MC^{2}MABS: A Monte Carlo Model Checker for Multiagent-based Simulations

Benjamin Herd, Simon Miles, Peter McBurney, and Michael Luck

Department of Informatics, King's College London, United Kingdom

Abstract. Agent-based simulation has shown great success for the study of complex adaptive systems and could in many areas show advantages over traditional analytical methods. Due to their internal complexity, however, agent-based simulations are notoriously difficult to verify and validate.
This paper presents MC^{2}MABS, a Monte Carlo Model Checker for Multiagent-Based Simulations. It incorporates the idea of *statistical runtime verification*, a combination of statistical model checking and runtime verification, and is tailored to the approximate verification of complex agent-based simulations. We provide a description of the underlying theory together with design decisions, an architectural overview, and implementation details. The performance of MC^{2}MABS in terms of both runtime consumption and memory allocation is evaluated against a set of example properties.

Keywords: agent-based simulation, verification, formal methods, testing

1 Introduction

Agent-based simulation (ABS) is rapidly emerging as a popular paradigm for the simulation of complex systems that exhibit a significant amount of non-linear and emergent behaviour [28]. It uses populations of interacting, heterogeneous and often adaptive agents to model and simulate various phenomena that arise from the dynamics of the underlying complex systems. Although social science has been its traditional domain, ABS is also increasingly being used for the analysis of complex (socio-)technical, often also safety-critical systems in areas such as avionics [5], the design and analysis of robot and UAV swarms [30], and increasingly also the Internet of Things [17].

In that context, and similar to other software systems, *correctness* plays a central role and questions of quality assurance become increasingly important. It is common to distinguish between *verification* and *validation*. Whereas the former is targeted towards a system's correctness with respect to its specification (i.e. its correct implementation), the latter ensures a sufficient level of accuracy with respect to the intended application domain. In the context of modelling and simulation, the distinction between verification and validation becomes more complicated; it can thus be more useful to talk about *internal* and *external validation* instead [34, 4].

With their ability to produce complex, emergent behaviour from the action and interaction of its components, ABSs are notoriously difficult to understand and to engineer. In the multiagent systems community, formal and semi-formal verification approaches (both qualitative and quantitative) have shown great power. Methods and tools for the verification of ABS in particular, however, are still largely missing. With potentially large populations, different observational levels, heterogeneity, a strong focus on emergent behaviours, and a significant amount of randomness, ABSs possess a range of characteristics which, in their combination, represent a great challenge for verification.

In this paper, we present our research efforts on the ABS verification problem and introduce MC^{2}MABS— the *Monte-Carlo Model Checker for Multiagent-Based Simulations* — a prototypical statistical runtime verification approach and framework for complex agent-based simulation models. We start with a motivational example from the area of swarm robotics in Section 2. An overview of related work and some theoretical background on statistical model checking and runtime verification is given in Sections 3 and 4. The framework itself is described in Section 5, followed by a brief performance evaluation w.r.t. both runtime and memory consumption in Section 5.3. The paper concludes with a summary and ideas for future work.

2 Motivational example: collective behaviour in swarm robotics

In order to motivate the usefulness of verification for the analysis of agent-based simulation models, we introduce a small scenario from the area of swarm robotics. Although purely formal approaches for the analysis of swarm robotic models have shown to be useful [19, 23], they are not always applicable. For example, in order to be analytically tractable, purely formal approaches typically pose strong homogeneity assumptions upon the individual agents. This is appropriate as long as details of the environment, particular interactions, and individual differences (e.g. w.r.t. faulty behaviour) are irrelevant for the analysis. However, many emergent phenomena only become apparent if interaction, heterogeneity, and locality are taken into account. In this case, formal analysis may become intractable and the only way to investigate the dynamics of the scenario is simulation.

We focus here on *swarm foraging*, a problem which has been widely discussed in the literature on cooperative robotics [7]. Foraging describes the process of a group of robots searching for food items, each of which delivers energy. Individual robots strive to minimise their energy consumption whilst searching in order to maximise the overall energy intake. The study of foraging is important because it represents a general metaphor to describe a broad range of (often critical) collaborative tasks such as *waste retrieval*, *harvesting* or *search-and-rescue*. A good overview of multirobot foraging has been given by Cao *et al.* [7]. In a foraging scenario, robots move through the space and search for food items. Once an item has been detected within the robot's field of vision, it is brought back to the nest and deposited which delivers a certain amount of energy to the robot. Each action that the robot performs also consumes a certain amount of energy. The overall swarm energy is the sum of the individual energy levels.

A designer's main challenge is to tune the parameters of the individual agents such that the swarm as a whole is able to *self-organise* efficiently, i.e. to adapt to environmental circumstances such as food density. Since there is no central control, adaptation has to *emerge* from the agents' local actions and interactions. Due to the irreducibility of emergent phenomena, designing a distributed algorithm with a particular emergent behaviour in mind can be highly non-trivial. Interesting mechanisms have been presented by Liu *et al.* [24]. Here, agents adapt their behaviour according to three *cues*: (i) internal cues (personal success in retrieving food), (ii) environmental cues (collisions with teammates while searching for food), and (iii) social cues (teammate success in food retrieval). Depending on those influences, agents increase or decrease their searching and resting times with the goal of achieving an optimal collective division of labour.

