The controller then strives to reach the states where the counter has the value L, avoiding the error states up to this bound on the number of states, but without guaranteeing that the error states will be avoided in future.

We thus need to find the minimum-cost controller strategy to reach the set of goal nodes $G = \{(n, L) \mid n \in N\}$, where the cost of not reaching them (due to a failure to avoid the error states) is infinite. In the logic *rPATL*, which is an extension of the logic ATL with the probabilistic and reward operators, this can be expressed as $\varphi \equiv \langle\langle\{C\}\rangle\rangle R^c_{\min=?}[F^\infty g]$, where C is the controller player, and g is an atomic proposition labelling the goal states.

The overall solution then proceeds as follows: in each step, use the model checking algorithm from [8] to find the strategy[1] solving the *rPATL* formula φ on the game $G'_{S,E,L,s}$ starting in the current state (note that this includes taking into account any changes in the model); make any changes which are immediately required by this strategy; and await the next system transition.

If the error condition is based on the states of just a small number of processes, this approach can be augmented using a form of *target enlargement* by first computing, for this subsystem, the set U of unsafe states (as defined in the simple case). The cylinder over U (i.e. the system states where the relevant processes are in an U-state) is then used in place of E.

4 Extensions

The game-based approach to the quantitative runtime steering problem that we introduced in the previous section is sufficient for the simplified setting, but not for many realistic large-scale complex systems scenarios. In this section, we use our framework as a basis to discuss a number of challenging variations. We briefly describe them and suggest possible solutions.

4.1 Compositional Analysis

In case the error condition is given in terms of some local error states in a subset of processes, it is tempting to try analysing these processes in isolation. If the

[1] An implementation of strategy generation for the logic *rPATL* is in progress, extending the functionality of [23].

bound L is relatively small, it may well be possible that the message channels to some of these processes are sufficiently full so that the behaviour of the other processes cannot influence them within this number of steps. Even if this is not the case, there may only be a small number of messages which could be both sent and received before the time bound, allowing us to restrict the analysis of the processes not contributing to the error.

4.2 Partial Observability

In order to cope with partial observability, the controller needs to keep track of the states in which the system processes might be, and the corresponding probabilities. The recent paper [4] gives a good overview of the complexity of this problem in general stochastic games.

The classic approach to this problem occurs in the case of partially observable Markov decision processes (POMDPs) $M = (Q, \Sigma, O, \Delta^0, T, o)$, which are MDPs extended with a set O of observables and an observation function $o : Q \to O$. The idea is that an observer infers at each step a distribution D_i based on D_{i-1}, the action a_i, and the observation $o(q_i)$; these distributions are referred to as *belief states*.

The channel systems we are considering are an extreme case, in that we only have the actions available, corresponding to a trivial (singleton) O. The controller could then maintain a sequence of belief states Δ_i, starting with the Dirac distribution Δ_0 such that $\Delta_0(s^0) = 1$, and updating based on the observed actions c_i. Let $S_i \subseteq Q$ be the *support* of c_i, i.e. the set of states q for which $T(q, c_i)$ is defined, and let p_i be the probability $\sum_{q \in S_i} \Delta_i(q)$ of S_i in Δ_i. After c_i is observed, the new belief state is

$$\Delta_{i+1}(q) = \sum_{q' \in S_i} \Delta_i(q') T(q', c_i, q)/p_i.$$

This could then be combined with the steering game as described in the previous section, where we now have an initial distribution (Δ_i) instead of an initial state. One problem with this approach is that the support of the distributions involved might grow substantially over time. For the belief states, this can be addressed to some extent by realizing that we can maintain individual distributions for the processes instead of one distribution over product states, which could complement the compositional approach when it is viable.

So far, while we have treated the internal state of processes as unobservable, we were still assuming that the controller knows about all actions which occur. There are two quite realistic deviations from this which make the analysis much harder:

– There may be a delay between the time when an action occurs and the time when the controller becomes aware of it. This can be treated as a special case of incomplete information and dealt with as in [25].

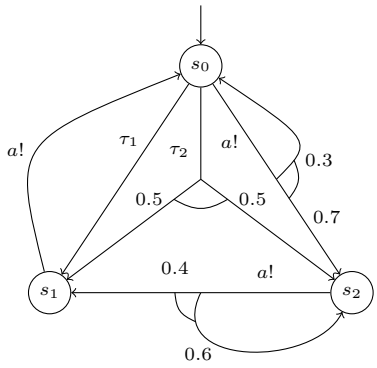

Fig. 5. State inference in systems with silent transitions. Even if the process is known to be in s_0 initially, the presence of τ-transitions means that, when observing an $a!$-event, we only know that the state was in $\{s_0, s_1, s_2\}$ before.

– Some or all internal actions can be *silent*. In particular, since the controller cannot know about their occurrence, in this case a system step would actually consist of some number of such silent transitions followed by an observable one. The example in Figure 5 illustrates this situation: when observing an action, there are several possible sequences of silent actions which might have occurred (in this case, τ_1, τ_2, or neither) resulting in several possible belief states $\{s_0, s_1, 0.5s_1 + 0.5s_2\}$, with the successor states $\{0.3s_0 + 0.7s_2, s_0, 0.5s_0 + 0.2s_1 + 0.3s_2\}$. Pursuing this naive extension of the belief state-based analysis leads to exponentially growing sets of distributions, which suggests the need for more compact (and less precise) representations for the possible process states.

4.3 Multi-way Communication Channels

The communication in our system model uses point-to-point channels, each with a dedicated sender and receiver. It would be interesting to generalize this to channels with arbitrary sets of readers and writers. Adapting the basic algorithms to this setting is straightforward (it changes the set of system transitions, which just needs to be reflected in the game structure), but the compositional approach described above becomes more difficult, since the assumptions about the possible interactions within the chosen time frame are no longer valid.

