Model Checking Stochastic Automata

JEREMY BRYANS
University of Stirling
and
HOWARD BOWMAN and JOHN DERRICK
University of Kent at Canterbury

Modern distributed systems include a class of applications in which non-functional requirements are important. In particular, these applications include multimedia facilities where real time constraints are crucial to their correct functioning. In order to specify such systems it is necessary to describe that events occur at times given by probability distributions; stochastic automata have emerged as a useful technique by which such systems can be specified and verified.

However, stochastic descriptions are very general, in particular they allow the use of general probability distribution functions, and therefore their verification can be complex. In the last few years, model checking has emerged as a useful verification tool for large systems. In this article we describe two model checking algorithms for stochastic automata. These algorithms consider how properties written in a simple probabilistic real-time logic can be checked against a given stochastic automaton.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods*; *Model checking*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Mechanical verification*

General Terms: Verification, Performance

Additional Key Words and Phrases: Distributed systems, stochastic automata, model checking

1. INTRODUCTION

In this article we describe and compare two model checking algorithms for stochastic automata. Our main reason for building such algorithms is to support the verification of non-functional properties in distributed multimedia systems.

The research presented here was supported by the UK Engineering and Physical Sciences Research Council under grant number GR/L95878 (A Specification Architecture for the Validation of Real-time and Stochastic Quality of Service).

Authors' addresses: J. Bryans, Department of Computing Science and Mathematics, University of Stirling, Stirling, FK9 4LA, Scotland; email: jwb@cs.stir.ac.uk; H. Bowman and J. Derrick Computing Laboratory, University of Kent at Canterbury, Canterbury, Kent, CT2 7NF, UK; email: {H.Bowman,J.Derrick}@kent.ac.uk.

The advent of distributed multimedia applications such as video conferencing, collaborative virtual environments, video on demand and so on, place great demands on the specification and design of such systems because of the need to describe and verify non-functional requirements [Bowman et al. 1998]. These non-functional requirements typically involve real time constraints such as placing bounds on end-to-end latency, and are often called *Quality of Service (QoS)* [Bowman et al. 1998] requirements because they reflect the overall quality of delivery as opposed to the functional aspects.

In order to specify and verify such constraints it is necessary not only to be able to describe deterministic timing concerns but also probabilistic and stochastic systems. That is, in practice timings cannot be assumed to be fixed (deterministic timings) but events can occur at different times with particular probabilities. Therefore it is necessary to describe timings that occur according to certain *probability distributions*. For example, in a network specification it is not sufficient to assume that the packet deliveries arrive at fixed predetermined times, instead we need to model the system where they might arrive at times determined by (for example) an exponential distribution.

There are now a number of techniques that can be used to describe such systems, for example, Queueing Systems [Kleinrock 1975], Generalised Stochastic Petri-nets [Marsam et al. 1984], Markov Chains [Stewart 1994], generalised semi-Markov processes [Glynn 1989], Stochastic Process Algebra [Hillston 1996] and Stochastic Automata [D'Argenio 1999] and so on. In this article we consider Stochastic Automata (which are related to timed automata [Alur and Dill 1994]). We define two *model checking* [Baier and Kwiatkowska 1998] algorithms for them.

Stochastic automata are a very promising specification and verification paradigm. In particular they allow the study of both functional and non-functional requirements within the same description, giving a more complete view of overall performance than, say, a queueing theory description of the problem as well. They also support not just exponential distributions but general distributions as well. The issue here is the following. In a stochastic specification we need to associate a distribution function F with an action a so that we can describe the probability of the time delay after which a can happen. Stochastic automata naturally allow general distributions, in contrast say to stochastic process algebras which usually restrict themselves to exponential distributions [Hillston 1996].

In practice it is unrealistic to only consider exponential distributions and it is necessary for arbitrary distributions (e.g. uniform, gamma, deterministic etc.) to be considered. For example, it is often assumed that packet lengths are exponentially distributed. However, in reality this is not the case, rather they are either of constant length (as in ATM cells [Tanenbaum 1996]) or they are uniformly distributed with minimum and maximum size (as in Ethernet frames [Tanenbaum 1996]). Stochastic automata allow such arbitrary distributions to be used.

There are ostensibly two ways to move from the tractable case of exponential distributions to the less tractable case of generalised distributions. One approach is to make small generalisations of Markov chains by allowing limited

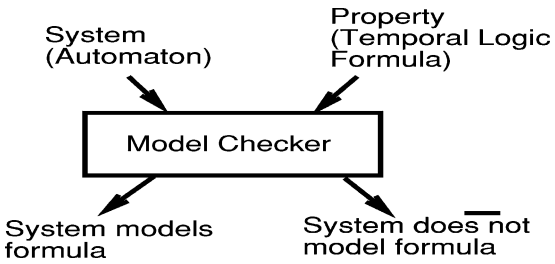

Fig. 1. Model checker.

forms of nonmemoryless behaviour (see e.g., GSPNs [Marsam et al. 1984]). However, the problem with this approach is that there will always be classes of distributions that cannot be modelled. The alternative is to allow any distribution, but then use heuristics and coarse approximation techniques to contain the problem of intractability. The majority of work on this topic follows the first of these approaches. However here we investigate the feasibility of the second approach and thus we impose few constraints on the generality of the distributions we allow in our stochastic automata.

Because stochastic automata are related to timed automata, verification strategies for stochastic automata can be derived by using the extensive work on verification for timed automata, see for example, Larsen et al. [1997]; Daws et al. [1995]; Henzinger et al. [1997]. The particular verification technique we consider is model checking [Alur et al. 1990]. This is perhaps the most successful technique to have arisen from concurrency theory. The basic approach is to show that an automaton description of a system satisfies a temporal logic property, see Figure 1.

In accordance with a number of other workers, for example Baier et al. [1999], we view the application of model checking to analysis of stochastic systems as a very exciting combination, since it provides a form of generalised transient analysis—for example the property $[\neg error\ \mathcal{U}_{<1000}\ error] < 0.01$ states that the probability of first reaching an *error* state within 1000 time units is less than 1 percent, and whether a particular stochastic system satisfies this property can be investigated.

In defining our model checking algorithm we draw heavily on the experience of model checking timed automata [Larsen et al. 1997]. However, the move from timed to stochastic automata leads to new issues that must be tackled. In particular, many of the properties that we wish to verify are inherently probabilistic. Conventional model checking allows us to answer questions such as "Is a particular sequence of events possible?", but in stochastic model checking we want to ask "What is the probability of this sequence of events?". To do this we will check an automaton against a simple probabilistic temporal logic.

We present two approaches to model checking stochastic automata. Both approaches are enumerative in the sense that, in showing whether a property holds, they enumerate reachable configurations of the system. However, the methods by which they determine the probability of being in a particular configuration are quite different. Specifically, one derives probabilities by integrating

the relevant probability density functions, while the second responds to the difficulties incurred in evaluating these integrals (which will become clear during the paper) by employing a discretisation process.

There are a number of reasons why we present two algorithms.

—The first (integral based) algorithm represents a very natural and indeed elegant approach to the problem. However, it becomes computationally intractable for many stochastic model-checking problems. The approach will only be satisfactory if sufficient probability mass can be apportioned in order to answer the model-checking question from the first few unfoldings of the automaton. In particular, it is depth in the exploration tree, rather than branching width, that is most demanding on the algorithm.

—In contrast, the second algorithm is not hampered by such intractability. However, in order for the algorithm to be applied, automata have to satisfy certain constraints. Most significantly, the probability distributions used must have a finite lower bound. Thus, if a stochastic automaton is to be checked for which these constraints do not hold, the first algorithm is effectively the only one on offer.

—We also believe that the two algorithms can eventually complement each other. The first is efficient in early iterations of exploration (in particular, branching width is not a problem for it), while the second algorithm could be employed in later stages, when depth of unfolding does become an issue. In fact, a nice illustration of using the two algorithms together is presented in Section 6.1 where model checking of unbounded until formulae is considered.

The structure of the article is as follows. In Section 2 we introduce stochastic automata illustrated by a simple example. In Section 3 we define a small probabilistic real-time logic, in which we can express simple properties against which we wish to check our stochastic automata. The first algorithm is presented in Section 4, and the second in Section 5. Then in Section 6 we look at an example of the operation of the second algorithm, in Section 7 we consider some issues of correctness and convergence relating to the second algorithm, and in Section 8 we look at the time and space complexity. We conclude in Section 9.

2. STOCHASTIC AUTOMATA

In this section we introduce stochastic automata using a small example. Stochastic automata are related to timed automata [Alur and Dill 1994], however stochastic clock settings are used, instead of the strictly deterministic timings used in timed automata. We begin with the formal definition of stochastic automata, then present a simple example. We use the definition of stochastic automata presented in D'Argenio et al. [1998].

Definition 2.1. A *stochastic automaton* is a structure $(\mathcal{S}, s_0, \mathcal{C}, \mathbf{A}, \longrightarrow\!\!\!\!\!\triangleright, \kappa, F)$ where:

- $\mathcal{S}$ is a set of *locations* with $s_0 \in \mathcal{S}$ being the *initial location*, $\mathcal{C}$ is the set of all *clocks*, and $\mathbf{A}$ is a set of *actions*.

- $\longrightarrow \subseteq \mathcal{S} \times (\mathbf{A} \times \mathcal{P}_{\text{fin}}(\mathcal{C})) \times \mathcal{S}$ is the set of *edges*, where $\mathcal{P}_{\text{fin}}(\mathcal{C})$ is the finite powerset of clocks. If s and s' are states, a is an action and C is a subset of $\mathcal{C}$, then we denote the edge $(s, a, C, s') \in \longrightarrow$ by $s \xrightarrow{a,C} s'$ and we say that C is the *trigger set* of action a. We use $s \xrightarrow{a} s'$ as a shorthand notation for $\exists C.s \xrightarrow{a,C} s'$.
- $\kappa : \mathcal{S} \rightarrow \mathcal{P}_{\text{fin}}(C)$ is the *clock setting function*, and indicates which clocks are to be set in which states.
- $F : \mathcal{C} \rightarrow (\mathcal{R} \rightarrow [0, 1])$ assigns to each clock a *distribution function* such that, for any clock x, $F(x)(t) = 0$ for $t < 0$; we write F_x for $F(x)$ and thus $F_x(t)$ states the probability that the value selected for the clock x is less than or equal to t.

Each clock $x \in \mathcal{C}$ has an associated random variable with distribution F_x. To facilitate the model checking, we introduce a function ξ, which associates locations with sets of atomic propositions.

$$\xi : \mathcal{S} \mapsto \mathcal{P}(AP)$$

where AP is the set of atomic propositions. Although, when presenting examples we will sometimes not inlcude this function. If this is the case then the atomic propositions of our formulae will be locations (in $\mathcal{S}$) of the automaton.

It is necessary to impose some limitations on the stochastic automata that can be used with the model checking algorithms. In particular, we require that each clock distribution function F_x must have a positive finite upper bound and a non-negative lower bound, and must be continuous between these bounds. The finiteness constraints mean that there are certain distribution functions that we must approximate. We further assume that clocks are only used on transitions emanating from states in which they are set.

As an example, consider the simple packet producer (which is a component in a large multimedia specification) in Figure 2. This is written

$$\begin{array}{l}(\{s_0, s_1, s_2\}, s_0, \{x, y, z\},\\ \{tryagain, conc, send, fail\}, \longrightarrow, \kappa, \{F_x, F_y, F_z\})\end{array}$$

where

$$\begin{array}{rl}\longrightarrow \;=\; & \{(s_0, tryagain, \{x\}, s_0), (s_0, conc, \{x\}, s_1),\\ & (s_1, send, \{z\}, s_0), (s_0, fail, \{y\}, s_2)\}\end{array}$$

$$\kappa(s_0) = \{x, y\},\ \kappa(s_1) = \{z\},\ \kappa(s_2) = \{\}$$

and the distribution functions for clocks x, y and z are

$$\begin{array}{rll}F_x(t) &= 2t - t^2, & \text{if } t \in [0, 1]\\ &= 0, & \text{if } t < 0\\ &= 1, & \text{otherwise}\end{array}$$

$$\begin{array}{rll}F_y(t) &= t^2, & \text{if } t \in [0, 1]\\ &= 0, & \text{if } t < 0\\ &= 1, & \text{otherwise}\end{array}$$

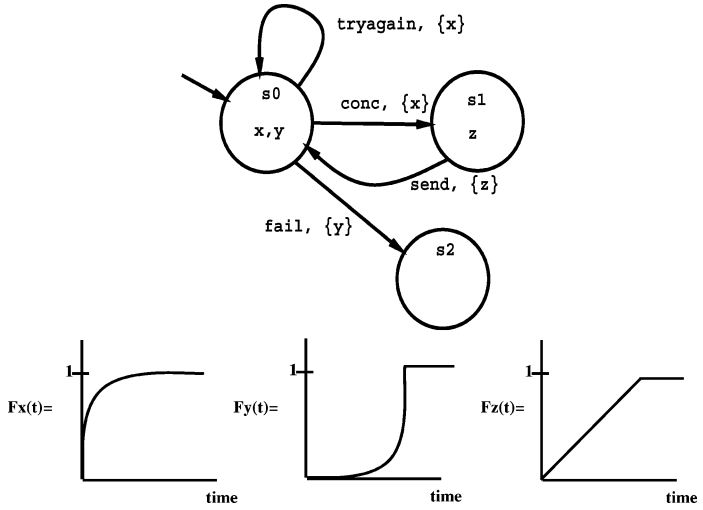

Fig. 2. The packet producer.

and

$$\begin{aligned} F_z(t) &= t, \quad \text{if } t \in [0,1] \\ &= 0, \quad \text{if } t < 0 \\ &= 1, \quad \text{otherwise} \end{aligned}$$

as depicted. The horizontal axis measures time, and the vertical axis measures the probability of the clock being set to a value less than that time.

