Using Statistical Model Checking for Measuring Systems*

Radu Grosu[1], Doron Peled[2], C. R. Ramakrishnan[3], Scott A. Smolka[3], Scott D. Stoller[3], and Junxing Yang[3]

[1]Vienna University of Technology
[2]Department of Computer Science, Bar Ilan University
[3]Department of Computer Science, Stony Brook University

Abstract. State spaces represent the way a system evolves through its different possible executions. Automatic verification techniques are used to check whether the system satisfies certain properties, expressed using automata or logic-based formalisms. This provides a Boolean indication of the system's fitness. It is sometimes desirable to obtain other indications, measuring e.g., duration, energy or probability. Certain measurements are inherently harder than others. This can be explained by appealing to the difference in complexity of checking CTL and LTL properties. While the former can be done in time linear in the size of the property, the latter is PSPACE in the size of the property; hence practical algorithms take exponential time. While the CTL-type of properties measure specifications that are based on adjacency of states (up to a fixpoint calculation), LTL properties have the flavor of expecting some multiple complicated requirements from each execution sequence. In order to quickly measure LTL-style properties from a structure, we use a form of statistical model checking; we exploit the fact that LTL-style properties on a path behave like CTL-style properties on a structure. We then use CTL-based measuring on paths, and generalize the measurement results to the full structure using optimal Monte Carlo estimation techniques. To experimentally validate our framework, we present measurements for a flocking model of bird-like agents.

1 Introduction

Model checking aims to check that a model of a system satisfies a given specification. Recent results [1, 7, 9] show how to extend model checking into a more general method for measuring quantitative properties of a given system. Measurements can provide information about time, energy, the probability of an event occurrence, etc. In this paper, we explore the use of statistical model checking techniques for measuring quantitative properties of systems. We illustrate the power of these techniques for measuring the aggregate behavior of a flock of bird-like agents.

* The 2nd author is supported by ISF grant "Efficient Synthesis Method of Control for Concurrent Systems", award 126/12.

Techniques for the verification of quantitative properties have always relied on the ability to measure the quantities of interest: clock values in real-time systems, probabilities in stochastic systems, etc. There have been several efforts to apply similar techniques for measuring more general quantitative properties based on the operations needed to compute the measurements. For instance, [1] addresses the problems associated with measurements involving operations such as limit average, maximum and discounted sum; and [7] provides a logic for expressing a generalized class of quantitative properties.

In [10], we showed a neighborhood-based measurement scheme that is based on CTL-like specifications: creating a set of nodes in a graph corresponding to each state in the state space, and performing a measurement in a distributed manner propagating values from one node to its neighbors. When performing real-valued measurements, that scheme was limited to finite structures (weighted automata). The neighborhood-based measurement has the characteristic of CTL specification in the sense that the measurement is based on values that can be, temporarily or permanently, assigned to states. When the specification is sequence-based (or path-based), as in the logic LTL, or similar specification formalism that deal with quantitative real-time values that depend on the execution path, this technique does not work: the value of a state can depend on some complicated information related to the path through which the state was reached; states with different histories can have different values, and, to make things more complicated, this is affected from the rest of the execution yet to come. The difference is similar to the difference between CTL and LTL model checking, e.g., see [6] and [8]. While in CTL, we can put a temporary Boolean value per state while calculating subformulas, LTL model checking is done by providing some product of the state space with an automaton representing some essential summary of the sequence so far.

Since we are interested here in sequence based measuring, we make the following observation: on a single sequence, the neighborhood-based technique is the same as for a structure. We thus make our measurement sequence by sequence. We are limited in doing so by the fact that there may be infinitely many sequences and also the sequences themselves can be infinite. Hence, we measure finitely many sequences, using statistical model checking Monte Carlo technique. Moreover, we base our measurements on finite prefixes of executions.

While traditional model checking explores the state space exhaustively, statistical model-checking techniques sample from the state space, ensuring that sufficient samples are drawn, in order to verify a given property with a desired confidence level and error margin. Given a stochastic system S, a Boolean (temporal) property ϕ, and a real-valued parameter θ, statistical model checking determines whether $pr(S \models \phi) \geq \theta$ [14, 22]. This paper explores the use of Monte Carlo techniques for simultaneously measuring Boolean and quantitative properties, where the quantitative measures are dependent on the Boolean part.

Technical Approach and Contributions.

1. *Specification of measurement computations.* Following [10], we describe a specification formalism closely resembling a synchronous dataflow language

for measurement computation. At a high level, we associate a set of measurement variables with each state in the state space. We specify a mechanism for computing the values of variables at a state, based on the values of variables at the state's neighbors. The formalism is general enough to encode bounded model checkers (MCs) [13, 16] for Boolean temporal properties expressed in CTL or LTL. The advantage of the proposed measurement specification and related MC algorithm is in its simplicity and efficiency. It allows one to assemble the measurement specification from subproperties, just as CTL combines its temporal specification from its subformulas. The synchronous dataflow framework can be viewed as a generalization of *testers* proposed in [18]. Section 3 describes the specification formalism in greater detail.

2. *Model measurement using Monte Carlo techniques.* The computational mechanism described by the synchronous dataflow formalism associates a set of measures with each system execution (which may be trace or tree, depending on whether the property is linear or branching-time). The quantity of interest at a higher level may be an aggregate property covering the set of possible system executions (e.g., the average over possible runs). To this end, we develop a Monte Carlo technique (MCT) for generating a suitable set of samples (traces or trees) so that aggregate quantities can be computed to a desired confidence bound. The novelty of our MCT is that it jointly computes Boolean (satisfaction) and real (e.g., mean) values, using information from both strands to improve efficiency. Section 4 describes our MCT.