ABS represents a powerful approach to study the dynamics of a distributed swarm algorithm. However, tuning the individual parameters such that the overall emergent behaviour is optimal can be hard. Verification — both qualitative and quantitative — can be of great help during the design process. As a starting point, one could, for example, formulate and verify the following qualitative safety property upon the simulation model: "the swarm must never run out of energy" (formally: $\neg\ \mathbf{F}(\mathit{energy} \leq 0)$). Although useful, such a pure macro-level criterion is rarely sufficient since, despite the whole swarm always having enough energy, individual agents may still run out. In order to solve this problem, a more fine-grained, quantified criterion may be formulated. Rather than stating that the swarm as a whole must never run out of energy, one may, for example, stipulate that "no individual agent must ever run out of energy" (formally: $\forall\, a \bullet \neg\ \mathbf{F}(\mathit{energy}_a \leq 0)$). Checking this criterion would catch those cases in which individual agents run out of energy, but one would still not know (i) *how many* of them do, and (ii) *why* this is the case.

In addition to conventional qualitative safety and liveness checking which provides clear yes/no answers but little explanatory insights, *quantitative analysis* may help to shed further light on the dynamics of the system. In addition to the two safety criteria above, it may, for example, be useful to answer the following questions.

- What is the *avg./min./max. probability* of an agent running out of energy?
- What *fraction of time* does an agent spend homing/resting/etc.?
- What *fraction of time* does an agent spend transitioning, e.g. from depositing to resting? (= overall probability of recharging)
- *How likely* is an agent to transition
  - from searching to grabbing? (= probability of finding food)
  - from grabbing to depositing? (= probability of losing out on food)
- What is the *correlation* between an agent's type and its probability of doing sth.?

Summarising the example above, we believe that a verification approach for ABS needs to satisfy the following requirements.

Efficiency: the approach should allow for the verification of complex simulation models in a timely manner. Due to the highly iterative nature of the modelling process, a user should be able to choose between a high level of accuracy at the expense of verification time and a lower level of accuracy but quicker results.

Expressivity: properties need to be formulable in a *formal, unambiguous way* and verifiable on *different levels of observation*: individual agents, groups of agents, as well as the whole population. Furthermore, the approach should allow for the verification of *both qualitative and quantitative properties*.

Flexibility: due to the sensitivity of complex systems to local differences and environmental conditions, a verification approach should not impose any unrealistically strict limitations on the simulation models that are verifiable, e.g. by assuming that agents are entirely homogeneous, by abstracting away the environment, etc.

Immediacy: in order to ensure the relevance of the verification results, the gap between the model to be verified and the actual simulation model should be as small as possible. Ideally, verification is performed *upon the original simulation model*.

As described in the following section, existing approaches do not currently satisfy those requirements in their combination. With the framework presented in this work, we aim to close this gap.

3 Related work

Model checking multiagent systems: Since its beginnings around 30 years ago, model checking has gained huge significance in computer science and software engineering in particular and has been successfully applied to many real-world problems. Model checking has also gained increasing importance in the multiagent community and numerous approaches have been presented in literature [9]. In alignment with the classical problems studied in the community, multiagent verification typically focusses on qualitative properties involving notions such as time, knowledge, strategic abilities, permissions, obligations, etc. In order to allow for the verification of larger agent populations, model checking algorithms for temporal-epistemic properties have also been combined successfully with ideas such as bounded model checking [26], partial order reduction [25] and parallelisation [20]. Despite impressive advances, however, verification still remains limited to either relatively small populations or scenarios with strong homogeneity assumptions [33].

In recent years, probabilistic approaches to model checking have also gained increasing importance in the multiagent community. Examples include the verification of systems with uncertainty w.r.t. communication channels and actions [11], qualitative and quantitative analysis of agent populations with uncertain knowledge [37], verification of probabilistic swarm models [18], or automated game analysis [3]. Similar to their non-probabilistic counterparts, these approaches also suffer from the state space explosion and are thus either limited to relatively small systems or dependent upon strong homogeneity or symmetry assumptions which increase their scalability but also limit their applicability to the verification of complex simulation models.

Verification of agent-based simulations: In the simulation community, most work on quality assurance focusses on validation, in particular statistical analysis of simulation output [35, 29]. Albeit related, those approaches possess a different flavour than the one described in this work since they focus on the *external validity* of the model, i.e. the link between the model world and the real world[1]. Verification (or *internal validation*), on the other hand, focusses on the link between the model and the theory. If mentioned at all in the simulation literature, verification is mostly equated with conventional code verification, e.g. through testing, reviews, or static analysis.

In the ABS community, a number of testing and monitoring approaches have been presented [6, 39, 32]. Most of these approaches are based on top of existing modelling frameworks such as Repast or Mason; consequently, correctness properties (in the form

[1] McKelvey refers to this link as the model's *ontological adequacy* [31].

of test cases) are formulated in the target language, typically in Java. An approach that combines ABS with numeric analysis has been presented by Wolf *et al.* [10]. The main motivation of the work is to determine if an individual-based system exhibits certain macroscopic emergent behaviour. To that end, repeated simulation is paired with numeric analysis from the system dynamics domain in order to detect deviations or to approximate the steady-state behaviour of the simulation. An advantage of the approach is its ability to speed up simulation by *steering* it into the direction required by the analysis algorithm. Due to its global focus, the approach is restricted to macro-level analysis; nevertheless, the power lies in the fact that it allows for the analysis of properties which conventional testing is not able to deal with.