The packet producer starts in location s_0, and attempts to establish a connection with its medium. Three options are possible at this stage. Either the medium allows a connection, the medium tells the packet producer to try again or the medium takes too long and the connection fails (is timed out). These options are modelled in the automaton by setting clocks x and y according to the functions F_x and F_y. If clock x expires first then there is a nondeterministic choice between the transition labelled *conc* (which moves the automaton to state s_1) and the transition labelled *tryagain* (which moves the automaton back to state s_0). This choice is nondeterministic because in reality it would depend on the medium, which we have not specified here. If clock y expires first, then action *fail* is triggered (we say that $\{y\}$ is the *trigger set* of *fail*) and the automaton moves to state s_2. This corresponds to the medium taking too long to respond, and nothing further happens.

This example has been chosen because it is small enough that we can show, in their entirety, the set of configurations that our model checking algorithms enumerate. Thus it can be used to illustrate our two algorithms. In addition, we have chosen it because it is canonical in the sense that it illustrates the key concepts of stochastic automata—simultaneous enabling of multiple transitions generating non-determinism. The reader should also notice that this is a good example of a situation in which steady-state analysis is not interesting. Specifically, in the steady state, all the probability mass will be in state s_2. Thus, the sort of questions we wish to ask about such a system are about its transient

behaviour, for example, what is the probability of reaching state s_2 within a particular period of time and indeed this is exactly the type of question we will be able to formulate with the logic we introduce in the next section and answer with our model checking algorithms.

3. A PROBABILISTIC REAL-TIME TEMPORAL LOGIC

3.1 The Logic

In this section, we introduce a simple probabilistic temporal logic. The purpose of the logic is to express properties that we wish to check the stochastic automaton against. The logic we define allows us to check a range of such properties.

Recall that automata contain nondeterminism, and so we resolve this using the notion of *adversaries* (see for example Baier and Kwiatkowska [1998]). This is a standard approach, however, to clarify its necessity consider the stochastic automaton in Figure 2. It initially enters location s_0 and the two clocks, x and y, are set according to their corresponding distribution functions. Our algorithms will determine (among other things) the probability that clock x is set to a value smaller than clock y and that y is less than x. This determines which clock will expire first. Now consider the former of these two possibilities. Even though we know the probability that $x < y$, we do not know how to apportion that probability mass between the transitions labelled *tryagain* and *conc*. The problem being that the choice between them is not controlled by clock expiry—they both become possible at the same instant—when x expires. Such situations are described as non-deterministic in the literature. In process algebra terms, the choice between these two actions would be resolved by the environment in which the system is placed. Intuitively, the adversary plays the role of the environment—it resolves any nondeterministic choices that the stochastic automaton must make. An adversary may vary its behaviour according to the previous behaviour of the automaton, or it may prescribe that for all nondeterministic choices a particular branch is always preferred. See D'Argenio [1999] for examples of adversaries.

We assume that when we wish to model check a property against an automaton, we are also given an adversary to resolve the nondeterminism within the automaton. (Without this adversary, enumerative analysis would not be possible; the provision of an adversary is a prerequisite of model checking.) We can now, for example, answer such questions as "Given a stochastic automaton and an adversary, is the probability of a *send* event occurring within 5 time units greater than 0.8?."

The syntax of our logic is

$$\psi ::= \mathsf{tt} \mid \mathsf{ap} \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid [\phi_1 \ \mathcal{U}_{\sim c}\, \phi_2] \simeq p$$

$$\phi ::= \mathsf{tt} \mid \mathsf{ap} \mid \neg\phi \mid \phi_1 \wedge \phi_2$$

where $[\phi_1 \ \mathcal{U}_{\sim c}\, \phi_2] \simeq p$ is a *path formula*. The path formulae can only be used at the top level—they cannot be nested. This is because the model checking algorithm we give can only evaluate path formulae from the initial state and is a necessary restriction of the current approach. Further: $c \in \mathbf{N}$ (natural

numbers), ap is an atomic proposition, $p \in [0, 1]$ is a probability value and $\simeq, \sim \in \{<, >, \leq, \geq\}$.

We can define a number of derived operators. For example, other propositional operators are defined in the usual way:-

$$\begin{aligned} \mathsf{ff} &\equiv \neg\mathsf{tt} \\ \phi_1 \vee \phi_2 &\equiv \neg(\phi_1 \wedge \phi_2) \\ \phi_1 \Rightarrow \phi_2 &\equiv \neg\phi_1 \vee \phi_2 \end{aligned}$$

and we can define a number of temporal operator abbreviations.

$$\begin{aligned} [\Diamond_{\sim c}\phi] \simeq p &\equiv [\mathsf{tt}\,\mathcal{U}_{\sim c}\ \phi] \simeq p \\ [\Box_{\sim c}\phi] \simeq p &\equiv [\neg\Diamond_{\sim c}\neg\phi] \simeq p \\ [\Box\phi] \simeq p &\equiv [\Box_{\geq 0}\phi] \simeq p \\ [\Diamond\phi] \simeq p &\equiv [\Diamond_{\geq 0}\phi] \simeq p \end{aligned}$$

$$\begin{aligned} \forall[\phi_1\,\mathcal{U}_{\sim c}\phi_2] &\equiv [\phi_1\,\mathcal{U}_{\sim c}\phi_2] = 1 \\ \exists[\phi_1\,\mathcal{U}_{\sim c}\phi_2] &\equiv [\phi_1\,\mathcal{U}_{\sim c}\phi_2] > 0 \\ \forall\Box\phi &\equiv \forall[\Box\phi] \\ \exists\Box\phi &\equiv \exists[\Box\phi] \\ \forall\Diamond\phi &\equiv \forall[\Diamond\phi] \\ \exists\Diamond\phi &\equiv \exists[\Diamond\phi] \end{aligned}$$

where $\forall$ and $\exists$ are the branching time temporal logic operators, *for all* and *exist* [Emerson 1990]. See Baier et al. [1999] for similar definitions.

With this syntax, an example of a valid formula that we can check would be $[\mathsf{tt}\,\mathcal{U}_{<10}\,\mathsf{s}_2] < 0.2$, which says that the probability of reaching a fail event within 10 time units is less than than 0.2.

3.2 Model Checking

It should be clear that since we do not allow temporal formulae to be nested we can use the following recipe in order to model check a formula ψ of our logic against a stochastic automaton A.

1. For each until subformula (i.e. of the form $[\phi_1\,\mathcal{U}_{\sim c}\phi_2] \simeq p$) in ψ perform an individual model check to ascertain whether
$$A \models [\phi_1\,\mathcal{U}_{\sim c}\phi_2] \simeq p$$
2. Replace each until formula in ψ by tt if its corresponding model check was successful, or ff otherwise.
3. Replace each atomic proposition in ψ by tt or ff depending upon its value in the initial location of A.
4. ψ is now a ground term: truth values combined by a propositional connective ($\neg$ and $\wedge$). Thus, it can simply be evaluated. The automaton is a model of ψ if this evaluation yields tt, and is not otherwise.

This recipe employs standard techniques apart from the individual checking that $A \models [\phi_1\,\mathcal{U}_{\sim c}\phi_2] \simeq p$ and this is what our two algorithms address.

4. THE REGION-TREE ALGORITHM

In this section we introduce the first algorithm.

In model checking, we take a temporal logic predicate and seek to establish whether it is true for our particular specification. In order to do this, we need to define a means by which we can check the stochastic automaton against this

logic. To achieve this the temporal logic and the specification must have the same semantic model. In D'Argenio et al. [1998], stochastic automata are given a semantics in terms of *probabilistic transition systems*, so the temporal logic is given a semantics in terms of probabilistic transition systems as well (see Appendix A).

4.1 Region Trees

For practical purposes, however, we cannot construct the probabilistic transition system, since it is an infinite structure, (both in branching and depth.) We instead construct a *region tree* from the specification. This is finitely branching, but may be infinite in depth. Thus, a particular region tree represents an unfolding of the stochastic automaton to a certain depth. In fact, we use the temporal logic formula to construct a probabilistic region tree, which is used to verify the temporal logic formula. More precisely, the region tree is expanded until sufficient probability mass has been allocated to ascertain the truth or falsity of the formula (this will become clearer shortly.) In this section, we describe how to construct region trees from stochastic automata.

We begin with the definition of a valuation, which we use to record the values of all the clocks in a particular state at a particular moment in time. The unique clock $a \in \mathcal{C}$, which we add to the set of clocks, is used to facilitate the model checking. It keeps track of the total time elapsed in the execution of the stochastic automaton, but plays no part in the behaviour of the automaton.

Definition 4.1. A *valuation* is a function $v : \mathcal{C} \bigcup \{a\} \to \mathcal{R} \bigcup \{\bot\}$ such that $v(x) = \bot$ or $v(x) \leq x_{\max}$, where $x_{\max}$ is the maximum value to which clock x can be set. If $d \in \mathcal{R}_{\geq 0}$, $v - d$ is defined by $\forall x \in \mathcal{C} \bigcup \{a\} \cdot (v-d)(x) \stackrel{\text{def}}{=} v(x) - d$. The function $\min(v)$ returns the value of the smallest defined clock in the valuation v.

Since we assume that clocks are only used in the states in which they are set, there is no need to remember their value once the state has been exited. Only the clock a maintains its value; the rest are set to $\bot$. At the initialisation of a stochastic automaton, clock a is set to some natural number, (we will show later how we choose this; it depends on the formula we are interested in) and all other clocks are undefined. We define this initial valuation as $\mathbf{O}_n$, if $\mathbf{O}(a) = n$.

We also need a notion of equivalence between valuations, which will enable us to construct the regions within the probabilistic region tree. The issue here is the following. Although the size of the tree will be potentially infinite, at each node we wish to have *finite* branching. We achieve this because, although there are an infinite number of valuations possible for any particular state, there are a finite number of valuation equivalence classes. This gives us the finite branching.

Definition 4.2. Two clock valuations v and v' are *equivalent* (denoted $v \cong v'$) provided the following conditions hold:

—For each clock $x \in \mathcal{C} \bigcup \{a\}$, either both $v(x)$ and $v'(x)$ are defined, or $v(x) = \bot$ and $v'(x) = \bot$.

—For every (defined) pair of clocks $x, y \in \mathcal{C} \bigcup \{a\} \cdot v(x) < v(y) \iff v'(x) < v'(y)$.

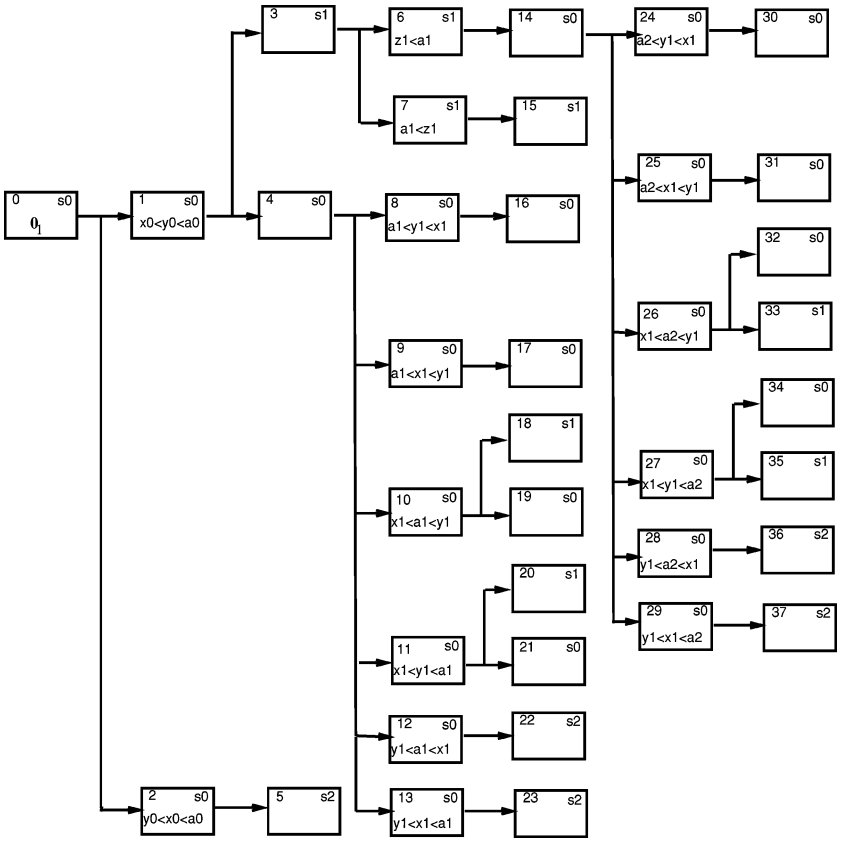

Fig. 3. The region tree.

The same clocks are defined in each valuation, and the order of the values of the defined clocks is all that is important.

The reason that the order of the values of the defined clocks is all that is important in the definition of a valuation equivalence class is that the actions are triggered by the first clock to expire. Therefore we only need to know whether one clock is greater than or less than another. Also note that there is a probability of zero that different clocks are set to the same value. This is because all distributions are assumed to be continuous.

We are now in a position to describe how a region tree is constructed from a stochastic automaton. Intuitively, we build the region tree by "unfolding" the stochastic automaton. At each newly reached state, we calculate all possible valuations (up to $\cong$) and the probabilities of each one, then from each of these (state,valuation) pairs we calculate the possible new states and repeat.

Suppose we wish to construct the region tree for the stochastic automaton in Figure 2.

The resulting region tree (up to a particular level of unfolding) is given in Figure 3. The first node is labelled with the location s_0, where the automaton starts, the valuation 0_1, (i.e. $(1, \perp, \perp)$) since clocks x and y have not yet been

set, and clock a is set to value one. Clock a is set according to the time value on the formula in which we are interested; we will give the example formula in Section 4.2. The clocks x and y are then set, giving a potential $3! = 6$ different equivalence classes. However, these can be reduced to two by observing that clock a will be fixed on 1 and $x_{\max} = y_{\max} = 1$ and the probability of either x or y being set to exactly 1 is zero.[1] Using the convention that we subscript the clock variables by the iteration number, in order to distinguish different settings of the same clock, the two possible equivalence classes are therefore $v_0(y) < v_0(x) < v_0(a)$ and $v_0(x) < v_0(y) < v_0(a)$, where $v_0(a) = 1$ in both cases. For convenience, we will write x_0 for $v_0(x)$, y_0 for $v_0(y)$ and a_0 for $v_0(a)$.