3. *An integrated model of flocking.* A number of different flocking models (FMs) have been developed to describe and explain the flocking behavior of birds [19, 21, 15, 4, 3]. The FMs generally consider each bird as an agent, where all agents are governed by the same control law. Certain input variables in the law executed by each agent are based on the attributes (e.g., velocity, position) of other agents in the flock. Such FMs are useful in understanding how emergent behaviors of the collection of agents arise from decisions made individually by each agent. We consider an integrated FM that uses a control law comprising a variety of terms from the existing literature. We consider quantitative objectives for the flock's behavior (e.g., velocity matching, the extent to which the velocities of agents are aligned), given in our measurement-specification formalism. The results of the measurement can be used to synthesize parameters (e.g., weights of different terms in the flocking control law) that optimize the objectives. Section 2 describes the FM in detail.

Section 5 reports preliminary results for measuring the velocity-matching objective of the flocking model. The results provide insight into the effectiveness of two approaches to controlling the accuracy of the Monte Carlo estimation.

2 Flocking

We illustrate our method with measurements of the behavior of a flock of agents. The flocking model is biologically inspired and may be useful in the design of

controllers for unmanned vehicles. In our flocking model, autonomous agents move in 2-dimensional space; the generalization to 3 dimensions is straightforward. Each agent's motion is determined by a locally executed control law. Every agent runs the same control law. Each agent has sensors that report the positions and velocities of all agents. The control law takes that information as input and returns an acceleration (i.e., change in velocity) for the agent.

Our broader goal is to develop methods for the design of control laws that cause the flock's behavior to satisfy given Boolean and quantitative objectives. An example of a Boolean objective is *collision avoidance*, i.e., that agents always maintain a specified minimum separation from each other. An example of a quantitative objective is *velocity matching* (VM), i.e., that agents gradually match velocities with each other, so the entire flock moves together. Note that this is a quantitative objective when the goal is to maximize VM, and a Boolean objective when the goal is to achieve VM above a specified threshold.

The control law typically contains several terms, each aimed at satisfying one or more objectives, and numerous parameters, including a weight for each term. Our broader goal is to develop methods that find values for the parameters that best achieve the specified Boolean and quantitative objectives. The two key components of the design method are an optimization framework and a measurement framework, used to measure how well the behavior of a flock with a given control law satisfies specified objectives.

There is some existing work on using genetic algorithms to adjust parameters in flocking control laws [2, 20]. They consider relatively limited and specific forms for the flocking control law and the objectives; in particular, the objective is expressed as a fitness function that is simply hand coded in a programming language. In contrast, our work aims to be more general and flexible, both in terms of considering a larger variety of terms in the flocking control law, including the terms in the flocking models in [19, 21, 15, 4, 3], and considering more varied and complex objectives, expressed more abstractly in a measurement framework.

Running Example. To illustrate the ideas in this paper, we consider a control law with a few selected terms. Let $x(t)$ and $v(t)$ be the vector of 2-dimensional positions and velocities, respectively, of all agents at time t. Let $x_i(t)$ and $v_i(t)$ be the position and velocity, respectively, of agent i at time t. Let k be the number of agents. The equation of motion and the control law are given in Figure 1. The acceleration is a weighted sum of the terms described next, with the *speed limit* function *spdLim* (formal definition omitted) applied to the sum to ensure that the magnitude of the velocity does not exceed $v_{\max} = 2$.

The *velocity-averaging* term *va*, adopted from Cucker and Dong's model [3], is designed to align the velocities of all agents, by gradually shifting them towards the flock's average velocity. The *strength function* ϕ specifies the strength of the velocity-matching influence between two agents as a function of the distance between them; H, β, ..., are parameters of the model.

The *collision avoidance* term *ca*, adopted from Cucker and Dong's model [3], is designed such that the separation between every pair of agents is always larger than d_{ca}^2. The *velocity matching* function *VM* (called *alignment measure* in [3]) measures the alignment the velocities of all agents (smaller values indicate better

$$\dot{x}_i(t) = v_i(t) \tag{1}$$

$$\dot{v}_i(t) = spdLim(w_{va} \cdot va_i(t) + w_{ca} \cdot ca_i(t) + w_{ctr} \cdot ctr_i(t) + w_{rpl} \cdot rpl_i(t)) \tag{2}$$

$$va_i(t) = \sum_{j=1}^{k} \phi(||x_i(t) - x_j(t)||)(v_j(t) - v_i(t)) \tag{3}$$

$$\phi(r) = \frac{H}{(1+r^2)^{\beta}} \tag{4}$$

$$ca_i(t) = VM(t) \sum_{j \neq i} f(||x_i(t) - x_j(t)||^2)(x_i(t) - x_j(t)) \tag{5}$$

$$VM(t) = (\frac{1}{k} \sum_{i>j} ||v_i(t) - v_j(t)||^2)^{\frac{1}{2}} \tag{6}$$

$$f(r) = (r - d_{ca})^{-2} \tag{7}$$

$$ctr_i(t) = \frac{||v_i(t)||}{||relCtr_i(t)||} \cdot relCtr_i(t) - v_i(t) \tag{8}$$

$$nbors_i(t,d) = \{j \mid j \neq i \wedge ||x_j(t) - x_i(t)|| \leq d\} \tag{9}$$

$$relCtr_i(t) = \left(\frac{1}{|nbors_i(t)|} \sum_{j \in nbors_i(t,d_{ctr})} x_j(t) \right) - x_i(t) \tag{10}$$

$$rpl_i(t) = \frac{||v_i||}{||offset_i(t)||} \cdot offset_i(t) - v_i \tag{11}$$

$$offset_i(t) = \frac{1}{|nbors_i(t,d_{rpl})|} \sum_{j \in nbors_i(t,d_{rpl})} (x_j - x_i) \tag{12}$$

Fig. 1. Flocking model

alignment). The *repelling function* f specifies the strength of the collision-avoidance influence between two agents as a function of the distance between them.