Semi-formal and formal verification approaches similar to those for MAS described above but particularly tailored to ABS are still largely missing. The work closest related to ours is that of Sebastio and Vandin [36]. They present MultiVeStA, a statistical analysis tool which pairs discrete event simulation with statistical model checking. Similar to MC^{2}MABS, MultiVeStA can be coupled with existing simulators and allows for the verification of properties about expected values of observations. Properties are propositional in nature, i.e. more complex calculations or aggregations have to be 'wrapped' into propositions. Since MultiVeStA is not tailored to ABS, no internal structure is imposed on the simulation traces. As a consequence, there is no direct notion of observational levels in the property specification language.

It is important to note that, although useful in its own right, (semi-)formal verification will be even more powerful if it is embedded in a proper experimental environment. It has been argued that, if the systematic design of experiments was fully realised, the transparency of a simulation model could be increased significantly [27]. We believe that, rather than serving as an alternative, (semi-)formal verification may well become an integral part of this process.

4 Background

Statistical model checking: Conventional model checking aims to find an accurate solution to a given property by exhaustively searching the underlying state space which is, in general, only possible if the space is of manageable size [2]. One solution that works for probabilistic systems is to use a *sampling approach* and employ statistical techniques in order to generalise the results to the overall state space. In this case, n paths or *traces* are sampled from the underlying state space and the property is checked on each of them; statistical inference, e.g. *hypothesis testing*, can then be used to determine the significance of the results. Approaches of this kind are summarised under the umbrella of *statistical model checking*; a good overview is given by Legay *et al.* [21]. Due to its independence of the underlying state space, statistical model checking allows for the verification of large-scale systems in a timely, yet approximate manner.

One particular approach, *Approximate Probabilistic Model Checking*, provides a *probabilistic guarantee* on the accuracy of the approximate value generated by using *Hoeffding bounds* on the tail of the underlying distribution [12]. According to this idea, $\ln\left(\frac{2}{\delta}\right)/2\epsilon^2$ samples need to be obtained in order for the estimated probability Y to deviate from the real probability X by at most ϵ with probability $1-\delta$, i.e.

$Pr(|X - Y| \leq \epsilon) \geq 1 - \delta$. In this case, the number of traces grows logarithmically with δ and quadratically with ϵ. It is, however, interesting to note that the sample size is completely independent from the size of the underlying system.

Runtime verification: Runtime verification attempts to circumvent the combinatorial problems of conventional model checking by focussing on the *execution trace* of a system rather than on its universal behaviour and performing correctness checks on-the-fly [22]. Due to its focus on the *execution trace* of a running system, it avoids most of the complexity problems that are inherent to static techniques; in that respect, runtime verification bears a strong similarity to testing. However, in contrast to conventional testing, runtime verification typically allows for the formulation of the system's desired behaviour in a more rigorous way, e.g. using temporal logic, and can thus be considered more formal. In general, runtime verification provides a nice balance between rigorous and strong but complex formal verification one one hand, and efficient but significantly weaker conventional software testing on the other hand. It can thus be seen as a lightweight alternative for systems that are not amenable to formal verification.

Due to its focus on individual execution traces, runtime verification views time as a linear flow and properties are thus often formulated in a variant of linear temporal logic (LTL). Those properties are then translated into a *monitor* which is used to observe the execution of the system and report any satisfaction or violation that may occur. In order for a monitoring approach to be efficient, it necessarily needs to be *forward-oriented*; having to rewind the execution of a system in order to determine the truth of a property is generally not an option. In terms of monitor construction, two different approaches, *automaton-based* and *symbolic*, can be distinguished. In this work, we follow a symbolic approach which strongly relies upon the notion of *expansion laws*. Informally, expansion laws allow for the decomposition of an LTL formulae into two parts: the fragment of the formula that needs to hold in the *current* state and the fragment that needs to hold in the *next* state in order for the whole formula to be true. It is useful to view both fragments as *obligations*, i.e. aspects of the formula that the trace under consideration needs to satisfy immediately and those that it promises to satisfy in the next step. For example, the expansion law for the 'until' operator of LTL is shown below:

$$\phi_1 \, \mathbf{U} \, \phi_2 \equiv \phi_2 \vee (\phi_1 \wedge \mathbf{X}(\phi_1 \, \mathbf{U} \, \phi_2)) \tag{1}$$

The equivalence states that, in order for a formula $\phi_1 \, \mathbf{U} \, \phi_2$ to be satisfied at time t, either (i) ϕ_2 needs to be satisfied at time t, or (ii) ϕ_1 needs to be satisfied at time t and $\phi_1 \, \mathbf{U} \, \phi_2$ needs to be satisfied at time $t + 1$. Expansion laws play an important role for the idea of runtime verification since they form the basis for a decision procedure which can be used to decide in a certain state if a given property has already been satisfied or violated. By decomposing a formula into an immediate and a future obligation, optimality can be achieved: as soon as the immediate obligation is satisfied and no future obligation has been created, the entire formula is satisfied and the evaluation finishes.

5 The verification framework

Considering the complexity of ABSs, we believe that a combination of statistical model checking and runtime verification, which we refer to as *statistical runtime verification (SRV)*, may serve as an interesting alternative to formal verification. Due to their probabilistic nature, ABSs can be seen as special variants of Monte Carlo simulations and each execution thus naturally produces a random trace of the underlying space. By verifying a property on a sufficiently large number of simulation runs, its probability can thus be estimated to an arbitrary level of precision. Clearly, the usefulness of this idea is critically dependent on the number of traces analysed. As described further below, the efficiency of each trace check can be improved significantly by interleaving simulation and verification. As opposed to formal macro-level analysis, SRV preserves the individual richness of the ABS and allows for the verification of interesting properties in a semi-formal way. Due to its focus on independent traces, SRV is easily parallelisable and thus highly scalable by exploiting the power of modern parallel hardware.