If clock x is set to less than clock y, the automaton will allow time to pass in location s_0, and each clock will count down, until clock x reaches zero. Then, either action *tryagain* or action *conc* will fire (the choice is nondeterministic), and the automaton will enter location s_0 or s_1 respectively. The time at which this occurs will obviously vary according to the initial value of the clock x. The possible locations entered are depicted by regions 3 and 4 in the region tree in Figure 3, where clocks x and y (since they are irrelevant in these regions) are not recorded. The initial value of clock a when moving from region 1 to either region 3 or region 4 will be $1 - x_0$ (we will denote this value as a_1). Thus, it will be in the range (0, 1).

If clock y is set to less than clock x (represented by region 2), then the action *fail* fires, causing the automaton to enter location s_2, and this is depicted by region 5 in the region tree. Again, in absolute terms, all we can say about the value of clock a at this stage is that it lies in the range (0, 1).

From region 3 there are two possibilities. Either clock z is set to less than a_1, (region 6), or it is set to greater than a_1 (region 7). From region 6 the action *send* will occur before the clock a expires, moving the automaton to location s_0 and the region tree to region 14. From region 7 the clock a will expire before the action *send* occurs. The region tree moves to region 15, and the automaton remains in state s_1.

From region 4 (location s_0) both clocks x and y are reset according to their probability density functions, to values x_1 and y_1. Since we cannot now be sure about the value of clock a, we have $3! = 6$ equivalence classes, and these are represented by regions 8 to 13 when we unfold the SA another level.

In regions 8 and 9 a_1 is less than the (new) initial values of clocks x and y: these regions represent the case where clock a expires before either of x_1 and y_1. When we consider a particular temporal logic formula this will represent the case where time has run out, and so the region tree moves to either region 16 (if y expired first) or region 17 (if clock x expired first).

Regions 10 and 11 represent the valuation equivalence classes where x_1 is less than both a_1 and y_1, and so from these clock x will expire first, either action *tryagain* or *conc* will be performed, and the stochastic automaton will enter either location s_0 or s_1 (regions 18–21). Similar explanations can be made for the remaining regions.

[1]This coincidence of a, $x_{\max}$ and $y_{\max}$ is assumed in order to simplify our presentation; the next iteration illustrates the general case.

The region tree can be expanded further if necessary. There is no need to continue to expand regions 5, 15, 16, 17, 22 and 23, because in all of these either the clock a has expired or the stochastic automaton has reached location s_2, which is a deadlocked state, and there is no further information to be gained. In Figure 3, further regions are derived from region 14 in the same way as above; these are needed when we build the probabilistic region tree in the next section.

4.2 Probabilistic Region Trees

Given a stochastic automaton, adversary and formula $\psi = [\phi_1 \, \mathcal{U}_{\sim c} \, \phi_2] \simeq \mathrm{p}$ the model checking algorithm iterates until the formula is found to be either true or false.

An iteration unfolds the region tree by expanding each leaf node. At each iteration stage there are two steps. The first step resolves the nondeterministic choices in the newly expanded region tree using the given adversary. The second step then calculates the probabilities on each node in the newly expanded part of the tree.

The region tree (Figure 3) represents an unfolding of the stochastic automaton without the nondeterministic choices being resolved. The probabilistic region tree (Figure 4) records the resolution of the nondeterministic choices and the probabilities at the final nodes represent the chances of taking the particular sequence of actions that end in that node.

At each iteration, we update the information we have on the probability of a path satisfying the formula. To do this, we define three new propositions, and each node of the probabilistic region tree is labelled with p, f or u: p, if it has *passed* (it is the end of a path that models the until formula ψ); f, if it has *failed* (it is the end of a path that cannot model ψ), or u, if it is *undecided*. We also have two global variables, Σp and Σf, that keep running totals of the probabilities of the *pass* and *fail* paths.

The basic idea of the model checking algorithm is that we check the values of Σp and Σf at each stage, and if we cannot deduce from these the truth or falsity of the formula we are checking, we look more closely at the undecided nodes. That is, we extend the undecided paths by each possible subsequent action, label these new nodes p, f or u, and calculate their probabilities. We then add these probabilities to Σp and Σf and repeat.

We will begin by demonstrating the technique for an example. The full algorithm appears as appendix B. Consider the example stochastic automaton (Figure 2).

Let us consider the formula

$$\psi = [(\phi_0 \vee \phi_1) \, \mathcal{U}_{<1} \, \phi_2] \geq 0.9$$

where ϕ_0 (resp. ϕ_1, ϕ_2) is the proposition that we are in state s_0 (resp. s_1, s_2). The question[2] we are therefore asking is: is the probability of reaching location s_2 (failing) within one time unit greater than 0.9?

[2]In fact, the algorithm can easily be adapted to handle questions such as "What is the probability (to within some ϵ) of a formula such as $[\phi_0 \, \mathcal{U}_{<1} \, \phi_2]$ being true?."

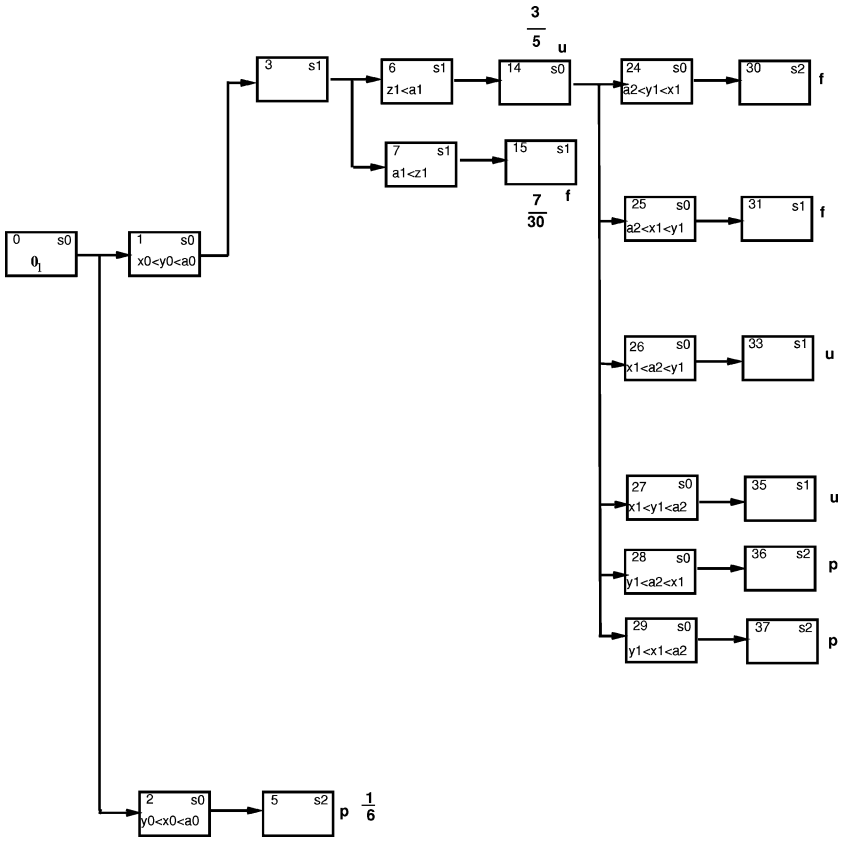

Fig. 4. The probabilistic region tree.

Note that a steady state analysis will tell us only that the automaton will fail (reach state s_2) eventually, but here we want to obtain information about the transient behaviour of the automaton. The nondeterministic choice that has to be made is between location s_1 and s_2. We will consider the benevolent adversary—the one that always chooses location s_1.

Consider region 1 first (Figure 3). It has two possible outgoing transitions, and the choice between them is made nondeterministically. So we must refer to the adversary, which chooses location s_1, that is, region 3. Region 4 is not generated. We note that the value of clock a is greater than zero (so time has not run out), and that proposition $\phi_0 \vee \phi_1$ is true (so the temporal logic formula is able to be satisfied), so this region is labelled with u (undecided).

In region 5 proposition ϕ_2 is true, and clock a is still greater than zero, so this region is labelled as passed p, and region 5 becomes a terminal node.

In region 6 a_1 is greater than the (new) initial value of clock z, therefore the *send* action will fire before clock a expires. The region is therefore labelled u.

In region 7 a_1 is less than the (new) initial value of clock z, therefore time will run out before the *send* action has a chance to fire. The region is therefore labelled f.

From region 6 the *send* action moves the automaton to location s_0 (region 14); from here there are 6 possibilities for the setting of the clocks.

Regions 24 and 25 represent the valuation equivalence classes where a_1 is less than x_1 and y_1. Since clock a will expire before either clock x or clock y, we know that these paths will not reach location s_2 in less than one time unit, so regions 30 and 31 will be labelled f. The remainder of the tree is generated in a similar manner.

Figure 4 represents two unfoldings. In order to determine whether the formula is true we also have to calculate the probabilities on the nodes. If the sum of the *pass* and the sum of the *fail* nodes is sufficient to tell us whether the formula is true then we can stop here, otherwise we unfold the tree another level.

To determine the probabilities on the arcs, we need to use probability density functions P_x, P_y and P_z of the functions F_x, F_y and F_z, which we find by differentiating F_x, F_y and F_z between their upper and lower bounds and setting them to zero everywhere else.

$$\mathsf{P}_x(t) = \begin{array}{ll} 2-2t, & \text{if } t \in [0,1] \\ 0, & \text{otherwise} \end{array}$$

$$\mathsf{P}_y(t) = \begin{array}{ll} 2t, & \text{if } t \in [0,1] \\ 0, & \text{otherwise} \end{array}$$

$$\mathsf{P}_z(t) = \begin{array}{ll} 1, & \text{if } t \in [0,1] \\ 0, & \text{otherwise} \end{array}$$

Evaluating the function F_x at a point a gives the probability that clock x is set to a value less than a, and if $a > b$, then $F_x(a) - F_x(b)$ gives the probability that clock x is set to a value between a and b, provided a and b are constants. The same calculation using the corresponding probability density function (pdf) would be $\int_b^a \mathsf{P}_x(x)\,dx$, which at first sight appears more complicated. The advantage is that these functions can be used to calculate the probability that clock x is set to a value less than y, where y is a random variable set according to the distribution function F_y. If, for example, we wished to calculate the probability of the equivalence class in region 1 ($v_0(x) < v_0(y) < v_0(a)$, where $v_0(a) = 1$) we would evaluate $\int_0^y \mathsf{P}_x(x)\,dx$, to give us a function that returns the probability that $v_0(x)$ is between 0 and y, multiply this by the pdf $\mathsf{P}_y(y)$, and integrate between zero and one:

$$\int_0^1 \int_0^y \mathsf{P}_x(x)\,dx \mathsf{P}_y(y)\,dy$$

which gives us the probability that x will be less than y, where x and y are random variables conforming to the distribution functions F_x and F_y.

We will now evaluate the probabilities of some of the arcs in the example. In the following, we will continue to subscript the clock variables by the iteration number in order to distinguish different settings of the same clock.

In our example, to determine the probability on arc (0, 2), where the value to which clock y is initially set (which we will refer to as y_0) is less than the

Table I. The Integrals

$\int_0^1 \int_0^{1-z_1} \int_{x_0}^1 \mathsf{P}_y(y_0)\,dy_0 \mathsf{P}_x(x_0)\,dx_0 \mathsf{P}_z(z_1)\,dz_1$
$\int_0^1 \int_{1-z_1}^1 \int_{x_0}^1 \mathsf{P}_y(y_0)\,dy_0 \mathsf{P}_x(x_0)\,dx_0 \mathsf{P}_z(z_1)\,dz_1$

value to which clock x was initially set (x_0), ($y_0 < x_0$) we perform the double integration

$$\int_0^1 \int_0^{x_0} 2y_0 dy_0\,(2 - 2x_0)\,dx_0$$

which evaluates to $\frac{1}{6}$.

Arc (0, 1) must have the value $1 - \frac{1}{6} = \frac{5}{6}$, since it is the only other possibility, and can be calculated as

$$\int_0^1 \int_{x_0}^1 2y_0 dy_0\,(2 - 2x_0)\,dx_0$$

These two arcs represent the setting of the clocks, and are therefore instantaneous.

From Region 2 the only region that can be reached is the leaf node region 5, therefore the arc (2, 5) has probability 1.

Calculating probabilities on the paths through region 3 is more complicated. Consider arc (3, 6) first. In fact, we must calculate the probability of the path (0, 1, 3, 6) in its entirety rather than determine separately the conditional probability of arc (3, 6). We do this as follows.

The clock setting information we know is: the first time the clocks x and y are set, the initial value of x is less than the initial value of y ($x_0 < y_0$); when z_1 is set, the sum of x_0 and z_1 is less than the initial value of clock a ($x_0 + z_1 < 1$). These constraints are captured as the combination of the integrals $\int_{x_0}^1 \mathsf{P}_y(y_0)\,dy_0$ (to ensure that $x_0 < y_0 < 1$), $\int_0^{1-z_1} \mathsf{P}_x(x_0)\,dx_0$ (to ensure that $x_0 + z_1 < 1$), and $\int_0^1 \mathsf{P}_z(z_1)\,dz_1$ (since all constraints have been captured in the first two integrals.)

The combination is given as the first integral in Table I and equals $\frac{3}{5}$.

The path (0, 1, 3, 7) differs only in the fact that a_1 ($= 1 - x_0$) is less than z_1, and can be calculated as the second integral in Table I which equals $\frac{7}{30}$. The only difference is that $\mathsf{P}_x(x_0)$ is integrated between $1 - z_1$ and 1.