The *centering* term *ctr*, adopted from Reynolds' model [19], causes agents to form cohesive groups (sub-flocks), by shifting each agent's velocity to point towards the centroid (i.e., average) of the positions of its d_{ctr}-neighbors, where an agent's *d-neighbors* are the agents within distance d of the agent. The function $nbors_i(t,d)$ returns the set of indices of d-neighbors of agent i at time t. The function $relCtr_i(t)$ returns the relative position (i.e., relative to agent i's current position) of the centroid of its d_{ctr}-neighbors at time t. The definition of $ctr_i(t)$ applies a scaling factor to $relCtr_i(t)$ that yields a vector with the same magnitude as $v_i(t)$ and pointing in the same direction as $relCtr_i(t)$.

The *repulsion term*, adopted from Reynolds' model [19], causes agents to move away from their d_{rpl}-neighbors. The function $offset_i(t)$ returns the average of the offsets (i.e., differences in position) between agent i and its d_{rpl}-neighbors. The scaling factor applied to the offset in the definition of $rpl_i(t)$ is similar to the scaling factor used in the centering term.

The initial state is chosen stochastically. In our experiments, agents' initial positions are chosen uniformly at random in the box $[0..k, 0..k]$, and their initial velocities are chosen uniformly at random in $[0..1, 0..1]$. In more detailed models, stochastic environmental factors (e.g., wind) can be modeled with probabilistic transitions. In this case, the simulator would select from the corresponding probability distribution when the transition is taken.

Our experiments use the following parameter values: $k = 10, w_{va} = 0.6, w_{ca} = 0.1, w_{ctr} = 0.2, w_{rpl} = 0.1, H = 0.1, \beta = 1/3, d_{ca}^2 = 0.1, d_{ctr} = 5, d_{rpl} = 3$. We simulate the behavior of the flock using discrete-time simulations with a time step of 1 second and a duration of 50 seconds of simulation time. We chose this duration since we experimentally observed that the velocity-matching objective function stabilizes within 50 steps (a larger value had little effect).

Objective: Velocity Matching. We consider velocity matching as an objective for the running example. We use the velocity-matching function VM in equation (6) to measure how well the velocities are aligned in a state. Note that smaller values indicate better alignment. We consider two aspects of how well velocity matching is achieved: (1) how long it takes for VM to fall below a specified threshold θ and remain below θ for the rest of the execution; we measure this time as a fraction of the duration of the simulated execution, so the value is between 0 and 1; and (2) the average value of VM after it falls below the threshold θ and remains there; we measure this as a fraction of the maximum possible value of VM, so the value is also between 0 and 1. To combine these two quantities into a single fitness measure suitable for use in an optimization framework, we take a linear combination of them, with weights 0.01 and 0.99, respectively, so that these two quantities are of the same scale. In Section 3, we describe how to formally compute this linear combination using an LTL-style measurement.

3 Neighborhood-based and Sequence-based Measurements

In this section, we provide a formalism for expressing neighborhood-based measurements. Our formalism is state-based: each state is assumed to contain a tuple of constants and measurement variables. The variables are initiated, then, at each clock tick, the states, synchronously, exchange their values with their neighbor states, and apply an update function to obtain a new measurement. An expression over the state variables is assigned to each state, and its value is calculated at each tick. The value of the expression must decrease with each update, so that it bounds the amount of steps that can be performed by each state.

Definition 1. *A* state space $\mathcal{S}$ *is a triple* $\langle S, s_0, R \rangle$ *where*

- S *is a finite set of states.*
- $s_0 \in S$ *is the* initial state.
- $R \subseteq S \times S$ *is a relation over* S*, where if* $(s, s') \in R$*, then* s' *is the* successor *of* s*.*

Let $R^\bullet(s) = \{q|R(s,q)\}$ *be the* successors *of* s, ${}^\bullet R(s) = \{q|R(q,s)\}$ *its predecessors, and* $N(s) = {}^\bullet R(s) \cup R^\bullet(s)$. *Let* $n(s) = |N(s)|$ *be the number of neighbors of* S. *We assume that the size* z *of the state space, and the length of its maximal path (its* width*)* w *are known.*

Often, the state space is generated from some given system, where the states represent, e.g., the assignment of values to variables, and a successor state is generated from its predecessor by firing some atomic transition. The connection between a system and its state space is orthogonal to the focus of this paper.

3.1 Neighborhood-based measurement

In [10], we propose a measuring specification based on neighborhoods for a state space. We associate with each state the following components:

- A tuple of *measurement* variables V over some finite domains.
- Initial value for each measurement variable from its domain.
- A well founded set $\langle W, < \rangle$, where W is a set of values and $<$ is a partial order on W where no infinite decreasing chain exists.
- An expression E, based on the variables V that results in values from W. We denote the current value of E at state s as $E(s)$.
- Each state may have some constants assigned to it (this can be extended so that constants are assigned also to edges).
- An *update* function f for the variables V. It can be based on the current version T of the variables (denoted by $T(s)$), the constants on the state, and a version V^q for each neighbor q of s (also, constants on the edges to and from neighbors). For some purposes, it is sufficient to use updates based only on successors or on predecessors. The update function must satisfy that $E > E'$, where E' is the expression E after applying the update.