MC^{2}MABS is a practical framework that incorporates the idea of SRV. Its design is based on four central requirements, as informally motivated in Section 2: (i) *efficiency* (timely and tunably accurate verification of large-scale ABSs), (ii) *expressivity* (formulation and verification of qualitative and quantitative correctness properties in a formal, rigorous way), (iii) *flexibility* (verification of arbitrary ABSs), and (iv) *immediacy* (verification of the ABS itself, not a simplified model thereof). An overview of the framework is given below, the source code as well as additional documentation is available online [1].

5.1 Architectural overview

A high-level overview of MC^{2}MABS is shown in Figure 1. The framework comprises as its central components (i) an estimator, (ii) a modelling framework, (iii) a property parser, (iv) a simulator, and (v) a runtime monitor. All components are described in more detail in Section 5.2 below. The typical sequence of actions in a verification experiment using MC^{2}MABS is as follows:

1. The user provides (i) the logic of the ABS by utilising the modelling framework, (ii) an associated correctness property, and (iii) the desired precision of the verification results as inputs to the verification framework.
2. The correctness property is parsed by the property parser and transformed into an expanded property that is used by the runtime monitor.
3. The estimator determines the number of simulation traces necessary to achieve the desired level of precision.
4. The simulator uses the model together with additional configuration information to produce a set of simulation traces.
5. Each simulation trace is observed by a runtime monitor which assesses the correctness of the trace using a given correctness property; due to the online nature of the monitor, a verdict is produced as soon as possible.
6. The individual results are aggregated into an overall verification result and presented to the user.

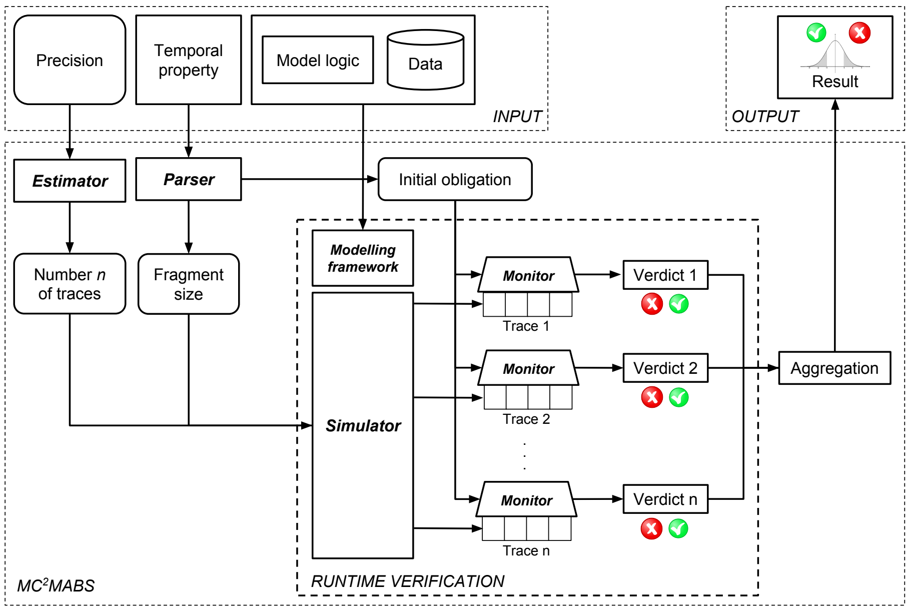

Fig. 1. The overall architecture of MC^{2}MABS

Due to the decoupling of simulation and verification, MC^{2}MABS supports both *ad-hoc* and *a-posteriori* verification. Ad-hoc verification is synonymous to runtime verification and assesses the correctness of a system during its execution. A-posteriori verification assumes the existence of traces prior to the actual verification. The latter mode can be useful, for example, if the traces have been obtained with a different simulation tool, e.g. NetLogo [38] or Repast [8]. In that case, the simulator of MC^{2}MABS is merely used to 'replay' the pre-existing output for the purpose of verification.

5.2 Components

Estimator: The main purpose of the estimator is to determine the number of traces necessary to achieve a certain level of precision (provided by the user) w.r.t. the verification results. MC^{2}MABS uses an algorithmic variant of the Hoeffding bounds briefly mentioned in Section 4. Due to its approximate nature, the Hoeffding bound often overestimates the actually required number of samples by a significant degree. The procedure we use instead operates directly on the Binomial distribution [13]. It has the same theoretical dependence on δ and ϵ but, due to its accurate nature, it returns a lower *total sample size*. In the presence of resource constraints, this (theoretically irrelevant) difference can represent a critical practical advantage. Different levels of precision and their corresponding sample size are shown below:

1. confidence $\delta = 99\%$, accuracy $\epsilon = 1\% \Rightarrow$ 13,700 traces
2. confidence $\delta = 99.9\%$, accuracy $\epsilon = 0.1\% \Rightarrow$ 24,000 traces
3. confidence $\delta = 99.9$, accuracy $\epsilon = 0.1\% \Rightarrow$ 2,389,000 traces

Modelling framework: Instead of providing a dedicated model description language — a path taken by most existing verification tools — we decided to allow for the formulation of the underlying model in a high-level programming language. This is motivated by the observation that ABSs often contain a significant level of functional complexity (probability evaluations, loading and manipulation of external data, location-based search algorithms, etc.). Any simple modelling language would thus significantly (and unnecessarily) limit the range of models which it is capable of describing. As a consequence, we decided to take a different path and realise the communication between the simulation model and the monitor through a *service provider interface (SPI)* which provides a basic skeleton for the underlying model and limits the prescriptive part of the framework to a handful of callback functions. In order to maintain a high level of performance (which is crucial for the generation of large batches of traces), we use C++ as the modelling language. As a compiled multi-paradigm language, C++ offers a good balance between usability and performance.