At this stage in the algorithm, $\Sigma\mathsf{p} = \frac{1}{6}$ and $\Sigma\mathsf{f} = \frac{7}{30}$. Since $\Sigma\mathsf{f} > 1 - 0.9$ we can deduce that the formula is false, and in this case, there is no need to further unfold the node labelled u.

The accuracy with which we know the values of $\Sigma\mathsf{p}$ and $\Sigma\mathsf{u}$ will increase as the probabilistic region tree is extended, and in some cases it may need to be extended to infinity for perfect accuracy. However, we can achieve accuracy to within an arbitrary tolerance ϵ with a finite probabilistic region tree.

The major drawback of this algorithm is its complexity: with every new unfolding of the probabilistic region tree not only does the number of nodes to be considered increase, but also the number of integrations required to determine the probability on an individual node increases exponentially. It therefore

becomes intractable after a few iterations. This is the issue we try to tackle with the second algorithm. Rather than integrate the probability density functions, we discretise the ranges of the functions and work with the resulting approximations.

5. THE MATRIX ALGORITHM

In this section we present an overview of the second algorithm. The second algorithm takes a stochastic automaton SA, together with a bounded until temporal logic formula TL, a time step parameter δ and an adversary *pick*. For convenience we will present only the case where TL is of the form $[\phi_0 \, \mathcal{U}_{\leq c} \, \phi_1] > p$. Minor modifications to the algorithm would allow any of $\geq p$, $\leq p$ or $< p$. We use the atomic propositions ϕ_0 and ϕ_1 as part of the formula because anything more complex can be reduced to these by standard model checking techniques. Using $\leq c$ guarantees that the algorithm will terminate, although we discuss the $\geq c$ case in Section 6.1.

A single iteration of the algorithm will return one of three results: true, false or undecided. If it returns true, then the automaton models the formula. If it returns false, then the automaton does not model the formula. If it returns undecided, then the algorithm was unable to determine whether the automaton models the formula. In this case, the algorithm can be re-applied with a smaller value for the timestep δ. The question of convergence to the correct answer as δ tends to zero is discussed in section 7. For the remainder of this section we assume δ to be fixed.

A stochastic automaton has a finite number of clocks each with a probability distribution function (pdf). For each state, the set of clocks has an (arbitrary) order, and the algorithm makes use of this ordering.[3] We assume that each clock has non-zero lower and upper bounds on the values to which it can be set. The first of these is a new constraint and was not required for the first algorithm. This has been done so that δ can initially be chosen to be less than the minimum of all these lower bounds.

The algorithm works by creating a snapshot of the automaton at each time point $n\delta$ ($n \in \mathbf{N}$)[4] and extracting some global information about the probability of the formula $[\phi_0 \, \mathcal{U}_{\leq c} \, \phi_1]$ being satisfied at this point.[5] To build the next snapshot, the algorithm picks out at each time point $n\delta$ the transitions that the automaton is capable of during the next interval of length δ. Because δ is less than the minimum of all the clock lower bounds, a maximum of one transition per path can occur in each interval. Recording all possible states of the automaton at each time point is therefore enough to record all the possible transitions.

The algorithm stops when either enough information has been gathered to determine the truth or falsity of the formula, or enough time has passed so that $n\delta > c$, and allowing time to pass further will make no difference to the information we already have. In this case the result undecided is returned.

[3]However, the choice of ordering is arbitrary and does not carry any meaning. Any ordering will be sufficient.

[4]We will speak of the time instants generated by $n\delta$ ($n \in \mathbf{N}$) as time points.

[5]We also require that $\exists n.n\delta = c$, which ensures that one of the snapshots will be at exactly time c.

5.1 Data Structures

The principal data structures used by the algorithm are matrices. For each state s in the stochastic automaton we derive a matrix for a given time t (which is a rational number and calculated as $n\delta$), denoted $matrix(s, t)$, which is a record of the probabilities of the various combinations of clock values in state s at time t.

Each matrix $matrix(s, t)$ will have $\#\kappa(s)$ dimensions. Each dimension is associated with a particular clock, and the ordering of the dimensions corresponds to the ordering of the clocks. The dimension associated with a clock c will have $\lceil \frac{c_{\max}}{\delta} \rceil$ entries, where $c_{\max}$ is the largest value to which the clock c can be set, and $\lceil \frac{c_{\max}}{\delta} \rceil$ is the smallest integer greater than or equal to $\frac{c_{\max}}{\delta}$. For a clock c_i, we will abbreviate $\lceil \frac{c_{i\max}}{\delta} \rceil$ by N_i.

The valuation function v gives the value of a particular clock: $v(c_i)$ is the value of clock c_i.

Each entry in the matrix $matrix(s, t)$ is the probability that at time point t, the automaton is in state s, and each clock is within a particular time range. Thus, the value $matrix(s, t)[k_1 \ldots k_n]$ is the probability that at time point t, the automaton is in state s, and $v(c_i) \in (\delta(k_i - 1), \delta k_i]$ for each clock c_i.

A further data structure we shall need is $live(t)$, which is the set of states "live" at time t (i.e. their matrices at time t contain at least one non-zero entry, and the formula is still undecided). In order to get an accurate picture of the automaton at time $t + \delta$, we must take into account all states live at time point t.

A *snapshot* of the automaton at time t is the set of all matrices $matrix(s, t)$ where s is in $live(t)$.

Let $pr(c_i \in (\delta(k_i - 1), \delta k_i])$ be the probability that clock c_i is initially set to a value in the range $(\delta(k_i - 1), \delta k_i]$. Before the algorithm proper begins, we calculate all these values from the clock probability distribution functions, which are entered into the algorithm as part of the stochastic automaton.

5.2 Variables

The algorithm also uses a number of auxiliary variables.

$prob(s, t)$ is the probability of entering state s during the time range $(\delta(k - 1), \delta k]$ (where $t = \delta k$) and is defined for states s live at time $\delta(k - 1)$, and s' live at time δk.

$new\_states(s, t)$ is the set of states that can be reached from a state s during a time range $(\delta(k - 1), \delta k]$.

total_pass is a probability value. It is incremented at each iteration. The iterations of the algorithm correspond to the time points, and *total_pass* records the probability of the automaton having passed the formula at that time. *total_fail* is also a probability value; it records the probability of the automaton having failed the formula as the algorithm progresses.

error is an upper bound on the possible errors of *total_pass* and *total_fail*. After an iteration, we know that the actual probability of the automaton having passed the formula is in the range $[total\_pass, total\_pass + error]$, and similarly for *total_fail*.

5.3 Overview

The second algorithm is given in detail in Appendix C. We begin here with a pseudocode description.

```
build matrix(s_0, 0)
check formula against s_0 and t = 0  → pass
                                     → fail
            ↓ undecided
repeat
  t := t + δ
  forall locations s in live(t − δ)
    build matrix(s, t)              (record possible new locations)
                                    (increment prob. of entering new locations)
                                    (increment error)
    update live(t)
  forall locations s' in live(t)
    check formula against location:
      if pass then add probability to total_pass
      if fail then add probability to total_fail
      if undecided then update matrix(s', t)
until (formula has passed, or
       formula has failed, or
       t has reached the limit set by the formula)
set all locations undecided at last iteration to false
if total_pass > formulaprobability then output pass
elseif total_fail > 1 − formulaprobability then output fail
else output undecided
```

We now present the formula for initially calculating matrices, then describe the algorithm in overview, outlining the procedures involved.

If there are n clocks in state s, then $matrix(s, t)$ is calculated using the probability distribution functions of the clocks in state s as follows:

$$\forall 1 \le k_1 \le N_1$$
$$\vdots$$
$$\forall 1 \le k_n \le N_n \bullet matrix(s,t)[k_1 \ldots k_n] := \prod_{l=1}^{n} pr(v(c_l) \in (\delta(k_l - 1), \delta k_l])$$

The algorithm begins by calculating $matrix(s_0, 0)$, where s_0 is the initial state of the stochastic automaton.

$live(0)$ will either be $\{s_0\}$ or the empty set, according to whether the formula TL is made true or false by state s_0, or whether we cannot yet decide. This is determined as follows. If state s_0 models proposition ϕ_1, then the formula TL is immediately true and $live(0)$ is the empty set. Otherwise, if s_0 models ϕ_0 we cannot yet decide, and so $live(0)$ contains s_0. If the state models neither proposition then the formula TL is immediately false, and $live(0)$ is the empty set.

If the initial step does not determine whether the formula is true or false, we perform a number of iterations. Each iteration builds the snapshot at time point $t+\delta$, based upon the snapshot at time point t. The sequence of snapshots builds progressively more information as to whether the stochastic automaton has passed or failed the formula.

In the case of a bounded until formula with a $\leq c$ subscript,[6] the number of iterations is finite (i.e. the algorithm always terminates) because the iterations terminate either when sufficient information has been extracted to determine whether the formula passes or fails, or after the $\frac{c}{\delta}$th iteration, since the formula cannot become true after time c if it hasn't already.

If the information at time t is not enough to determine the truth or falsity of the formula, we build the snapshot for time point $t+\delta$. We now describe an individual iteration.

An iteration consists of two sections. In the first, we consider all of the states that are currently undecided. These are all the states in $live(t)$. For each state we create the matrices at time $t+\delta$, update $live(t+\delta)$ and calculate $prob(s', t+\delta)$ for states s' which can be reached in the interval $(t, t+\delta]$. In the second, we look at all that which can be reached in the interval $(t, t+\delta]$, and consider them with respect to the temporal logic formula. We then either update the global probabilities if the states cause the formula to pass or fail, otherwise we update the respective matrices.

Note that in this algorithm a matrix is updated at most twice. Once within procedure *new_time_matrix*(refer to Appendix C), if the state was *live* at the previous time, and once within the procedure *new_state_matrix*, if the state is reachable via a transition in the previous interval.

5.3.1 *Creating and Updating Matrices.* We begin with some necessary notation. Let us assume δ is a fixed rational number greater than zero.

Definition 5.1. If $c_1, \ldots, c_n$ are the clocks on state s, a *valuation*[7] is the vector of results of the valuation function $v(c_i)$ from clocks to $\mathcal{R}$, which gives the values of each of the n clocks.

Two valuations v and v' are $(\delta-)$ equivalent if

$$\forall c_i . \exists k_l \in \mathbf{N} . v(c_i) \in (\delta(k_l - 1), k_l] \wedge v'(c_i) \in (\delta(k_l - 1), k_l]$$

A *valuation equivalence class* (or clock configuration) is a maximal set of equivalent valuations.

If δ is understood, we can abbreviate this configuration as $(k_1, \ldots, k_n)$. For a state s and a time point t, the probability $\prod_{l=1}^{n} pr(v(c_l) \in (\delta(k_l - 1), \delta k_l])$ is an (s,t)-*clock configuration probability* (or just a clock configuration probability when s and t are understood).

There are two different procedures for updating a matrix. The first (encapsulated in the procedure *new_time_matrix*) corresponds to the situation within the stochastic automaton where time passes, but the state remains unchanged.

[6] i.e. $[\phi_0 \, \mathcal{U}_{\leq c} \, \phi_1] > p$. See Section 6.1 for a discussion of how $>c$ time bounds are handled.
[7] We alter the definition of valuation slightly here for the second algorithm.

In this case we must shift the clock configuration probabilities in the previous matrix down by one index step (which corresponds to δ time passing) and add the result to the matrix we are updating.

We also at this stage determine the new states that can be reached from the current state during the δ time passing, and the probability of entering these states. We do this by looking at all the clock configurations where at least one of the indices has the value one. If the clocks are set within such a configuration then we know that at least one clock will expire during the ensuing δ timestep.

If only one index in the configuration has the value one then only one clock can expire, and only one state can be entered from this clock configuration, so that state is added to the set of states that can be entered from the current state at the current time.

If more than one index in the configuration has the value one, then we simply do not go any further into the automaton and the configuration probability is added to *error*.

The second way to update a matrix corresponds to a transition from one state to another within the automaton. It is described in the procedure *new_state_matrix*. For each matrix entry we calculate the clock configuration probability, multiply it by the probability of moving into this state at this time, and add it to the matrix entry we are updating.

5.3.2 *Termination of an Iteration.* When the iteration terminates, it will output one of three results: true, false or undecided. true means that the automaton models the temporal formula: $SA \models [\phi_0 \, \mathcal{U}_{\leq c} \, \phi_1] > p$. false means that $SA \not\models [\phi_0 \, \mathcal{U}_{\leq c} \, \phi_1] > p$, and undecided means that the algorithm could not accumulate enough information to decide whether or not the automaton modelled the formula.

The algorithm makes the output decision based on the three global variables *total_pass*, *total_fail* and *error*.

total_pass is a lower bound on the probability that the stochastic automaton models the formula, and *total_fail* is a lower bound on the probability that the stochastic automaton does not model the formula. *error* is the largest amount by which *total_fail* or *total_pass* may be wrong. In a sense, it records the size of the uncertainty introduced by the choice of δ.

If neither of these situations holds then the errors introduced by the algorithm are too large to determine an answer with this value of δ. In this case, we can rerun the algorithm with a smaller δ, and in Section 7 we show that the sum of the errors tends to zero as δ tends to zero. Note, however, that in the case where the probability that SA models $[\phi_0 \, \mathcal{U}_{\leq c} \, \phi_1]$ is exactly p, we cannot guarantee that there will be a δ small enough to allow the algorithm to generate a true or a false. However, if this is not the case, by reducing δ, a solution will eventually be reached. This is ostensibly what is proved in Theorem 7.1. This is the sort of limitation that has to be accepted when working with generalised distributions. Although, in practice, we believe that such situations are likely to arise very rarely. Also note that our first algorithm does not have this limitation. Thus, it could potentially be used in situations in which this second algorithm fails to converge to a solution.