The measuring specification is, in itself, also an algorithm, which can be implemented directly. Basically, it consists of the following:

With each tick of the clock, execute per each state of the state s space:
  If $E(s)$ is not minimal, then do
    Send $T(s)$ to all neighbors.
    Receive $T^q(s)$ from the neighbors, $q \in \{q_1, q_2, \ldots q_{n(s)}\}$ of s.
    Let $V := f(V, V^{q_1}, \ldots, V^{q_{n(s)}})$ od

Note that with the recalculation of the values of V in the state, the expression E would decrease.

Example. An example of measuring is to find what is the maximal value assigned to any node reachable from the current node. We set up the following:

- A constant c per each state (denoted $c(s)$) representing the measured value.
- A variable m that contains the current calculated maximum, initialized to c.
- A counter d initialized to w, the width of the structure.

– The well founded domain is the natural numbers with the usual $<$. The decreasing expression E assigned to each state is simply d.
– The update function takes the values of successor states $R^\bullet$ and calculates the multiple assignment

$$(m, d) := (max(m, m^1, m^2, \ldots m^{n(s)}), d - 1)$$

We show now (see also [10]) an implementation of CTL model checking using our measuring principles. This algorithm resembles the original Clarke and Emerson algorithm [6], with the help of bounded model checking. This is just an example to show that the power of our measuring formalism exceeds that of CTL model checking. However, one may want to use the formalism without fixing a different temporal property as the basis for measurements.

Recall that the syntax of the CTL formulas is as follows:

$$\varphi := p|\neg\varphi|(\varphi \vee \varphi)|EX\varphi|(\varphi \wedge \varphi)|(\varphi EU \varphi)|(\varphi AU \varphi)$$

The semantics is as usual, see [6].

First, the variables for model checking a CTL property φ include for each subformula η a variable v_η. These variables have three possible values: **U**, **T** and **F**, where **U** (for *undefined*) being the initial value. There is also a *phase counter* variable *pc*, which is set initially to the number of subformulas in φ, and a *downcounter dc*. The downcounter is set to 1 at the beginning phases that are associated with Boolean subformulas or EX, and to w (or to z) when the phase is associated with EU or AU. This is because information as far as s state away from the current state may affect the value of the EU or AU formula.

Now, at each step, we decrement one from *dc*, and if it reaches 0, decrement one from *pc*, unless it is zero, and then we terminated. When we move to a new phase, we set *dc* to 1 or w, according to the type of the subformula. We handle the subformulas according to their size. In this way, when dealing with some subformula η, the measurement variables for its subformulas are already calculated and set to **T** or **F**. Depending on the type of the subformula, we perform an action. For a Boolean operators, we perform the same operator on the values of the corresponding measurement variables, e.g., if the subformula is $(\eta \vee \rho)$ then we set $v_{(\eta \vee \rho)}$ to $v_\eta \vee v_\rho$. Since we use three valued logic, we need to extend the Boolean operators. Accordingly, we have $(\mathbf{T} \vee \mathbf{U}) = \mathbf{T}$, $(\mathbf{F} \wedge \mathbf{U}) = \mathbf{F}$, and the symmetric situations. For the other cases involving at least one **U**, the result is **U**.

For the subformula $(\eta EU \rho)$, we obtain at each step the values of $v^{q_i}_{(\eta EU \rho)}$ from each successor $q \in N(s)$. The values v_ρ and v_η are calculated in a previous phase. Then we set up

$$v_{(\eta EU \rho)} := v_\rho \vee (v_\eta \wedge \bigvee_{q \in N(s)} v^q_{(\eta EU \rho)})$$

For $(\eta AU \rho)$, just replace in the formula above $\bigvee_{i \in 1..m}$ with $\bigwedge_{i \in 1..m}$. Now, if we finished the current phase (*dc* becomes 0) and $v_{(\eta EU \rho)}$ is still **U**, then we set it to **F**.

In [10], we showed how to use such a measuring specification for calculating whether a robot can be reached into some docking station before its battery is critically discharged. There, we used the backwards propagation of values to check whether the shortest paths from each state are still short enough we used forwards propagation to check whether critical discharge (including over cycles in the path of the robot) occur.

3.2 Path measurements

We discuss now measurements that depends on the history of the execution, as well as its future. While our measurement formalism does not depend on a logic, it is easy to explain the different characterization of such measurement using the different requirements between verifying the temporal logics CTL and LTL. The temporal logic LTL [17] has a different characterization than CTL, since it can claim multiple properties for each single execution path. In fact, there is an exponential number of ways, in the size of the LTL property, in which a state can evolve to satisfy the property, depending on its history. Model checking of LTL properties is facilitated by a product of the state space with an automaton that represents updating and memorizing the available possibilities. For this reason, the neighborhood based measurement we proposed in [10] would make very little sense for path based properties (such as LTL formulas).

There are two problems that we need to face in such measuring:

1. The paths that are measured may be infinite. Although it is known that if there is an execution that satisfies an LTL property then there is one that is ultimately periodic (see, e.g., [8]), i.e., consists of a finite prefix, and a finite part that repeats indefinitely. However, measuring non-Boolean properties may have different results where ultimately periodic sequences are not good representatives (see, e.g., [1] for measuring the limit of the average of values along a sequence).
2. There are multiple paths in the structure (possibly infinite), and we are interested to give a measurement of all the paths, or sufficiently many of them.