Alternatively, rather than hosting the actual model logic, the modelling framework can also be used to implement logic that controls an external simulation tool such as NetLogo or Repast (e.g. running in 'headless mode'), collects the resulting data and forwards it to the verifier. In this case, MC^{2}MABS acts as a 'man-in-the-middle' and extends existing simulation frameworks with a verification capability.

Property parser: The property parser is responsible for translating a textual representation of a correctness property into an expanded version which is then used by the monitor to observe the temporal dynamics of a simulation trace. The parser uses a formal grammar that defines the space of valid properties. MC^{2}MABS supports *simLTL*, a variant of LTL tailored to the formulation of properties about ABS traces [14]. As opposed to conventional LTL, simLTL allows for the formulation of properties about *individual agents* as well as about arbitrary *groups of agents*. This is achieved by a subdivision of the language into an *agent layer* and a *global layer*. Furthermore, the language is augmented with *quantification* and *selection* operators. These features make it possible to formulate properties such as the following:

– It is true for *every* agent that the energy level will never fall below 0
– No more than *20%* of the agents will eventually run out of energy
– Agents *of group x* are *more likely* to run out of energy than those of *group y*

Furthermore, as explained below, the formulation of properties is closely linked with the way the simulator performs the sampling from the probability space underlying the simulation model. This is also accommodated by the property specification language which allows for (i) the annotation of properties in order to denote the length of *trace fragments* required for their verification as well as (ii) the formulation of higher-order properties, e.g. about the correlation of events, as described in the next paragraph.

Simulator: The simulator is responsible for executing the simulation model repeatedly in order to obtain a set of traces used for subsequent verification by the monitor. Technically, by repeatedly executing the simulation model, the simulator performs a sampling from the underlying probability space. By interpreting the probability space in different ways, different levels of granularity with respect to property formulation can be achieved [16]. So, for example, by interpreting a trace of length k produced by the simulation model not as a single sample from the *distribution of traces* of length k but instead as a set of k samples from the *distribution of states*, properties about individual states and their likelihood become expressible; by interpreting the trace as a set of $k/2$ samples from the *distribution of subsequent states*, properties about transitions and their likelihood become expressible. In general, a single trace of length k can be interpreted as a set of samples of trace *fragments* of length $1 \leq i \leq k$. Furthermore, by relating probabilities of individual properties, statements about *correlations of events* can be made. This allows for a high level of granularity and expressivity with respect to property formulation and verification. Technically, the simulator is tightly interwoven with both the modelling framework and the monitor (described below). At the current stage, all simulation replications are executed sequentially. However, since the individual replications are entirely independent, the framework is efficiently parallelisable.

Monitor: The runtime monitor is the central component of the verification framework. Its main purpose is to observe the execution of a single trace as generated by the simulator and check its correctness against the background of a given property on-the-fly, i.e. while the trace is being produced. In the case of thousands of traces that need to be assessed, online verification represents a critical advantage: as soon as a property can be satisfied or violated, the monitor is able to produce a verdict and move on to the next trace. For properties that are satisfiable or refutable at some point along the trace, this leads to significant improvements in speed over an exhaustive approach. As indicated in Section 4, the core of a monitor is a temporal formula; it is constructed from the temporal property provided by the user by exploiting expansion laws. The monitor is written in Haskell. Apart from the code being close to the mathematical description of the algorithms, an important decision for choosing Haskell as the underlying programming language was its inherent support for *lazy evaluation*. Given the potentially considerable complexity of the underlying simulation, unnecessary computation is to be avoided in any case. Against this background, it is, for example, important to postpone calls from the monitor to the underlying simulator until a new state is strictly required for evaluating the current property (as defined by the expansion laws). Furthermore, it is important to keep the underlying simulation strictly *forward-oriented*, i.e. such that ticks are simulated in ascending order only and no tick is simulated twice. In that context, Haskell's lazy evaluation strategy is of great help. To illustrate this, consider the problem of evaluating a property ψ on fragments of a trace π. In an offline setting, π would have to be constructed in its entirety prior to evaluation. If ψ is either satisfiable or refutable on a prefix of π, computation would be wasted. In order to avoid that, π must not be produced prior to evaluation. To this end, instead of holding a sequence of global states (which it would if π was the result of a full simulation run), π holds a sequence of *thunks*, i.e. not yet evaluated expressions. Thanks to Haskell's lazy evaluation strategy, a thunk is only evaluated if strictly necessary. In this way, the simulation of the next time step can be postponed in order to achieve the desired online effect. Furthermore, lazy evaluation supports easy *sharing* of computations. Each tick is thus only simulated once which achieves the strict monotonicity effect mentioned above.