6. EXAMPLE

The second algorithm requires more stringent restrictions on the stochastic automaton than the first one, because the clock distribution functions must have positive lower bounds, (as opposed to the non-negative lower bounds required by the first). Therefore in order to illustrate the second algorithm, we will use the automaton in Figure 2, but alter slightly each of the clock distribution functions, by shifting each of them half a time unit to become

$$\begin{aligned} F_x(t) &= 2\left(t-\frac{1}{2}\right)-\left(t-\frac{1}{2}\right)^2, && \text{if } t \in \left(\frac{1}{2},\frac{3}{2}\right] \\ &= 0, && \text{if } t \le \frac{1}{2} \\ &= 1, && \text{otherwise} \end{aligned}$$

$$\begin{aligned} F_y(t) &= \left(t-\frac{1}{2}\right)^2, && \text{if } t \in \left(\frac{1}{2},\frac{3}{2}\right] \\ &= 0, && \text{if } t \le \frac{1}{2} \\ &= 1, && \text{otherwise} \end{aligned}$$

and

$$\begin{aligned} F_z(t) &= t-\frac{1}{2}, && \text{if } t \in \left(\frac{1}{2},\frac{3}{2}\right] \\ &= 0, && \text{if } t \le \frac{1}{2} \\ &= 1, && \text{otherwise} \end{aligned}$$

In this section, we will consider the temporal formula $[(a_0 \vee a_1)\,\mathcal{U}_{\le \frac{3}{2}} a_2] > \frac{1}{2}$, where $s_i \models a_i, i \in \{1,2,3\}$. We now illustrate this algorithm by applying it to the example.[8] We set δ equal to $\frac{1}{2}$.

Sections A, B and C below correspond to the sections A, B and C in the algorithm description in Appendix C. Within section C, line numbers correspond to the line numbers of the algorithm.

Section A. This section initialises all the variables to zero, and calculates all the probabilities of clocks falling in the ranges $(0,\delta]$, $(\delta,2\delta]$ and so on from the probability distribution functions entered as part of the stochastic automaton.

In our example, the probabilities that the clocks x, y and z are in the ranges $(0,\delta]$, $(\delta,2\delta]$ or $(2\delta,3\delta]$ are given by

	x	y	z
$(0,\delta]$	0	0	0
$(\delta,2\delta]$	$\frac{3}{4}$	$\frac{1}{4}$	$\frac{1}{2}$
$(2\delta,3\delta]$	$\frac{1}{4}$	$\frac{3}{4}$	$\frac{1}{2}$

[8]The type of situation where this algorithm would do very badly is if one clock has a very small lower bound and all the rest have very high lower bounds. This is accentuated if the first clock is hardly used. It might even be that the state where the first clock is used is unreachable or has a very low probability of being reached. Thus a criterion for the algorithm to work efficiently is that all pdf lower bounds are "similar."

These are easy to obtain from the clock probability distribution functions. Indeed, the ease of determining these probabilities is the main benefit of this algorithm and contrasts with the intractable manner in which the integrals explode in the first algorithm.

Section B. The initial state s_0 does not model a_1, but it does model the proposition a_0, and so the procedure *init_matrix* is called. This returns $matrix(s_0, 0)$ which is as follows

y			
3	0	$\frac{9}{16}$	$\frac{3}{16}$
2	0	$\frac{3}{16}$	$\frac{1}{16}$
1	0	0	0
	1	2	3 x

and is easily derivable from the probabilities above. The procedure also sets $live(0)$ to $\{s_0\}$.

If $c_{x_{\max}}$ is the upper bound of x, and $c_{y_{\max}}$ is the upper bound of y, there will be $\lceil c_{x_{\max}} \times \frac{1}{\delta} \rceil$ entries on the x axis, and $\lceil c_{y_{\max}} \times \frac{1}{\delta} \rceil$ entries on the y axis, so in this case (where $N_x = \frac{3}{2}$, $N_y = \frac{3}{2}$ and $\delta = \frac{1}{2}$), we get a 3×3 matrix.

This matrix tells us for example that when the clocks in the initial state are first set, the probability of clock x being set within the range $(\delta, 2\delta]$ and clock y being set within the range $(2\delta, 3\delta]$ is $\frac{3}{16}$. That is, for the clock configuration $\langle(\delta, 2\delta], (2\delta, 3\delta]\rangle$, the clock configuration probability is $\frac{3}{16}$.

Section C. We now enter the iterative part of the algorithm, where each iteration corresponds to increasing the time by one time unit (δ), and the snapshot produced at the end of iteration n corresponds to a view of the automaton at time $n\delta$. The three global probability values[9] are all still zero (lines 1-1a), so *ct* (current time) becomes δ. Only the state s_0 is live at time zero, so *new_time_matrix* is called (line 6) for $matrix(s_0, \delta)$. This returns a number of parameters: $matrix(s_0, \delta)$, $new\_states(s_1, \delta)$, *prob* and *error*.

The procedure *new_time_matrix* will return the $matrix(s_0, \delta)$ as

y			
3	0	0	0
2	$\frac{9}{16}$	$\frac{3}{16}$	0
1	$\frac{3}{16}$	$\frac{1}{16}$	0
	1	2	3 x

where each clock has advanced one time unit from $matrix(s_0, 0)$. So, at time δ, the probability of clock x being within the range $(0, \delta]$ and clock y being within the range $(\delta, 2\delta]$ is $\frac{9}{16}$.

[9]These are the probability values that are updated throughout the algorithm: *total_pass*, *total_fail* and *error*.

The probability of staying in state s_0 for at least half a time unit is 1; this follows from the fact that no clock can be set to less than δ ($\frac{1}{2}$ time unit). Thus $prob(s_0, \delta) = 1$.

None of the edge values (those with at least one clock in the range $(0, \delta]$) of the previous time matrix ($matrix(s_0, 0)$) is non-zero (so there is no possibility of any clock reaching zero and causing a transition to fire). The second half of the procedure (lines 10–23, which would determine the new states reached from state s_0) is therefore not executed and the global probability values (*total_pass*, *total_fail* and *error*) are all still zero. $new\_states(s_0, \delta)$ will be returned as {}, since no new states can be reached at time δ.

The next step (lines 7–11 of section C) is to calculate the live states at time δ, and since $remain(s_0, \delta) = true$ (it is possible to remain in state s_0 at time δ) we include s_0.

Since there are no states that can be reached from state s_0 in the time interval $(0, \delta]$, lines 12–22 of section C are not executed.

All of the global probability values are still zero, (i.e. we don't have enough information to decide the truth or falsity of the formula at this stage, lines 1-1a of Section C), and $2\delta \leq \frac{3}{2}$ (we have more time in which to gain more information, lines 2–3 of Section C), so we begin a second iteration.

On the second iteration of the while loop, *ct* is set to 2δ. Only s_0 was live at the last iteration ($live(\delta) = \{s_0\}$), so at line 6 we call *new_time_matrix* for $matrix(s_0, 2\delta)$.

This again returns a number of parameters, for example $matrix(s_0, 2\delta)$ becomes

y				
3	0	0	0	
2	0	0	0	
1	$\frac{3}{16}$	0	0	
	1	2	3	x

where the entry $matrix(s_0, 2\delta)(1, 1)$ is taken from the clock configuration $(\delta, 2\delta], (\delta, 2\delta]$ in the previous time matrix $matrix(s_0, \delta)$, thus the probability of staying in state s_0 in the interval $(\delta, 2\delta]$ is $\frac{3}{16}$. However this is not the final version of $matrix(s_0, 2\delta)$, because some of the clock configurations lead to transitions that lead back to state s_0.

All the other clock configurations ((1, 1), (1, 2) and (2, 1)) in $matrix(s_0, \delta)$ lead to transitions. Lines 10–22 of procedure *new_time_matrix* are executed for each of these three configurations.

For clock configuration (1, 1), clock x is (arbitrarily) chosen to fire, and we assume that the adversary *pick* chooses the action *conc*, leading to state s_1. Line 13a of the procedure adds state s_1 to $new\_states(s_0, 2\delta)$, and $prob(s_1, 2\delta)$ becomes $\frac{3}{16}$ (line 14). Clock configuration (1, 1) is one where some error may be introduced into the algorithm result. Choosing clock x and action *conc* meant that we go to a state where the formula *TL* can still be true, but choosing the other clock may not lead to such a state. We therefore allow for the possible error introduced here by adding the clock configuration probability to *error*, which

becomes $\frac{3}{16}$. Clock configurations $(1, 2)$ and $(2, 1)$ are dealt with similarly, but they do not cause any addition to *error* since from these clock configurations only one clock can expire in the next interval.

Now, the *new_time_matrix* procedure is finished, and lines 7–11 of Section C determine the value of $live(2\delta)$ which is $\{s_0, s_1, s_2\}$, because at time 2δ the automaton may be in any state.

Lines 12-22 of Section C consider each new state that can be reached in time interval $(\delta, 2\delta]$. State s_0 still allows the temporal logic formula to be true, and so procedure *new_state_matrix* is called (line 17). However, $prob(s_0, 2\delta) = 0$, and therefore $matrix(s_0, 2\delta)$ is not altered.

State s_1 still allows the temporal logic formula to become true (line 13) and so procedure *new_state_matrix* is called (line 17). The probability of entering state s_1 in this interval is $\frac{9}{16}$, so $matrix(s_1, 2\delta)$ is

0	$\frac{9}{32}$	$\frac{9}{32}$	
1	2	3	z

In state s_2 the formula is true, and so $prob(s_2, 2\delta)$ $(\frac{1}{16})$ is added to *total_pass* (line 14).

In the final iteration, the global probability values become: $total\_pass = \frac{1}{16}$, $total\_fail = \frac{9}{16}$ and $error = \frac{6}{16}$. The iterations stopped because the value of time became too large—not because the global probabilities contained enough information to make a decision. This means that *total_pass* $(\frac{1}{16})$ is a maximum possible probability value of the formula $[(a_0 \vee a_1)\, \mathcal{U}_{\leq \frac{3}{2}}\, a_1]$ (with any clock ordering) and $total\_pass - error$ $(-\frac{5}{16})$ is a minimum possible probability value.

Thus, since we wish to determine whether the actual probability value is greater than $\frac{1}{2}$, the algorithm will output fail.

If we were interested in a similar formula with a probability value in the range $[0, \frac{1}{8}]$, we could reduce the size of δ, and take snapshots (e.g.) every $\frac{1}{4}$ time unit. This (for the reasons outlined in Section 7) will reduce the size of the *error* variable.

6.1 Unbounded Until Formulae

As just presented the second algorithm only handles until formulae of the form

$$[\phi_1\, \mathcal{U}_{\leq c}\, \phi_2] \simeq p$$

however a combination of the second and first algorithms yields a method to verify unbounded until formulae, those of the form

$$[\phi_1\, \mathcal{U}_{> c}\, \phi_2] \simeq p$$

The basic idea is to observe that verification of a formula such as $\phi_1\, \mathcal{U}_{> c}\, \phi_2$ can be split into a conjunction of separate verifications

(a) Check that ϕ_1 holds at all times until c time units have elapsed; and
(b) Check that there exists an $X > c$ such that ϕ_2 holds at time X, and that ϕ_1 holds for all times strictly greater than c and less than X.

Thus, we can model check formulae such as $[\phi_1 \mathcal{U}_{>c} \phi_2] \simeq p$ in the following way.

(i) Run (the obvious slight adaption of) the second algorithm to check that (a) holds. This will finish with a certain amount of probability mass in the variable *total_fail* and no probability mass in *total_pass*. The reason for the latter is that pass states can only be revealed once time has passed beyond c. In addition, $live(c)$ will indicate the locations that are still undecided—from which we must explore further.

(ii) Run the first algorithm using $live(c)$ as the starting locations and the initial timing regions determined from the remaining matrices (this can be done in a straightforward manner). However, notice that running the first algorithm in this situation does not incur the problems of intractability that it does in the general case. Specifically, since the time bound on the until has been satisfied we ostensibly only have an untimed until verification. Consequently probabilities can be assigned to nodes without requiring the global clock to be taken into account thus, they can be evaluated "locally." Hence, the exponential explosion in the number of integrals to be considered does not occur.

7. CORRECTNESS AND CONVERGENCE

For a single run with fixed δ, we wish to prove two things: that the algorithm terminating with pass implies that the automaton models the formula, and that the algorithm terminating with fail implies that the automaton does not model the formula.

If the algorithm outputs pass then the variable *total_pass* must be greater than p (where p is taken from the temporal formula $[\phi_0 \mathcal{U}_{\leq c} \phi_1] > p$). The only place where *total_pass* gets incremented is line 14 of section C (see full algorithm in Appendix C). If the current state q models ϕ_1 (and all previous states in the path model ϕ_0) we add the probability of entering the state q at the current time point. If the sum of these probabilities is greater than p then the algorithm outputs pass.

We will consider the case when the algorithm outputs pass. Consider the initial state. Note that for any clock configuration, the probability of all paths that commence with the clocks being set somewhere within this configuration is equal to the clock configuration probability. Furthermore, for an arbitrary state s and time c and configuration, the probability of all paths that go through this configuration at this time is the probability of the configuration multiplied by the probability of reaching that state at that time.

The probability of reaching state s at time c is the 2nd parameter of procedure *new_state_matrix*.[10]

If every valuation in a configuration corresponds to the same automaton transition, and this transition is the final one in a path that models the formula,

[10] In fact, it is greater than or equal to this sum, because some routes through the transition system may have already passed or failed the formula, and therefore would not be considered further by the algorithm.

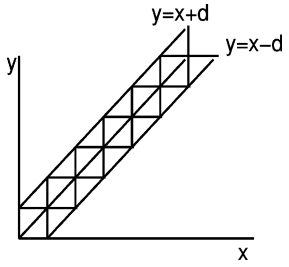

Fig. 5. Upper bound on error with clocks x and y. (In this diagram d denotes δ.)

then we add the clock configuration probability (multiplied by the probability of reaching that state at that time) to *total_pass*.