In order to tackle problem 1, we assume measurements that are affected mainly by a finite prefix of sequences. We may then decide to use some limit on the length of a sequence, and show, separately, how measurements are affected by changing this limit. In order to tackle problem 2, we use generalized Monte-Carlo measurements, and are satisfied when we can conclude that a large enough number of executions (defined as a parameter), has guaranteed some measurement threshold. This means that we have to provide the threshold (some value that the measurements either surpass or do not reach), and a the level of confidence required for this threshold.

3.3 Example: Velocity Matching Based Measurement

As an example, we show how to measure the objective of velocity matching in Section 2 using our measurement formalism. Although the measurement does not

necessarily depend on having a temporal property associated with the measure (e.g., calculating the average, the sum, etc.) we can, in this case look at the LTL property $\varphi_{VM} := FGp$, (eventually always p), where p is the proposition $VM \leq \theta$, and VM is the (normalized) velocity matching in a state.

Note that since in path measurement we have only a single successor and a single predecessor, we can distinguish them, e.g., denote the successor version of v by v' (and if we look at the predecessors, denote the predecessor of v by v'').

The variables we use are as follows:

- Boolean variables: B_p, B_{Gp} and B_{FGp}, all initialized to $\mathbf{U}$.
- Real variables:
  - vm: the velocity matching value, initialized to 0.
  - avg: the average velocity matching when Gp holds, initialized to 0.
  - $step$: the number of steps from current state to the first state where Gp holds, initialized to ∞.
  - lc: the linear combination of avg and $step$, initialized to 0.
- Down counter: l, which is initialized to w, the length of the paths we use (we use a constant length, which is 50)
- The values got from the successor are marked by a prime, i.e., ${B_{Gp}}'$, ${B_{FGp}}'$, avg', etc. For the tester corresponding to the last state of the path who has no successor, we assume the following: ${B_{Gp}}' = \mathbf{T}$, ${B_{FGp}}' = \mathbf{F}$, $avg' = 0$, $step' = \infty$.

At each step, we calculate the following:

- If $l > 0$:
  - $B_p = p$.
  - $B_{Gp} = B_p \wedge {B_{Gp}}'$.
  - $B_{FGp} = B_{Gp} \vee {B_{FGp}}'$.
  - $vm = VM(s)$, the velocity matching value in current state s. The calculation of $VM(s)$ is explained in Section 2.
  - $avg = \mathbf{if}\,(B_{Gp} = \mathbf{T})\,\mathbf{then}\,(avg' * (w - l) + vm)/(w - l + 1)\,\mathbf{else}\,avg'$.
  - $step = \mathbf{if}\,(B_{Gp} = \mathbf{T})\,\mathbf{then}\,1\,\mathbf{else}\,(step' + 1)$.
  - $lc = 0.99 * avg + 0.01 * (step/w)$.
  - $l := l - 1$.

The well founded ordering is the value of l. That is, we terminate when $l = 0$. At this point, we can take values (B_{FGp}, lc) from the initial node of the sequence.

4 Generalized Monte-Carlo Measurements

The formalism discussed in Section 3 takes as input a bounded sequence $s_{1:n}$ and returns a measure of it. For example, for the flocking model (FM) we presented in Section 2, we return a pair (b, r) of a Boolean and real value, respectively, where b is the value of $FG(VM \leq \theta)$, and r is the weighted sum of two quantities: how long it takes (as a fraction of n) for VM to fall below threshold θ and remain there; and, when b is true, the average of VM for the (maximal) subsequence where $G(VM \leq \theta)$ holds.

A particular sequence $s_{1:n}$ is generated by running a k-agent FM for a given initial state, which is assumed to be distributed in the box $[0..k, 0..k]$, for agent positions, and in the box $[0..1, 0..1]$, for agent velocities. The FM assumes that these distributions are uniform, but, in general, they can be any arbitrary distribution, including Gaussian.

Since each pair (b, r) is the result of an initialized execution of the FM, both b and r are the values of *independent*, *identically distributed* (IID) random variables $Z = (B, R)$. Assuming that r belongs to the interval $[0, 1]$ is not a limitation, as one can always normalize the values of R, provided that one knows its range. We do exactly this in Sections **??**-3, where the fitness value of the flock we compute is the weighted sum of two $[0, 1]$-normalized quantities.

A generalized measurement aims to efficiently obtain a joint estimate $\mu_Z = (\mu_B, \mu_R)$ of the mean values $\mathbf{E}[B]$, $\mathbf{E}[R]$ of B and R, respectively. Since an exact computation of μ_Z is almost always intractable (e.g. NP-hard), a Monte Carlo (MC) approach is used to compute an (ϵ, δ)-approximation of this quantity.

The main idea of MC is to use N random variables (RVs) $Z_1, \ldots, Z_N$, also called *samples*, IID distributed according to Z and with mean μ_Z, and to take the sum $\widetilde{\mu}_Z = (Z_1 + \ldots + Z_N)/N$ as the value approximating the mean μ_Z.

While MC techniques were used before to measure the satisfaction probability of temporal logic formulas [11, 9, 14], or to compute the mean of an RV [5, 12], the main novelty of this paper is to *jointly* measure the Boolean satisfaction and the mean real value. The Boolean-value computation is informing the real-value computation and *vice versa*, thereby increasing the efficiency of our approach.

Additive approximation. An important issue in an MC approximation scheme is to determine an appropriate value for N. If μ_Z is expected to be large, then one can exploit the Bernstein inequality and fix N to be $\Upsilon \propto ln(1/\delta)/\epsilon^2$. This results in an *additive* or *absolute-error* (ϵ, δ)*-approximation scheme*:

$$\mathbf{Pr}[\mu_Z - \epsilon \leq \widetilde{\mu}_Z \leq \mu_Z + \epsilon] \geq 1 - \delta$$

where $\widetilde{\mu}_Z$ approximates μ_Z with absolute error ϵ and with probability $1 - \delta$.