4.4 Soft Errors

We have so far used the assumption that the error states should be avoided if at all possible. Alternatively, one can consider situations in which they are merely undesirable and can be recovered from. In this case, instead of a safety property, the goal of the controller could, for example, be to minimize the time

spent in these error states; more generally, we could assign a separate cost function to states and look for controllers minimizing the discounted long-term cost. If combined with the action-based cost function, this leads to multi-objective properties for stochastic games, about which very little is currently known; see [14,17] for multi-objective model checking algorithms and implementation for MDPs.

5 Conclusion and Future Work

In this paper we have formulated the problem of quantitative runtime steering for large-scale complex systems modelled as a parallel composition of Markov decision processes communicating through bounded channels. We have shown how the simplified setting can be solved by employing stochastic games, and reduces to finding a controller which meets a given quantitative goal by manipulating the channel contents. Real-world scenarios are, however, more complex, and we have outlined the challenges and possible solutions in this case.

Quantitative runtime verification and steering are powerful new techniques that have the potential to significantly enhance fault prevention and therefore predictability and dependability of software systems. In future we will work on adding these techniques to the repertoire of automated verification.

Acknowledgements. This research is supported by EPSRC project EP/F001096.

References

1. R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola. Self-adaptive software needs quantitative verification at runtime. *Communications of the ACM*, 2012. To appear.
2. R. Calinescu and M. Kwiatkowska. Using quantitative analysis to implement autonomic it systems. In *Proc. ICSE'09*, pages 100–110. IEEE, 2009.
3. K. Chatterjee, L. de Alfaro, and T. Henzinger. The complexity of stochastic Rabin and Streett games. In *Proc. ICALP*, July 2005.
4. K. Chatterjee and L. Doyen. Partial-observation stochastic games: How to win when belief fails. In *Proc. LICS'12*, 2012. To appear.
5. K. Chatterjee and T. A. Henzinger. A survey of stochastic omega-regular games. *Journal of Computer and System Sciences*, 2011.
6. K. Chatterjee, T. A. Henzinger, B. Jobstmann, and A. Radhakrishna. Gist: A solver for probabilistic games. In *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 665–669. Springer, 2010.
7. K. Chatterjee, M. Jurdzinski, and T. Henzinger. Quantitative stochastic parity games. In *Proc. SODA'04*, pages 121–130, 2004.
8. T. Chen, V. Forejt, M. Kwiatkowska, D. Parker, and A. Simaitis. Automatic verification of competitive stochastic systems. In *Proc. 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'12)*, volume 7214 of *LNCS*, pages 315–330. Springer, 2012.

9. B. Cheng. Software engineering for self-adaptive systems: A research roadmap. In B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2009.
10. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.
11. J. Crow, J. Rushby, and P. Struss. Model-based reconfiguration: Diagnosis and recovery, 1994.
12. L. de Alfaro, T. Henzinger, and R. Majumdar. Discounting the future in systems theory. In *Proc. ICALP'03*, volume 2719 of *LNCS*. Springer, 2003.
13. L. de Alfaro and R. Majumdar. Quantitative solution of omega-regular games. In *STOC'01*, pages 675–683. ACM Press, 2001.
14. K. Etessami, M. Kwiatkowska, M. Vardi, and M. Yannakakis. Multi-objective model checking of Markov decision processes. *Logical Methods in Computer Science*, 4(4):1–21, 2008.
15. A. Filieri, C. Ghezzi, and G. Tamburrelli. Run-time efficient probabilistic model checking. In *Proc. ICSE'11*, pages 341–350, New York, NY, USA, 2011. ACM.
16. A. Filieri, C. Ghezzi, and G. Tamburrelli. A formal approach to adaptive software: continuous assurance of non-functional requirements. *Formal Aspects of Computing*, 24:163–186, 2012.
17. V. Forejt, M. Kwiatkowska, and D. Parker. Pareto curves for probabilistic model checking. In *Proc. 10th International Symposium on Automated Technology for Verification and Analysis (ATVA'12)*, LNCS. Springer, 2012. To appear.
18. S. Giro and M. Rabe. Verification of partial-information probabilistic systems using counterexample-guided refinements. In *Proc. 10th International Symposium on Automated Technology for Verification and Analysis (ATVA'12)*, LNCS. Springer, 2012. To appear.
19. M. Kwiatkowska. Quantitative verification: Models, techniques and tools. In *Proc. 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 449–458. ACM Press, September 2007.
20. M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.
21. M. Kwiatkowska, D. Parker, and H. Qu. Incremental quantitative verification for Markov decision processes. In *Proc. DSN-PDS'11*, pages 359–370. IEEE, 2011.
22. P. Oreizy, N. Medvidovic, and R. N. Taylor. Runtime software adaptation: framework, approaches, and styles. In *Proc. Companion of ICSE'08*, pages 899–910. ACM, 2008.
23. PRISM-games. http://www.prismmodelchecker.org/games/.
24. I. Sommerville, D. Cliff, R. Calinescu, J. Keen, T. Kelly, M. Kwiatkowska, J. McDermid, and R. Paige. Large-scale Complex IT Systems. *Communications of the ACM*, 55(7):71–77, July 2012.
25. S. D. Stoller, E. Bartocci, J. Seyster, R. Grosu, K. Havelund, S. A. Smolka, and E. Zadok. Runtime verification with state estimation. In S. Khurshid and K. Sen, editors, *RV*, volume 7186 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 2011.
26. M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak. Crystalball: predicting and preventing inconsistencies in deployed distributed systems. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, NSDI'09, pages 229–244, Berkeley, CA, USA, 2009. USENIX Association.