This is the only way in which the algorithm adds to the variable *total_pass*. Since the algorithm only outputs pass if *total_pass* is greater than the formula probability p, it is clear that the algorithm will only output pass if the automaton models the formula.

If more than one clock in the configuration is in the range $(0, \delta]$ then more than one of the clocks will have reached time 0 in the interval we are considering, so the clock configuration probability is added to *error* (line 12 of procedure *new_time_matrix*).

A similar argument applies in the case where the algorithm outputs fail.

Therefore the algorithm is sound in the sense that if we are given a definitive answer, this answer is correct. There remains, of course, the question of convergence to the correct answer; the following theorem summarises the situation.

THEOREM 7.1. *For every automaton SA and propositions ϕ_0 and ϕ_1 it is the case that if SA models $[\phi_0 \mathcal{U}_{\leq c} \phi_1]$ with probability p, then for any error e greater than zero, there is a timestep δ greater than zero such that for the formula $[\phi_0 \mathcal{U}_{\leq c} \phi_1] > q$, the algorithm will only return* undecided *if $q \in [p-e, p+e]$.*

First note that n independent single variable continuous probability distribution functions $f_1 \ldots f_n$ can always be combined to give a single n variable probability distribution function that is continuous in all dimensions: $f(x_1 \ldots x_n) = f_1(x_1) \times \cdots \times f_n(x_n)$.

For convenience, consider a location with two outgoing transitions and two clocks x and y with distribution functions f_x and f_y. Because f_x and f_y are both continuous, if we set $f(x, y) = f_x(x) \times f_y(y)$ we can (by the note above) say that

$$\forall \epsilon > 0. \exists \delta > 0. f(x, x+\delta) - f(x, x-\delta) < \epsilon$$

We will show that for any desired size of error we can choose a suitably small timestep.

Now, $\int_0^m f(x, x+\delta) - f(x, x-\delta)\, dx$[11] (the probability of the clock valuation falling between the two 45 degree lines in Figure 5) is greater than the sum of all contributions to the error variables (represented by the squares in the figure). Since the number of locations in the stochastic automaton is finite

[11] $m = \min\{x_{\max}, y_{\max}\}$, where $x_{\max}$ is the largest value to which clock x can be set.

(say N_s) and (for bounded until formulas with less than subscripts) the maximum number of visits to any location is finite (say N_v) for any desired error e we must ensure that, for every location, for the multivariate function associated with that location, we choose ϵ such that $\epsilon < \frac{e}{N_s \times N_v}$. If the timestep is set to the smallest δ necessary to ensure that every location provides errors less than $\frac{e}{N_s \times N_v}$, then total error provided by one location (over all time) will be less than $\frac{e}{N_s}$ and the total error provided by all locations will be less than e.

8. COMPLEXITY MEASURES

8.1 Time Complexity

The time complexity of the algorithm discussed in Section 5 depends on a number of factors, namely δ, t, n_1, n_2 and $|\mathcal{S}|$. The explanation of these parameters is as follows:

—t is the value of time given in the time-bounded until formula: $[a\, \mathcal{U}_{\leq t} b] \sim p$;
—δ is the chosen timestep;
—$|\mathcal{S}|$ is the number of states in the automaton;
—n_1 is the largest number of clocks in a single state and
—n_2 is the largest (positive finite) upper bound of all the clocks.

An upper bound on the number of matrices that need to be built in a single iteration is $|\mathcal{S}|$, where $\mathcal{S}$ is the set of all states in the automaton.

To calculate the time complexity we also need to calculate the size of the largest matrix. Each matrix is multi-dimensional, and $\frac{n_2}{\delta}$ will be the maximum number of entries over all matrices and all dimensions. For example, in the example in Section 6 all the matrices had 2 dimensions and the maximum number of entries in any dimension was 3 since $\delta = \frac{1}{2}$ and $n_2 = \frac{3}{2}$.

An upper bound on the size of the largest matrix will therefore be the number of elements in the largest dimension, raised to the power of the largest number of clocks on a single state.

The time complexity is thus bounded by the time taken to update all the possible matrices in each iteration of the while loop in the algorithm, multiplied by the maximum number of iterations the algorithm will perform in the worst case. This latter value is $\frac{t}{\delta}$, therefore the time complexity is

$$\frac{t}{\delta} \times \left(\frac{n_2}{\delta}\right)^{n_1} \times |\mathcal{S}|$$

Although this is exponential, the exponent n_1 should in general be fairly small ($\leq$3) because we only allow clocks to be used from the state in which they are set.

In fact, the algorithm could be optimised to provide a better time complexity, by limiting the size of the matrices to $\min(\frac{t}{\delta}, \frac{n_2}{\delta})$ since there is no need to consider the operation of the clock beyond the limit set by the time bound on the temporal formula. The size of the largest matrix would therefore be less than $(\min(\frac{t}{\delta}, \frac{n_2}{\delta}))^{n_1}$, where n_1 is the largest number of clocks in a single state.

An upper bound on the time complexity would therefore be

$$\frac{t}{\delta} \times \left(\min \left(\frac{t}{\delta}, \frac{n_2}{\delta} \right) \right)^{n_1} \times \mid \mathcal{S} \mid$$

The time complexity also relies heavily on δ, and the bigger the δ the lower the time complexity. To see the relationship with δ, note that the upper bound can be rewritten as

$$\left(\frac{1}{\delta} \right)^{n_1+1} \times t \times (n_2)^{n_1} \times \mid \mathcal{S} \mid$$

8.2 Space Complexity

An upper bound on the space complexity will be proportional to the product of the size of the biggest matrix and the largest number of matrices that need to be stored at one time. The size of the largest matrix is less than $(\frac{n_2}{\delta})^{n_1}$, (from time complexity calculations) and the largest number of matrices that need to be stored at any one time is twice the number of states in the automaton, $2 \times \mid \mathcal{S} \mid$. The upper bound on space complexity is therefore

$$2 \times \left(\frac{n_2}{\delta} \right)^{n_1} \times \mid \mathcal{S} \mid$$

9. CONCLUSIONS AND FURTHER WORK

In this article we have presented two algorithms for model checking bounded until formulae against stochastic automata. Both of these algorithms allow systems to be described using continuous probability distributions, and we believe that this represents an important advance. We are not aware of any other published model-checking algorithm for stochastic automata.

The principal advantage of the first algorithm is its generality: the clocks may be set according to any function, providing the corresponding probability density function is integrable. The major drawback of the algorithm is its complexity: with every new unfolding of the probabilistic region tree not only does the number of nodes to be considered increase, but also the number of integrations required to determine the probability on a single node increases exponentially.

The principal advantage of the second algorithm is its efficiency: the discretisation of the probability functions means that the required calculations are considerably simpler. A limitation in comparison to the first algorithm is that the probability distributions must have a finite lower bound. Consequently, there are some situations (i.e. when distribution functions do not fall into this category) in which, effectively, the only algorithm we offer is the first and its inefficiency has to be accepted.

It should also be acknowledged that with some varieties of automata, discretization can explode in size, in particular, if there is a great difference in the lower bound for different distributions it could be that δ is picked to be very small in relation to some distribution functions. This can have the consequence that a long sequence of unfoldings could be generated in which the

state of the system does not change.[12] However such difficulties are inherent in discretization approaches. An interesting topic for further work would be to develop "syntactic-level" techniques for analysing the stochastic automaton input in order to determine whether such long idling phases are likely to arise. If they are likely to arise, then the first algorithm could be tried instead of the second.

In addition, an advantage of both algorithms is that, since the "complete" model is not generated at any point the state space explosion (which typically hinders model checking) is contained. In particular, all data structures apart from those which reflect undecided nodes (i.e. u labelled regions in the first algorithm and *live* locations in the second algorithm) can be deleted. In this sense the algorithms yield a form of on-the-fly exploration—only keeping information about the "leaves" of the exploration tree.

A prototype implementation of the second algorithm has been developed [Bowman et al. 2000]. Our experience using ths implementation has been positive, although there are many performance gains that we have as yet been unable to implement.

Further work on the second algorithm will include relaxing the restrictions imposed on the stochastic automata, particularly the ability to set and use clocks anywhere in the automaton. Being able to do this would allow parallel composition.

It would also be good to increase the expressiveness of the logic, allowing nested untils and to extend the model checking algorithm itself to allow queries such as "what is the probability of $[\phi_0 \, \mathcal{U}_{\leq c} \, \phi_1]$?" and receive a probability value for an answer.

APPENDIX

A. SEMANTICS

A.1 Probabilistic Transition Systems

The definition of the semantics of stochastic automata is given in terms of probabilistic transition systems. The definition of probabilistic transition systems is reproduced from D'Argenio et al. [1998].

$\mathbf{N}$ is the set of non-negative integers. $\mathbf{R}$ is the set of real numbers, and $\mathbf{R}_{\geq 0}$ the set of non-negative reals. For $n \in \mathbf{N}$, let $\mathbf{R}^n$ denote the nth cartesian product of $\mathbf{R}$. $\mathbf{R}^0 \stackrel{\text{def}}{=} \{\emptyset\}$.

A *probability space* is a structure $(\Omega, \mathcal{F}, P)$ where Ω is a *sample space*, $\mathcal{F}$ is a *σ-algebra* on Ω and P is a *probability measure* on $\mathcal{F}$. In this work, as in D'Argenio et al. [1998], we consider only probability spaces isomorphic to some Borel space defined in a real hyperspace, whose coordinates come from independent random variables. We denote by $\mathcal{R}(F_1, \ldots F_n)$ the probability space $(\mathbf{R}^n, \mathcal{B}(\mathbf{R}^n), P_n)$ where $\mathcal{B}(\mathbf{R}^n)$ is the Borel algebra on $\mathbf{R}^n$ and P_n is the probability measure obtained from $F_1 \ldots F_n$, a given family of distribution functions. See Shiryayev [1984] for details.

[12] Intuitively this corresponds to ticking the clock while nothing is changing in the system. We are just waiting for the next interesting event to come along.

Let $\mathcal{P} = (\Omega, \mathcal{F}, P)$ be a probability space. Let $\mathcal{D} : \Omega \to \Omega'$ be a bijection. We lift $\mathcal{D}$ to subsets of Ω: $\mathcal{D}(A) \stackrel{\text{def}}{=} \{\mathcal{D}(a) \mid a \in A\}$ and define $\mathcal{F}' \stackrel{\text{def}}{=} \{\mathcal{D}(A) \mid A \in \mathcal{F}\}$. Now, it is clear that $\mathcal{D}(\mathcal{P}) \stackrel{\text{def}}{=} (\Omega', \mathcal{F}', P \circ \mathcal{D}^{-1})$ is also a probability space. Since $\mathcal{D}(\mathcal{P})$ is basically the same probability space as $\mathcal{P}$, we say that $\mathcal{D}$ is a *decoration* and we refer to $\mathcal{D}(\mathcal{P})$ as *the decoration of $\mathcal{P}$ according to $\mathcal{D}$*. This is used when we come to give a semantics to stochastic automata.

Definition A.1. Let $P(H)$ denote the set of probability spaces $(\Omega, \mathcal{F}, P)$ such that $\Omega \subseteq H$. A *probabilistic transition system* is a structure $\mathcal{T} = (\Sigma, \Sigma', \sigma_0, \mathcal{L}, T, \longrightarrow)$ where

(1) Σ and Σ' are two disjoint sets of *states*, with the *initial state* $\sigma_0 \in \Sigma$. States in Σ are called *probabilistic states* and states in Σ' are called *non-deterministic* states.
(2) $\mathcal{L}$ is a set of *labels*.
(3) $T : \Sigma \to P(\Sigma')$ is the *probabilistic transition relation*.
(4) $\longrightarrow \subseteq \Sigma' \times \mathcal{L} \times \Sigma$ is the *labelled (or non-deterministic) transition relation*. We use $\sigma' \stackrel{l}{\longrightarrow} \sigma$ to denote $\langle \sigma', l, \sigma \rangle \in \longrightarrow$, $\sigma' \stackrel{l}{\not\longrightarrow}$ for $\neg\exists \sigma \cdot \sigma' \stackrel{l}{\longrightarrow} \sigma$ and $\sigma' \longrightarrow \sigma$ for $\exists l \cdot \sigma' \stackrel{l}{\longrightarrow} \sigma$.

Since we are interested in timed systems, we set $\mathcal{L} = \mathbf{A} \times \mathbf{R}_{\geq 0}$, where $\mathbf{A}$ is a set of action names. A timed action transition will be described as $a(d)$, which indicates that the action a occurs exactly d time units after the system has been idling.

Definition A.2. A *valuation* is a function $v : \mathcal{C} \to \mathbf{R}_{\geq 0} \bigcup \{\perp\}$ such that $v(x) \leq x_{\max}$, where $x_{\max}$ is the maximum value to which clock x can be set. The set of all valuations is $\mathcal{V}$. If $d \in \mathbf{R}_{\geq 0}$, $v - d$ is defined by $\forall x \in \mathcal{C}.(v - d)(x) \stackrel{\text{def}}{=} v(x) - d$. We assume the set of clocks is ordered so, if $C \subseteq \mathcal{C}$, we can write $\vec{C}$ for the ordered form of C and $\vec{C}(i)$ for the i-th element. Let $C \subseteq \mathcal{C}$, $n = \#C$, and $\vec{D} \in \mathbf{R}^n$. We define $v[\vec{C} \leftarrow\!\shortmid \vec{D}]$ by

$$v[\vec{C} \leftarrow\!\shortmid \vec{D}](x) \stackrel{\text{def}}{=} \begin{cases} \vec{D}(i) & \text{if } x = \vec{C}(i), \quad \text{for some } i \in \{1, \ldots, n\} \\ \perp & \text{otherwise} \end{cases}$$

This definition will be used when we explain how clock values change as states change. It differs from the definition given in D'Argenio et al. [1998] because there clocks not in the set $\mathcal{C}$ maintain their values through this operation. This is because in D'Argenio et al. [1998] clocks may be used to trigger actions in any state, not just the state in which they are set. In this work, however, in order to simplify the model checking, we insist that clocks are only used in the states in which they are set, and therefore there is no need to remember their value once the state has been exited.