If Z is assumed to be a Bernoulli RV, then one can use the Chernoff-Hoeffding instantiation of the Bernstein inequality, and further fix the proportionality constant to $\Upsilon = 4\, ln(2/\delta)/\epsilon^2$, as in [11]. This bound can also be used for the joint estimation of RV $Z = (B, R)$, as B is a Bernoulli RV, and the proportionality constraint of the Bernstein inequality is also satisfied for RV R. This results in the *additive approximation algorithm* (AAA) below.

AAA algorithm
input: (ϵ, δ) `with` $0 < \epsilon < 1$ `and` $\delta > 0$.
input: `Random vars` Z_i `with` $i > 0$, `IID`.
output: $\widetilde{\mu}_Z$ `approximation of` μ_Z.

(1) Υ `= 4 ln(2 / δ) /` ϵ^2`;`
(2) **for** `(N = 0; N ≤` Υ`; N++) S = S +` Z_N`;`
(3) $\widetilde{\mu}_Z$ `= S/N;` **return** $\widetilde{\mu}_Z$`;`

It is important to note that in AAA, the number of samples N depends only on ϵ and δ, and is totally oblivious to the mean value μ_Z to be computed.

Multiplicative approximation. In case the mean value μ_Z is very low, the additive approximation $\widetilde{\mu}_Z$ may be meaningless, as the absolute error may be considerably larger than the actual value μ_Z. In such cases, a *multiplicative* or *relative-error* (ϵ, δ)-*approximation scheme* is more appropriate:

$$\mathbf{Pr}[\mu_Z - \mu_Z\epsilon \leq \widetilde{\mu}_Z \leq \mu_Z + \mu_Z\epsilon] \geq 1 - \delta$$

where $\widetilde{\mu}_Z$ approximates μ_Z with relative error $\mu_Z\epsilon$ and with probability $1 - \delta$.

In contrast to the Chernoff-Hoeffding bound $\Upsilon = 4\ln(2/\delta)/\epsilon^2$, required to guarantee an absolute error ϵ with probability $1 - \delta$, the *zero-one estimator theorem* in [12] requires a bound proportional to $N = 4\ln(2/\delta)/\mu_Z\epsilon^2$ to guarantee a relative error $\mu_Z\epsilon$ with probability $1 - \delta$.

When applying the zero-one estimator theorem, one encounters, however, two main difficulties. The first is that N depends on $1/\mu_Z$, the inverse of the value that one intends to approximate. This problem can be circumvented by finding an upper bound κ of $1/\mu_Z$ and using κ to compute N. Finding a tight upper bound is, however, in most cases very difficult, and a poor choice of κ leads to a prohibitively large value for N.

An ingenious way of computing N without relying on μ_Z or κ is provided by the *Stopping Rule Algorithm* (SRA) of [5]. When $\mathbf{E}[Z] = \mu_Z > 0$ and $\Sigma_{i=1} Z_i \geq \Upsilon$, the expectation $\mathbf{E}[N]$ of N equals Υ.

SRA algorithm
input: (ϵ, δ) `with` $0 < \epsilon < 1$ `and` $\delta > 0$.
input: `Random vars` Z_i `with` $i > 0$, `IID`.
output: $\widetilde{\mu}_Z$ `approximation of` μ_Z.

(1) $\Upsilon = 4\,(e-2)\,\ln(2/\delta)/\epsilon^2$; $\Upsilon_1 = 1 + (1+\epsilon)\,\Upsilon$;
(2) **for** (N=0, S=0; S$\leq \Upsilon_1$; N++) S = S + Z_N;
(3) $\widetilde{\mu}_Z$ = S/N; **return** $\widetilde{\mu}_Z$;

The second difficulty in applying the zero-one estimator theorem is the factor $1/\mu_Z\epsilon^2$ in the expression for N, which can render the value of N unnecessarily large. A more practical approach is offered by the *generalized zero-one estimator theorem* of [5] which states that N is proportional to $\Upsilon' = 4\rho_Z \ln(2/\delta)/(\mu_Z\epsilon)^2$, where $\rho_Z = \max\{\sigma_Z^2, \epsilon\mu_Z\}$ and σ_Z^2 is the variance of Z. Thus, if σ_Z^2, which equals $\mu_Z(1 - \mu_Z)$ for Z a Bernoulli random variable, is greater than $\epsilon\mu_Z$, then $\sigma_Z^2 \approx \mu_Z$, $\rho_Z \approx \mu_Z$ and therefore $\Upsilon' \approx \Upsilon$. If, however, σ_Z^2 is smaller than $\epsilon\mu_Z$, then $\rho_Z = \epsilon\mu_Z$ and Υ' is smaller than the Υ by a factor of $1/\epsilon$.

To obtain an appropriate bound in either case, [5, 9] have proposed the *optimal approximation algorithm* (OAA) shown above. This algorithm makes use of the outcomes of previous experiments to compute N, a technique also known as *sequential MC*. The OAA algorithm consists of three steps. The first step calls the SRA algorithm with parameters $(\sqrt{\epsilon}, \delta/3)$ to obtain an estimate $\widehat{\mu}_Z$ of μ_Z. The choice of parameters is based on the assumption that $\rho_Z = \epsilon\mu_Z$, and ensures

OAA algorithm
input: Error margin ϵ and confidence ratio δ with $0 < \epsilon \leq 1$ and $0 < \delta \leq 1$.
input: Random vars Z_i, Z'_i with $i > 0$, IID.
output: $\widetilde{\mu}_Z$ approximation of μ_Z.