5.3 Performance evaluation

In this section, we provide a brief empirical performance evaluation of MC^{2}MABS. A more detailed evaluation can be found elsewhere [13, 1]. We use as our example model a simple disease transmission scenario in which each agent can be either *susceptible*, *infected*, or *recovered*. We assume that transitions between the states are probabilistic and hardcoded and agents are entirely independent. We are aware that, as a consequence of these simplifying assumptions, the model could well be analysed analytically. Choosing an approximate approach may thus seem unnecessarily limiting here. However, it is *not* our goal to perform a realistic verification experiment here. We rather aim to illustrate the effects of *online verification* — the central aspect of our approach — on the performance of the tool. To this end, we have deliberately chosen a simple model. Since MC^{2}MABS is completely agnostic about the internals of the underlying model and solely focussed on the resulting *traces*, the simplicity of the model does not negatively impact the evaluation results. More comprehensive case studies that focus on the *usefulness* of MC^{2}MABS for the analysis of swarm-robotic scenarios can be found elsewhere [13, 15]. For the evaluation, we focus on the following four properties:

1. '$\mathbf{F}$ *allInf*': unquantified group property, not refutable before the end of the trace
2. '$\mathbf{G}$ *allInf*': unquantified group property, immediately refutable
3. '$\mathbf{F}(\forall\, inf)$': quantified group property, not refutable before the end of the trace
4. '$\mathbf{G}(\forall\, inf)$': quantified group property, immediately refutable

'*allInf*' describes a population-level proposition that is true if and only if all agents in the population are infected. '*inf*' describes an individual proposition that is true if and only if the agent under consideration is infected. Due to the use of 'finally' and 'globally', respectively, Properties 1 and 2 differ in terms of their *satisfiability*: Property 2 is immediately refutable, whereas the satisfiability of Property 1 can only be determined at the end of the trace. As shown below, this has a significant impact on the time needed for verification. The same is true for Properties 3 and 4.

Properties 1 and 3 as well as Properties 2 and 4 are semantically equivalent; they only differ in terms of their *observational level*: Properties 1 and 3 make a statement about the population as a whole, whereas Properties 2 and 4 are *individual* in nature; the distinction is only made to show the impact of quantification on performance.

We assess the performance against two dimensions: *runtime* and *memory consumption*. Since the executable binary file of MC^{2}MABS is a merge of code written in both C++ and Haskell, their independent evaluation is not easily possible. For that reason, all individual measurements have to be derived from the profiling results of the entire application. We focus on four major tasks:

Simulation (SIM): Time spent on executing the model logic; this describes the performance of the C++-based simulator.

Extraction (EXT): Time spent on extracting and transforming (*marshalling*) the group traces created by the C++ simulator into their corresponding Haskell vectors.

Verification (VER): Time spent on the actual evaluation of the simLTL property.

Other (OTH): Time spent on 'housekeeping', i.e. other, non simulation- or evaluation-related tasks such as garbage collection, system calls and profiling itself.

Table 1. Runtime consumption (in seconds) for different population sizes

	10 agents					100 agents					1,000 agents				
Prop.	SIM	VER	EXT	OTH	TOT	SIM	VER	EXT	OTH	TOT	SIM	VER	EXT	OTH	TOT
1	0.022	0.023	0.043	0.012	0.100	0.185	0.081	0.383	0.056	0.706	2.480	0.675	4.361	2.510	10.027
2	0.000	0.006	0.001	0.003	0.010	0.001	0.004	0.003	0.002	0.010	0.021	0.025	0.045	0.029	0.120
3	0.025	0.040	0.042	0.014	0.121	0.242	0.119	0.517	0.079	0.957	2.798	0.715	4.902	2.798	11.213
4	0.001	0.006	0.000	0.002	0.010	0.001	0.004	0.003	0.002	0.010	0.022	0.027	0.036	0.025	0.110

Total (TOT): Total runtime of MC^{2}MABS.

The evaluation was performed using `gprof`, Haskell's built-in profiling system on a 64 Bit Dell Latitude Laptop with two Intel® Core$^{\text{TM}}$ 2 Duo CPUs (2.8 GHz each), 8GB of memory and Linux Mint Rebecca (kernel version 3.13.0-24) as operating system. The numbers are averaged over 10 runs, each of which involves the execution of 100 individual simulation runs for the purpose of probability estimation. The results w.r.t. runtime consumption are shown in Table 1, the key points are briefly summarised below.

- MC^{2}MABS scales linearly with the size of the underlying population.
- Satisfiability of the formula has a large impact on the time spent on each task. Consider, for example, formula '**G** *allInf*' which is immediately refutable; in this case, evaluation is very quick, even for large populations.
- As the population size grows, an increasing fraction of time is spent on extraction (i.e. marshalling the data structures between C++ and Haskell) and housekeeping, in particular garbage collection. This may represent a bottleneck for large populations which we aim to address as part of the future work.
- MC^{2}MABS also scales linearly with the population size in the case of universally quantified formulae. However, housekeeping (particularly garbage collection) becomes a serious overhead as the number of agents grows. We plan to address this issue in the future, e.g. by employing strictness in some of the operations.