The main obstacle now in constructing the probabilistic transition system semantics is in showing how the clock probability functions are used to construct the probability spaces. We do this by defining a decoration function, discussed in Section A.1.

Let $SA = (\mathcal{S}, s_0, \mathcal{C}, \mathbf{A}, \longrightarrow\!\!\triangleright, \kappa, F)$ be a stochastic automaton. Let s be a location in $\mathcal{S}$ and $n = \#\kappa(s)$. Let v be a valuation in $\mathcal{V}$. Let $\mathcal{V}' = \{v[\overrightarrow{\kappa(s)} \leftarrow\!\!\dashv \vec{D}] \mid \vec{D} \in \mathbf{R}^n\} \subseteq \mathcal{V}$. We define the decoration function $\mathcal{D}_v^s : \mathbf{R}^n \to \{s\} \times \mathcal{V}' \times \{1\}$ by $\mathcal{D}_v^s(\vec{D}) \stackrel{\text{def}}{=} (s, v[\overrightarrow{\kappa(s)} \leftarrow\!\!\dashv \vec{D}], 1)$. Notice that $\mathcal{D}_v^s$ is a bijection. In the next definition, we use the probability space $\mathcal{R}(F_{x_1}, \ldots, F_{x_n})$ decorated according to some $\mathcal{D}_v^s$.

Definition A.3. Let $SA = (\mathcal{S}, s_0, \mathcal{C}, \mathbf{A}, \longrightarrow\!\!\triangleright, \kappa, F)$ be a stochastic automaton. The actual behaviour of SA is given by the PTS $I(SA) \stackrel{\text{def}}{=} ((\mathcal{S} \times \mathcal{V} \times \{0\}), (\mathcal{S} \times \mathcal{V} \times \{1\}), (s_0, \mathbf{0}, 0), \mathbf{A} \times \mathbf{R}_{\geq 0}, T, \longrightarrow)$, where in the initial valuation $\mathbf{0}$ clock a is set to some natural number (chosen according to the PRTL function, see Section 3), and each other clock is undefined. T and $\longrightarrow$ are defined as follows:

$$\frac{\overrightarrow{\kappa(s)} = \{x_1, \ldots, x_n\}}{T(s, v, 0) = \mathcal{D}_v^s(\mathcal{R}(F_{x_1}, \ldots, F_{x_n}))} \ \textbf{Prob}$$

$$\frac{\begin{array}{c} s \stackrel{a,\{x\}}{\longrightarrow\!\!\triangleright} s' \wedge d \in \mathbf{R}_{\geq 0} \wedge (v - d)(x) \leq 0 \\ \forall d' \in [0 \cdot d) \cdot \forall s' \cdot s \stackrel{b,\{y\}}{\longrightarrow\!\!\triangleright} s' \cdot (v - d')(y) > 0 \end{array}}{(s, v, 1) \stackrel{a(d)}{\longrightarrow} (s', (v - d), 0)} \ \textbf{Act}$$

Within a stochastic automaton, two forms of uncertainty may arise. One is the probabilistic uncertainty associated with the clock-setting. Although we know which clocks are to be set, the choice of values for these clocks is probabilistic. This is where the stochastic element of the model arises, and is defined by rule **Prob**. The other is the nondeterministic uncertainty that arises if two actions are simultaneously able to be performed, and is defined using the rule **Act**. This nondeterminism is resolved using an adversary (Definition A.6).
Definition of a PTS-path:

Definition A.4. A *PTS-path* is a *finite* or *infinite* sequence of states

$$\langle \sigma_0, \sigma_0', \sigma_1, \sigma_1', \ldots \rangle$$

where, σ_0 is the initial state, for each σ_i', there exists a probability space $(S, \mathcal{F}, P)$ such that $T(\sigma_i) = (S, \mathcal{F}, P)$, $\sigma_i' \in S$ and $\sigma_i' \longrightarrow \sigma_{i+1}$.

Definition A.5. An *SA-path* is a finite or infinite sequence

$$\langle (s_0, v_0), (s_0, v_0'), (s_1, v_1), (s_1, v_1'), \ldots, (s_n, v_n), (s_n, v_n'), \ldots \rangle$$

such that

—v_0 means no clocks are set.
—$v_i' \in \mathcal{R}(F_{x_1}, \ldots, F_{x_n})$ where $T(s_i, v_i, 0) = \mathcal{D}_v^s(\mathcal{R}(F_{x_1}, \ldots, F_{x_n}))$. Each valuation v_i' is a possible result of the clock setting functions.
—$(s_i, v_i', 1) \stackrel{a(d)}{\longrightarrow} (s_{i+1}, v_{i+1}, 0)$ for some d. Timed action transitions must be allowed by the SA.
—Finite paths end on a probabilistic state.

An SA-path is like a run of the SA expanded with clock values.

Definition A.6. An *adversary* of an SA is a function mapping sequences of states to states

$$adv :< s_0, s_1, \ldots, s_n > \longrightarrow s_{n+1}$$

such that $< s_0, s_1, \ldots, s_n, s_{n+1} >$ is a run of the SA.

Note that adversaries do not make any reference to time.

With an adversary, an SA becomes deterministic. The corresponding PTS contains no nondeterminism either.

If

$$\sigma = \langle (s_0, \mathbf{0}), (s_0, v_0'), (s_1, v_1), (s_1, v_1'), \ldots, (s_k, v_k), (s_k, v_k') \rangle$$

is a finite SA-path, then $\sigma[i] = s_i$ and $\sigma(x)$ is the state at time x.

$\mathcal{R}(F_{x_1}, \ldots, F_{x_n})$ is the Borel space $(\mathbf{R}^n, \mathcal{B}(\mathbf{R}^n), P_n)$ where P_n is the unique probability measure obtained from $\mathcal{R}(F_{x_1}, \ldots, F_{x_n})$.

Now, for all $j < k$, set A_j to be the maximal set of valuations equivalent to v_j which lead to state s_{j+1}.

Let

$$C(s_0, A_0, s_1, \ldots, s_{k-1}, A_{k-1}, s_k)$$

denote the *cylinder set* which contains all paths starting at s_0 and going through all states $s_j (j \leq k)$ and valuation sets $A_j (j \leq k)$.

The probability measure Pr on $\mathcal{F}(Path(s_0))$[13] is identified by induction on k by $Pr(C(s_0)) = 1$ and for $k \geq 0$:

$$Pr(C(s_0, A_0, \ldots, A_k, s_{k+1}) = Pr(C(s_0, A_0, \ldots, A_{k-1}, s_k)) \cdot P(A_k)$$

where $P(A_k)$ is the probability of the set A_k, and is taken from the relevant Borel space.

A.2 PRTL Semantics

In this section, we introduce the semantics for the temporal logic PRTL.

To facilitate model checking, we use Probabilistic Transition Systems as a semantic model for the definition of PRTL. In order to do this we must resolve two problems. The first is that PRTL is a *real-time* logic—it enables reference to specific instants in time—and the abstract definition of PTSs D'Argenio et al. [1998] does not contain reference to time. This is easily solved—we simply use the PTS generated by a Stochastic Automaton. This contains much more detailed state information, in particular, the values of clocks.

The second problem is that the PTS contains nondeterministic information, and this nondeterminism must be resolved before we can use the PTS to assign a semantics to our logic. We do this using adversaries.

Recall the syntax of PRTL:

$$\psi ::= \mathsf{tt} \mid \mathsf{ap} \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid [\phi_1 \, \mathcal{U}_{\sim c} \, \phi_2] \simeq p$$
$$\phi ::= \mathsf{tt} \mid \mathsf{ap} \mid \neg\phi \mid \phi_1 \wedge \phi_2$$

[13] $Path(s_0)$ is all paths possible from s_0, and $\mathcal{F}(Path(s_0))$ is the smallest σ−algebra on $Path(s_0)$.

where $c \in \mathbf{N}$, a is an atomic proposition, $p \in [0, 1]$ is a probability value and $\sim, \simeq \in \{<, >, \leq, \geq\}$.

The *path formulae* ψ can only be used at the outermost level—they cannot be nested. This is because the model checking algorithms only evaluate path formulae from the initial state.

Definition A.7. If $SA = (\mathcal{S}, s_0, \mathcal{C}, \mathbf{A}, \longrightarrow\!\!\triangleright, \kappa, F)$ is a Stochastic Automation and $PTS = (\Sigma, \Sigma', \sigma_0, \mathcal{L}, T, \longrightarrow)$ is the resulting Probabilistic Transition System, then $\Sigma(= \Sigma') \subseteq \mathcal{S} \times \mathcal{V}$, $\mathcal{L} \subseteq A \times \mathbf{R}_{\geq 0}$ and $\sigma_0 = (s_0, \mathbf{0})$. We must also introduce a function ξ, which maps SA locations to the logical propositions true in that location.

We only need to use the probabilistic states to define the logic, since once a probabilistic state has been entered the behaviour of the automaton is completely determined until the first clock expires.

The simple formulae ϕ are defined in the conventional way for each probabilistic region σ', but the until formulae ψ are defined only for the initial region σ_0. The model checking algorithm does not yet allow path formulae to be established for an arbitrary region.

- $s \models \mathrm{tt}$
- $s \models a$, provided $a \in \xi(s)$
- $s \models \phi_1 \wedge \phi_2$, provided $s \models \phi_1$ and $s \models \phi_2$
- $s \models \neg\phi$, provided $s \not\models \phi$

If σ is an SA-path, and ψ a path formula then

- $\sigma \models [\phi_1 \,\mathcal{U}\, \phi_2]$ iff $\exists k \geq 0 \cdot (\sigma[k] \models \phi_2 \wedge \forall 0 \leq i \leq k \cdot \sigma[i] \models \psi_1)$
- $\sigma \models [\phi_1 \,\mathcal{U}_{\sim t} \phi_2]$ iff $\exists x \sim t \cdot (\sigma(x) \models \phi_2 \wedge \forall y \in [0, x) \cdot \sigma(y) \models \psi_1)$

and

- $PTS \models [\phi_1 \,\mathcal{U}_{\sim t} \phi_2] \simeq p$ iff $Prob(s_0, \phi_1 \,\mathcal{U}_{\sim t} \phi_2) \simeq p$ where $Prob(s_0, \psi) \stackrel{\text{def}}{=} Pr\{\rho \in Path(s_0) \mid \rho \models \psi\}$

Thus, the Probabilistic Transition System PTS models $[\phi_1 \,\mathcal{U}_{\sim t} \phi_2] \simeq p$ provided $Prob(s_0, \phi_1 \,\mathcal{U}_{\sim t} \phi_2) \simeq p$.

B. FIRST ALGORITHM

Here, we give the definition of the first model checking algorithm for bounded until formulae. We will consider a PRTL formula of the form $[\phi_1 \;\mathcal{U}_{<c}\, \phi_2] > p$. "less than p" queries may be handled in a similar way.

Assume an adversary Adv, and that each SA location is mapped to either ϕ_1 or $\neg\phi_1$ and to either ϕ_2 or $\neg\phi_2$. Note that the algorithm can easily be extended to the more general case where locations contain sets of atomic propositions.

Add the (new) clock a to the set of all clocks.
Construct the PRG node $(s_0, \mathbf{0}_c)$.
Set $s = s_0$.
If $s \not\models \phi_1$ then stop with no, else

REPEAT

For each valuation equivalence class $[v_i]$ from $\kappa(s) \bigcup \{a\}$, form the node $(s, [v_i])$.

For each new node $(s, [v_i])$ choose a subsequent non-deterministic node $(s_j, \perp)$ according to the adversary Adv.

For each new non-deterministic node $(s_j, \perp)$

label 'p' if $s_j \models \phi_2$ and $v(a) > 0$.

label 'f' if $s_j \not\models \phi_1$ or $s_j \not\models \phi_2$ or $v(a) \leq 0$.

label 'u' otherwise

For each node labelled with either 'p' or 'f', calculate the probability of the corresponding path.

If $\Sigma_{\mathrm{p}} pr(s, [v]) > p$ then stop with yes.

If $\Sigma_{\mathrm{f}} pr(s, [v]) > 1 - p$ then stop with no.

Otherwise, repeat for each node labelled 'u'.

C. SECOND ALGORITHM

In this section we present a detailed description of the algorithm. It is divided into Section A (which initialises variables), Section B (the initial part of the algorithm) and Section C (the iterative part). Procedures used are described at the end.

The lines of code are prefaced with numbers, and the comments are delimited with double stars.

Section A

Model_check(*SA, Formula*, δ, *pick*)

note that the function *pick* is the adversary, used in procedure *new_time_matrix*.

We are assuming a TL formula of the form $[a_0 \ \mathcal{U}_{\leq t} a_1] \geq p$.

The $\geq p$ could easily be changed; the $\leq t$ is hardwired into the algorithm.

** **

We begin by initialising variables.

***ct*: (integer) current_time**

$ct := 0$

***total_pass* and *total_fail* are reals in [0, 1].**

At any point in the algorithm, *total_pass* is the accumulated

probability of all the passed paths and *total_fail* is the accumulated

probability of all the failed paths. We initialise them both to zero.

$total\_pass := 0$

$total\_fail := 0$

***error* is a real in [0, 1]. It is the accumulated probability of all paths**

which, because of the discretisation of the algorithm, we cannot determine exactly.

This is where the revised version of the algorithm differs from the initial one.

It is initialised to zero.