(1) $\Upsilon = 4\,(e-2)\ln(2/\delta)/\epsilon^2$; $\Upsilon_2 = 2\,(1+\sqrt{\epsilon})\,(1+2\sqrt{\epsilon})\,(1+\ln(3/2)/\ln(2/\delta))\,\Upsilon$;

(2) $\widehat{\mu}_Z$ = SRA(min{1/2,$\sqrt{\epsilon}$}, δ/3,Z);

(3) $N = \Upsilon_2\,\epsilon/\widehat{\mu}_Z$; $S = 0$;
(4) **for** (i=1; i$\leq$N; i++) $S = S + (Z'_{2i-1} - Z'_{2i})^2/2$;
(5) $\widehat{\rho}_Z = \max\{S/N, \epsilon\,\widehat{\mu}_Z\}$;

(6) $N = \Upsilon_2\,\widehat{\rho}_Z/\widehat{\mu}_Z^2$; $S = 0$;
(7) **for** (i=1; i$\leq$N; i++) $S = S + Z_i$;
(8) $\widetilde{\mu}_Z = S/N$; **return** $\widetilde{\mu}_Z$;

that SRA takes $3/\epsilon$ less samples than would otherwise be the case. The second step uses $\widehat{\mu}_Z$ to obtain an estimate of $\widehat{\rho}_Z$. The third step uses $\widehat{\rho}_Z$ to obtain the desired value $\widetilde{\mu}_Z$. Should the assumption $\rho_Z = \epsilon\mu_Z$ fail to hold, the second and third steps will compensate by taking an appropriate number of additional samples. As shown in [5], OAA runs in an expected number of experiments that is within a constant factor of the minimum expected number.

For simplicity, both in SRA and in OAA, we only showed the joint variable $Z = (B, R)$, and used a generic RV $\widehat{\rho}_Z$. In our implementation, however, we distinguish between the mean and the variances of B and R, and if we observe that the variance of R is very low, we stop, even if the variance of B is larger, as R is determining the value of B.

5 Experimental Results

We have implemented our flocking model presented in Section 2 in MATLAB. Recall that the property we measure is φ_{VM} defined in Section 3.

In order to get an *approximate measure for the flocking model*, within a required error margin ϵ and confidence level δ, we applied the generalized Monte-Carlo estimation (GMCE) algorithms discussed in Section 4. The GMCE algorithms use both the the boolean values b and the real values r obtained from the *path measurements* (b, r), in order to determine the stopping time, and to compute the mean values μ_B and μ_R of interest.

Every path measurement (b, r) is obtained by running the measurement algorithm defined in Section 3.3, over a *random execution of the flocking model*. This execution is generated by our *discrete-time simulator*, by first choosing the initial state uniformly at random and then integrating the differential equations given in Section 2, Figure 1

Table 1 shows the results of 5 runs of OAA with $\theta = 0.05$, $\epsilon = 0.3$ and $\delta = 0.3$. In the table, μ_R is the real part of the output of OAA; it estimates the mean of the quantitative measurement we defined in Section 3.3 for φ_{VM}. N is the number of samples used to compute μ_R.

We omit the boolean part μ_B because it is *always one* for this example. We also show the average (Avg) and the standard deviation (Std) of the results. We

compare our results with the AAA algorithm where a fixed number of samples N is used, $N = 4log(2/\delta)/\epsilon^2$, as shown in Table 2. The OAA algorithm requires significantly more samples, but considerably reduces the standard deviation.

Table 1. Results obtained from OAA

Runs	μ_R	N
1	0.00644	66846
2	0.00651	65592
3	0.00646	66696
4	0.00648	66173
5	0.00646	66588
Avg	0.00647	66379
Std	2.659e-05	505.9

Table 2. Results obtained from AAA

Runs	μ_R	N
1	0.00681	84
2	0.00685	84
3	0.00589	84
4	0.00630	84
5	0.00585	84
Avg	0.00634	84
Std	0.00047	0

Note that, in Table 2, we use the same error margin and confidence level as in Table 1. This additive approximation is, however, meaningless, because the additive error is much greater than μ_R itself. If we want to achieve the same accuracy as OAA, the error margin for AAA should be set to $\epsilon' = \mu_R * \epsilon \approx 0.00194$. This results in the sample size $N = 4log(2/\delta)/\epsilon'^2 = 2,016,282$, which is significantly larger than the sample size used in OAA.

Table 3 shows the results when we choose different values of ϵ and δ for the OAA algorithm. It is clear that choosing smaller values of ϵ and δ results in smaller standard deviations at the expense of a larger sample size. We obtain values of μ_R that are, however, highly consistent with one another from different values of ϵ and δ.

Table 3. OAA with different ϵ and δ

	$\epsilon = 0.1$, $\delta = 0.3$	$\epsilon = 0.1$, $\delta = 0.5$	$\epsilon = 0.3$, $\delta = 0.3$	$\epsilon = 0.3$, $\delta = 0.5$
Avg μ_R	0.00647	0.00648	0.00646	0.00648
Std μ_R	1.816e-05	2.370e-05	2.659e-05	3.267e-05
Avg N	131861	102412	66379	51410
Std N	326.2	447.7	505.9	216.8

6 Conclusions

Algorithmic methods for checking the consistency between a system and its specification [6] have been generalized into measuring properties of systems [1]. The study of such measurement techniques provided interesting algorithmic, complexity and computability results.

In a previous work [10] we proposed a simple formalism for fast measurement, based on repeatedly observing the neighborhood of the states of the system.