Table 2. Memory allocation (in Bytes) for different population sizes

	10 agents					100 agents					1,000 agents				
Pr.	SIM	VER	EXT	OTH	TOT	SIM	VER	EXT	OTH	TOT	SIM	VER	EXT	OTH	TOT
1	6.4e5	9.7e6	4.5e7	5.7e7	5.7e7	6.4e5	9.7e6	4.4e8	4.5e8	4.5e8	6.4e5	9.7e6	4.4e9	4.4e9	4.4e9
2	6.4e3	1.0e6	4.5e5	3.0e6	3.0e6	6.4e3	1.0e6	4.4e6	6.9e6	6.9e6	6.4e3	1.0e6	4.4e7	4.7e7	4.7e7
3	6.4e5	2.5e7	4.5e7	7.3e7	7.3e7	6.4e5	2.5e7	4.4e8	4.7e8	4.7e8	6.4e5	2.5e7	4.4e9	4.4e9	4.4e9
4	6.4e3	1.2e6	4.5e5	3.4e6	3.4e6	6.4e3	1.2e6	4.4e6	7.3e6	7.3e6	6.4e3	1.2e6	4.4e7	4.7e7	4.7e7

Profiling memory consumption for a lazy language like Haskell can be difficult. For example, expressions without arguments, so-called *Constant Application Forms (CAFs)*, are evaluated only once and shared for later use. Due to their global scope, CAFs are thus, strictly speaking, not part of the call graph and hence need to be treated differently. Straightforward analysis of memory allocated within the call graph only can thus be misleading. In the analysis below, all CAFs are aggregated under the 'Other' section. The results are shown in Table 2. The key points are briefly summarised below.

- Memory consumption for both verification and simulation is constant and memory consumption for extraction and marshalling increases linearly with population size.

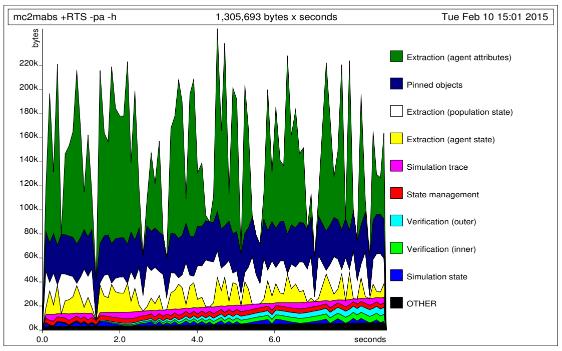

Fig. 2. Runtime heap profile for 1,000 agents

– Verification is the only evaluation step that the formula size has an impact on, which coincides with the runtime behaviour.

It is important to note that total memory allocation is not sufficient for understanding the full allocation behaviour of the program. It is useful to also analyse the *runtime heap profile* which describes memory allocation *over time*. An example for 1,000 agents (Property 3) is shown in Figure 2. For clarity, we restrict the number of functions and data structures shown to 10. In the graph, 'Pinned objects' refers to information in memory which is not movable by the garbage collection, e.g. memory allocated in the C++ part of the application. Furthermore, some of the functions in the Haskell implementation are split up into an inner and an outer part for technical reasons; this distinction is also reflected in the graphs. Finally, due to the pure functional nature of Haskell, global state cannot be maintained. In order to emulate this functionality, alternative options such as the *State Monad* have to be used. This state management accounts for a certain level of memory allocation which is also considered in the analysis. The graph shows that the peak memory allocation is stable and fairly low compared with the overall memory consumption; the graphs also show that the amount of garbage collection (indicated by the reduction in memory consumption) is clearly a function of the runtime.

6 Case studies

For space constraints, we cannot provide a full case study here. An exhaustive evaluation of three different versions of the swarm foraging scenario introduced in Section 2 including source code is provided in the first author's PhD thesis [13]. Another evaluation of the same scenario with a particular focus on the quantitative capabilities of MC^{2}MABS is provided in [16]. The paper illustrates how properties about transition probabilities, residence probabilities, state distributions, and correlations between events can be formulated and answered purely based on the analysis of individual traces.

7 Conclusions and future work

This paper described the architecture, design decisions and implementation details of MC^{2}MABS, a statistical runtime verification framework for ABSs. With properties formulated in temporal logic, high scalability due to the focus on individual and independent simulation traces and its ability to provide confidence intervals for the results, it aims to combine some of the strengths of both formal and informal verification techniques into a common framework. The approach aims to satisfy the four requirements introduced in Section 2 as follows:

Efficiency: Due to the approximate nature of the approach, simulation models with a large number of constituents are efficiently verifiable. The number of simulation runs necessary for verification is only dependent on the desired level of precision and not on the size of the underlying system. Since individual traces are entirely independent, the approach is also inherently parallel and therefore highly scalable.
Expressivity: Properties can be formulated in temporal logic and checked automatically. The syntax of the specification language provides quantification and selection operators and thus allows for the formulation of properties on different observational levels. In addition to that, the trace fragment-based semantics allow for the verification of quantitative properties.
Flexibility: Due to the reliance on simulation traces rather than on a formal model, arbitrary simulation models are verifiable.
Immediacy: Verification is performed upon the output of the original simulation.

It is important to stress here that we do not aim to propose a substitute for purely formal verification. In cases where formal verification is feasible, it is clearly preferable over an approximate approach such as the one describe here. However, for cases which are not (and may never be) formally verifiable, MC^{2}MABS can present an interesting alternative and help to gain a better understanding of the emergent simulation dynamics than is currently possible. First experiments produced encouraging results and showed that the tool allows for the verification of complex simulation models with a high level of confidence ($> 99\%$) and accuracy ($< 1\%$) in a timely manner [13, 15]. Before MC^{2}MABS can be used efficiently in the real world, there are, of course, still plenty of limitations and open problems to be overcome. Some are mentioned below.