** **

$error := 0$

***prob*(s, t) is the probability of moving (from anywhere) to location s**

at time t. (i.e. in interval $(t - \delta, t]$.)
For all combinations of locations and times, we initialise $prob$
to zero.
$\forall s \in S. \forall i \leq n.$
  $prob(s, \delta i) := 0$
$remain(s, t)$ is a boolean which is true if the probability of remaining
in location s during time interval $(t - \delta, t]$ is non-zero, false otherwise.
They are all initialised to false.
$\forall s \in S. \forall i \leq n.$
  $remain(s, \delta i) := false$
$live(t)$ is the set of locations "active" at the end of
interval $(t - \delta, t]$, which
we need for calculating the information for the next time interval.
For all time values, we initialise $live$ to the emptyset.
$\forall i \leq n.$
  $live(\delta i) := \emptyset$
We initialise all values in all matrices to zero.
The are n_s clocks in location s.
$\forall s \in S.$
  $\forall 0 \leq j \leq n.$
    $\forall 1 \leq i_1 \leq N_1$
      $\vdots$
    $\forall 1 \leq i_{n_s} \leq N_{n_s}. matrix(s, \delta j)[i_1 \ldots i_{n_s}] := 0$

call procedure for calculating probabilities of clocks falling in the ranges
$(0, \delta], (\delta, 2\delta]$ etc. This comes directly from the clock PDFs,
and is only calculated once. It is needed for determining the clock
probabilities.
C is the set of all clocks and F is the set of clock probability functions
This procedure returns pr, which is needed in new_state_matrix
and $init\_matrix$.
$clock\_config\_probs(C, F, \delta, pr)$
** **

Section B
Consider initial location of SA: s_0
If $s\_0 \models a\_1$ then formula is trivially true.
if $s\_0 \models a_1$ then
  $total\_pass := 1$
If $s\_0 \models a\_0$ then formula is undecided and we must
unfold SA further.
elseif $s\_0 \models a_0$ then
  Build the initial matrix, i.e. $matrix(s\_0, 0)$.
  This will then contain the probabilities
  of all the different clock settings for location s_0 at time zero.
  $init\_matrix(matrix(s\_0, 0))$
  The only location "live" at time zero will be s_0.

```
   live(0) := {s_0}
  **If s_0 does not model a_0 or a_1 then formula is trivially false.**
  else
   total_fail := 1
  end if

  **Section C**
  **Each iteration of the following loop unfolds the automaton by**
  **one time step of δ. States which cause the formula to**
  **pass/fail are pruned from the tree, and their probabilities added to**
  **total_pass/total_fail, while the undecided states are recorded**
  **for the next iteration.**
  **We continue while the values of total_pass, total_fail and error**
  **are not enough to determine whether the formula is true or false**
1:  repeat
    **Increment current_time**
2:  ct := ct + δ
      **for all states s that were live at the last clock tick**
4:    ∀s ∈ live(ct − δ)
        **set current_state to s.**
5:      cs := s
        **The procedure new_time_matrix returns**
        **matrix(cs, ct): the matrix for the current state at the current time.**
        **It also**
        **updates the function prob with the probability of remaining**
        **in the current state at the current time and the probabilities of**
        **moving to different states at the current time.**
        **It also updates the value of error.**
6:      new_time_matrix(matrix(cs, ct), new_states(cs, ct), remain(cs, ct),
        prob, error)
        **If the probability of remaining in current state at current time is zero**
7:      if remain(cs, ct) = false then
          **current state is not live at current time and**
          **only the states which can be reached from current state at current
            time**
          **are added to those live at current time**
8:        live(ct) := live(ct) ⋃ new_states(cs, ct)
9:      else **remain(cs, ct) = true**
          **The current state, plus all states which may be reached from it at**
          **the current time, must be added to the live states.**
10:       live(ct) := live(ct) ⋃ {cs} ⋃ new_states(cs, ct)
11:     end if
11a:  end forall **∀s ∈ live(ct − δ)**
  **Now, we have live(ct) and prob(cs, ct) for all cs in live(ct)**
  **i.e. all the states we could be in at time ct, and the probability of**
  **actually entering them in the previous time interval.**
  **  **
```

```
          **For every state which can be reached at the current**
          **time, we must see if it causes the formula to pass or fail, in**
          **which cases we adjust the values for total_pass or**
          **total_fail and remove the state from the live set. If we cannot
            yet **
          **tell whether the formula is true or false, we must build the
            state/time matrix.**
12:       ∀q ∈ live(ct)
            **if q ⊨ a1, then formula is true**
13:         if q ⊨ a1 then
              **total_pass is incremented by the probability of entering q**
              **from the current state at the current time**
14:           total_pass := total_pass + prob(q, ct)
              **State q is removed from the live set**
15:           live(ct) := live(ct) \ {q}
            **Otherwise, if q ⊨ a0 (and q is not a terminating state)**
            **then the formula may still be true,**
            **so we must build matrix(q, ct) and keep state q in the live(ct)
              set.**
16:         elseif q ⊨ a0 ∧ q ∉ terminating_states then
              **The procedure new_state_matrix returns**
              **matrix(q, ct): the matrix for state q at current time, and
                requires**
              **prob(q, ct): the probability of entering state q from the current**
              **state at the current time.**
17:           new_state_matrix(matrix(q, ct), prob(q, ct))
18:         else **If q does not model a_0 or it is a terminating state and also**
                **it does not model a_1 then the formula is false**
                **total_fail is incremented by the probability of entering q**
                **from the current state at the current time**
19:           total_fail := total_fail + prob(q, ct)
              **State q is removed from the live set**
20:           live(ct) := live(ct) \ {q}
21:         end if
22:       end forall **for all states in live(ct)**
23:   until total_pass > p **formula has passed**
24:       or
25:       total_fail ≥ 1 − p **formula has failed**
26:       or
27:       (error ≥ 1 − p ∧ error ≥ p) **no possibility of a pass or a fail**
28:       or
29:       ct = t **time's up.**
30:   if (ct = t) then
        **All states undecided at the last iteration are now false, so**
        **total_fail is set to 1 − total_pass − error**
31:     total_fail := 1 − total_pass − error
32:   end if
```

```
   ****
   **Output result, based on the values of**
   **total_pass, total_fail and error**
33: if total_pass > p then
      **SA models formula**
34: output pass
35: elseif **total_fail ≥ 1 − p**
      **SA does not model formula**
36: output fail
37: else **errors are too large; cannot decide**
38: output undecided
39: end if

**This procedure builds the initial matrix.**
**We assume there are n clocks associated with this state,**
**and c_l^{s_0} is the l th clock.**
**We abbreviate ⌈upper_bound(c_l^{s_0})⌉.1/δ by N_l.**
procedure init_matrix(matrix(s_0, 0))
begin procedure
   ∀1 ≤ i_1 ≤ N_1
          ⋮
   ∀1 ≤ i_n ≤ N_n.matrix(s_0, 0)[i_1 ... i_n] := ∏_{l=1}^{n} pr(c_l^{s_0} ∈ [i_l − δ, i_l))
end procedure

procedure new_time_matrix(matrix(cs, ct), new_states(cs, ct), remain(cs, ct),
prob, error)
   **This procedure updates a matrix by incrementing time, not by**
   **changing state. We can do this by considering the values in the previous
     time**
   **matrix. It also updates the function prob,**
   **and the variable error.**
   **There are n clocks in state cs.**
begin procedure
1: ∀1 ≤ i_1 ≤ N_1
          ⋮
2: ∀1 ≤ i_n ≤ N_n.
                    **If one of the matrix indices is at its maximum value, then
                      the**
                    **probability value in this position must be zero. This is**
                    **because this procedure is always the first to update
                      a state/time matrix.**
                    **  **
                    **  **
3:                  if ∃l ≤ n • i_l = N_l then
4:                     matrix(cs, ct)[i_1, ..., i_n] := 0
```

```
                    **otherwise the values in the matrix can be updated
                      simply from the**
                    **values in the previous time matrix.**
5:                  else **all clocks c_i are ≥ 1 and < N_i**
6:                    matrix(cs, ct)[i_1, ..., i_n] :=
7:                          matrix(cs, ct)[i_1, ..., i_n] + matrix(cs, ct − δ)
                            [i_1+1, ..., i_n+1]
                    **we record the fact that it is possible to remain in this
                      state**
                    **at this time.**
8:                    remain(cs, ct) := true
9:                  end if
9a:end forall
   **We now pick out the positions in the previous time matrix which,**
   **when moved forward one unit in time, result in a new state.**
10:∀1 ≤ i_1 ≤ N_1
        ⋮
11:∀1 ≤ i_n ≤ N_n
   **If more than one of the previous time matrix indices is one, we know that**
   **more than one of the clocks will have reached zero by ct, and so we**
   **add the probability to error.**
11a:                if #{c_l | c_l = 1} > 1 then
12:                   error := error + matrix(cs, ct − δ)[i_1, ..., i_n]
12a:                else if #{c_l | c_l = 1} = 1
                      **Given the stochastic Automaton SA, the state cs and
                        the clock cc**
                      **s' is the resulting state. If the clock is associated with
                        more than**
                      **one transition the function pick (the adversary) chooses
                        the**
                      **resulting state. Otherwise the state is the one
                        determined by the**
                      **transition relation of the SA.**
13:                   s' := pick(SA, cs, c_l)
13a:                  new_states(cs, ct) := new_states(cs, ct) ⋃ {s'}
                      **the probability of entering s' at time ct**
                      **is incremented by the matrix probability**
14:                   prob(s', ct) := prob(s', ct) + matrix(cs, ct − δ)[i_1, ..., i_n]
22:                 end if **line 11**
23:               end forall
24:end procedure


**This procedure builds a new matrix, where the state is new rather than
  the time**
**We assume there are n clocks associated with this state,**
**and c_l^s is the l th clock.**
```

We abbreviate $\lceil upper\_bound(c_l^s)\rceil.\frac{1}{\delta}$ by N_l.
The values in the matrix are calculated by multiplying the clock
probabilities by a factor of p, where p is the probability of
entering the state, and adding this value to the value already in
the position.

procedure new_state_matrix$(matrix(cs, ct), p)$
begin procedure
$\forall 1 \leq i_1 \leq N_1$
$\vdots$
$\forall 1 \leq i_n \leq N_n.matrix(cs, ct)[i_1, \ldots, i_n] := matrix(cs, ct)[i_1, \ldots, i_n] +$
$(p \times \prod_{l=1}^{n} pr(c_l^s \in [i_l - \delta, i_l)))$
end procedure

ACKNOWLEDGMENTS

Thanks are due to coworkers on this project for their input into this work: Gordon and Lynne Blair from Lancaster University. Also to Pedro D'Argenio, Joost-Pieter Katoen and Holger Hermanns. In particular Pedro's observations on clock equivalences for stochastic automata have greatly influenced our approach. Finally, we would like to thank Fawzi Daoud for initiating the submission of this article to ACM TOCL.

REFERENCES

ALUR, R., COURCOUBETIS, C., AND DILL, D. 1990. Model-checking for real-time systems. In *Proceedings of 5th LICS*. IEEE Computer Society, New York, 414–425.

ALUR, R. AND DILL, D. 1994. A theory of timed automata. *Theoretical Computer Science 126*, 183–235.

BAIER, C., KATOEN, J.-P., AND HERMANNS, H. 1999. Approximate symbolic model checking of continuous-time markov chains. In *Proceedings of CONCUR'99*. Springer-Verlag, Berlin.

BAIER, C. AND KWIATKOWSKA, M. 1998. Model checking for a probabilistic branching time logic with fairness. *Distrib. Comput. 11*, 125–155.

BOWMAN, H., BLAIR, L., BLAIR, G. S., AND CHETWYND, A. 1998. *Formal Specification of Distributed Multimedia Systems*. University College London Press., London.

BOWMAN, H., BRYANS, J., AND DERRICK, J. 2000. A model checking algorithm for stochastic systems. Tech. Rep. 4-00, University of Kent at Canterbury.

D'ARGENIO, P. 1999. Algebras and automata for timed and stochastic systems. Ph.D. thesis, University of Twente, Enschede, The Netherlands.

D'ARGENIO, P. R., KATOEN, J.-P., AND BRINKSMA, E. 1998. An algebraic approach to the specification of stochastic systems (extended abstract). In *Proceedings of the Working Conference on Programming Concepts and Methods*, D. Gries and W.-P. de Roever, Eds. Chapman & Hall, London.

DAWS, C., OLIVERO, A., TRIPAKIS, S., AND YOVINE, S. 1995. The tool KRONOS. In *Proceedings of Workshop on Verification and Control of Hybrid Systems III*, R. Alur, T. A. Henzinger, and E. D. Sontag, Eds. Number 1066 in LNCS. Springer-Verlag, Berlin, 208–219.

EMERSON, E. A. 1990. *Handbook of Theoretical Computer Science*. Elsevier Science Publishers., Amsterdam, Chapter 16: Temporal and Modal Logic, 996–1072.

GLYNN, P. 1989. A GSMP formalism for discrete event simulation. *Proc. IEEE 77,* 1, 14–23.

HENZINGER, T. A., HO, P.-H., AND WONG-TOI, H. 1997. Hytech: A model checker for hybrid systems. *Software Tools for Technology Transfer 1*, 110–122.

HILLSTON, J. 1996. *A Compositional Approach to Performance Modelling*. Distinguished Dissertations in Computer Science. Cambridge University Press., Cambridge.

KLEINROCK. 1975. *Queueing Systems, Volume I: Theory*. John Wiley, New York.
LARSEN, K. G., PETTERSSON, P., AND YI, W. 1997. Uppaal in a nutshell. *Int. J. Softw. Tools Tech. Transfer 1*, 134–152.
MARSAM, M. A., CONTE, G., AND BALBO, G. 1984. A class of generalised stochastic petri nets for the performance evaluation of multiprocessor systems. *ACM Trans. Comput. Syst. 2*, 93–122.
SHIRYAYEV, A. N. 1984. *Probability*. Springer Series in Soviet Mathematics. Springer-Verlag, Berlin.
STEWART, W. J. 1994. *Introduction to the Numerical Solution of Markov Chains*. Princetown University Press, Princetown, New Jersey.
TANENBAUM, A. S. 1996. *Computer Networks*. Prentice-Hall, New Jersey.

Received February 2000; revised November 2001; accepted April 2002