Information about partial measurements flow through the states to and from its neighbor states. Instead of providing a global logic base specification, which assert about the different paths of the state space, and their interconnection, our formalism is based on the view of a state and the information that flows through it from its predecessors and successors. An expression over some well founded set is used to control the termination of the accumulative data flow based measuring. This setting is quite general, and allows measurements that combine different arguments, both Boolean and numeric. The formalism is simple, and the measurement is efficient. If the measurement is provided in terms of some logic formalism, then it needs to be translated into our formalism.

Our measurement formalism is not appropriate for any formalism. An important class of such formalisms are those that are dependent on some memory accumulated on the execution path. In such specification, different paths that lead to the same state may result in differemt measurements. Even if the amount of memory needed to keep track of the history of the path is finite, the measurement is not unique per given state (as it depends on the history of the path). Performing the measurement path by path is often not feasible as there can be infinitly many or prohibitly many paths. We propose here a tradeoff between accuracy and exhastiveness, based on statistical model checking [9, 11, 12, 20]. We neighborhood measurement techniques to a statistically big enough sample of paths, and use statistical inference to conclude the measurement of the system.

Such complex measurements can appear naturally in systems that combine physical parameters and in biological systems. As a running example we used models for birds flocking [19, 21, 15, 4, 3, 2, 20]. Based on several researched models for flocking, we want to measure the well behavior of their combination. In particular, the eventual well matching of speads among birds. We cast the eventual spead matching measurement in terms of our formalism. Then we make experiments based on statistical model checking implemented using MATLAB.

References

1. Chatterjee, K., Doyen, L., Henzinger, T.A.: Expressiveness and closure properties for quantitative languages. Logical Methods in Computer Science 6(3) (2010)
2. Conley, J.F.: Evolving boids: Using a genetic algorithm to develop boid behaviors. In: Proceedings of the 8th International Conference on GeoComputation (GeoComputation 2005) (2005), http://www.geocomputation.org/2005/
3. Cucker, F., Dong, J.G.: A general collision-avoiding flocking framework. IEEE Trans. on Automatic Control 56(5), 1124–1129 (2011)
4. Cucker, F., Smale, S.: Emergent behavior in flocks. IEEE Trans. on Automatic Control 52(5), 852–862 (2007)
5. Dagum, P., Karp, R., Luby, M., Ross, S.: An optimal algorithm for Monte Carlo estimation. SIAM Journal on Computing 29(5), 1484–1496 (2000)
6. Emerson, E.A., Clarke, E.M.: Characterizing correctness properties of parallel programs using fixpoints. In: ICALP. pp. 169–181 (1980)
7. Finkbeiner, B., Sankaranarayanan, S., Sipma, H.: Collecting statistics over runtime executions. Formal Methods in System Design 27(3), 253–274 (2005)
8. Gerth, R., Peled, D., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: PSTV. pp. 3–18 (1995)

9. Grosu, R., Smolka, S.: Quantitative model checking. In: Proc. of the 1st International Symposium on Leveraging Applications of Formal Methods (ISOLA'04). pp. 165–174. Paphos, Cyprus (Nov 2004)
10. Grosu, R., Peled, D., Ramakrishnan, C., Smolka, S.A., Stoller, S.D., Yang, J.: Compositional branching-time measurements. In: From Programs to Systems—The Systems Perspective in Computing, Proceedings of ETAPS Workshop in honor of Joseph Sifakis. Lecture Notes in Computer Science, vol. 8415. Springer (2014)
11. Hérault, T., Lassaigne, R., Magniette, F., Peyronnet, S.: Approximate probabilistic model checking. In: Proc. Fifth International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2004) (2004)
12. Karp, R., Luby, M., Madras, N.: Monte-Carlo approximation algorithms for enumeration problems. Journal of Algorithms 10, 429–448 (1989)
13. Latvala, T., Biere, A., Heljanko, K., Junttila, T.: Simple bounded ltl model checking. In: IN: FMCAD. VOLUME 3312 OF LNCS. pp. 186–200. Springer (2004)
14. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: An overview. In: RV. pp. 122–135 (2010)
15. Olfati-Saber, R.: Flocking for multi-agent dynamic systems: Algorithms and theory. IEEE Trans. on Automatic Control 51(3), 401–420 (2006)
16. Penczek, W., Wozna, B., Zbrzezny, A.: Bounded model checking for the universal fragment of ctl. Fundam. Inf. 51(1), 135–156 (Mar 2002)
17. Pnueli, A.: The temporal logic of programs. In: FOCS. pp. 46–57 (1977)
18. Pnueli, A., Zaks, A.: Psl model checking and run-time verification via testers. In: The 14th International Symposium on Formal Methods. LNCS, vol. 4085, pp. 573–586. Springer Berlin / Heidelberg (Aug 2006)
19. Reynolds, C.W.: Flocks, herds and schools: A distributed behavioral model. In: Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1987). pp. 25–34. ACM (1987)
20. Stonedahl, F., Wilensky, U.: Finding forms of flocking: Evolutionary search in abm parameter-spaces. In: Multi-Agent-Based Simulation XI, Lecture Notes in Computer Science, vol. 6532, pp. 61–75. Springer (2011)
21. Vicsek, T., Czirók, A., Ben-Jacob, E., Cohen, I., Shochet, O.: Novel type of phase transition in a system of self-driven particles. Physical Review Letters 75, 1226–1229 (Aug 1995)
22. Younes, H.K.L.: Verification and Planning for Stochastic Processes. Ph.D. thesis, Carnegie Mellon (2005)