Accuracy: It is clear that, for highly safety-critical areas, a significantly higher level of precision than that used in our experiments is needed. With the current estimation procedure, the number of simulation traces increases quadratically with the level of accuracy which represents a critical bottleneck. One way to remedy this problem is to use ideas from *rare event sampling* in order to reduce the sample size needed.

Efficiency: An important advantage of trace-based verification is that it is efficiently parallelisable. At the current stage, MC^{2}MABS performs verification sequentially and does not exploit the capabilities of modern parallel hardware. A second important starting point for performance improvements is the performance of the simulation itself. For example, by making use of C++ template metaprogramming, some of the runtime calculations can be shifted towards compile time. Finally, MC^{2}MABS is technically subdivided into a simulation (written in C++) and a verification part (written in Haskell). Marshalling, i.e. translating and transferring the data structures between the two languages represents a significant bottleneck which also negatively influences the capability of the tool to analyse large batches of simulation traces.

Usability: The tool is still in a prototypical state and its usability is therefore still fairly low. Temporal logic and C++ are certainly not the most typical skills of an agent-based modeller. The choice has been made for the purpose of rigour and performance. But it is clear that, if the tool is to be used practically, higher-level interfaces need to be developed. The same holds for the connection to existing simulators such as Repast or NetLogo which currently requires manual efforts.

References

1. MC^{2}MABS website. https://github.com/bherd/mc2mabs. Access: 02/15.
2. C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.
3. P. Ballarini, M. Fisher, and M. Wooldridge. Uncertain agent verification through probabilistic model-checking. In *Safety and Security in Multiagent Systems*, volume 4324 of *LNCS*, pages 162–174. Springer, 2009.
4. G. K. Bharathy and B. Silverman. Validating agent based social systems models. In *Proc. Winter Simulation Conference*, pages 441–453. Winter Simulation Conference, 2010.
5. T. Bosse and N. Mogles. Comparing modelling approaches in aviation safety. In R. Curran, editor, *Proc. 4th Int. Air Transport and Operations Symposium, Toulouse, France*, 2013.
6. I. Cakirlar, . Grcan, O. Dikenelli, and S. Bora. RatKit: A repeatable automated testing toolkit for agent-based modeling and simulation. In *Proc. 15th Int. Workshop on Multi-Agent-Based Simulation*, 2014.
7. Y. U. Cao, A. S. Fukunaga, and A. Kahng. Cooperative mobile robotics: Antecedents and directions. *Autonomous Robots*, 4(1):7–27, Mar. 1997.
8. N. Collier. Repast: An extensible framework for agent simulation. *Natural Resources and Environmental Issues*, 8, 2001.
9. M. Dastani, K. V. Hindriks, and J.-J. Meyer. *Specification and verification of multi-agent systems*. Springer Science & Business Media, 2010.
10. T. De Wolf, T. Holvoet, and G. Samaey. Development of self-organising emergent applications with simulation-based numerical analysis. In *Engineering Self-Organising Systems*, volume 3910 of *LNCS*, pages 138–152. Springer, 2006.
11. M. I. Dekhtyar, A. J. Dikovsky, and M. K. Valiev. Temporal verification of probabilistic multi-agent systems. In *Pillars of Computer Science*, pages 256–265. Springer, 2008.
12. T. Hérault, R. Lassaigne, F. Magniette, and S. Peyronnet. Approximate probabilistic model checking. In *Proc. 5th Int. Conference on Verification, Model Checking and Abstract Interpretation*, volume 2937 of *LNCS*, pages 307–329. Springer, 2004.
13. B. Herd. *Statistical runtime verification of agent-based simulations*. PhD thesis, King's College London, 2015.
14. B. Herd, S. Miles, P. McBurney, and M. Luck. An LTL-based property specification language for agent-based simulation traces. TR 14-02, King's College London, Oct 2014.
15. B. Herd, S. Miles, P. McBurney, and M. Luck. Approximate verification of swarm-based systems: a vision and preliminary results. In *Engineering Systems for Safety: Proc. 23rd Safety-critical Systems Symposium*. CreateSpace Independent Publishing Platform, 2015.
16. B. Herd, S. Miles, P. McBurney, and M. Luck. Towards quantitative analysis of multiagent systems through statistical model checking. In *3rd Int. Workshop on Engineering Multiagent Systems*, 2015.
17. S. Karnouskos and T. de Holanda. Simulation of a smart grid city with software agents. In *3rd Europ. Symposium on Computer Modeling and Simulation*, pages 424–429, Nov 2009.
18. S. Konur, C. Dixon, and M. Fisher. Formal verification of probabilistic swarm behaviours. In *Swarm Intelligence*, volume 6234 of *LNCS*, pages 440–447. Springer, 2010.
19. S. Konur, C. Dixon, and M. Fisher. Analysing robot swarm behaviour via probabilistic model checking. *Robotics and Autonomous Systems*, 60(2):199–213, 2012.

20. M. Kwiatkowska, A. Lomuscio, and H. Qu. Parallel model checking for temporal epistemic logic. In *Proc. 19th European Conf. on Artificial Int.*, pages 543–548, 2010. IOS Press.
21. A. Legay, B. Delahaye, and S. Bensalem. Statistical model checking: an overview. In *Proc. 1st Int. Conf. on Runtime Verification*, pages 122–135. Springer, 2010.
22. M. Leucker and C. Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293 – 303, 2009.
23. W. Liu, A. Winfield, and J. Sa. Modelling swarm robotic systems: A case study in collective foraging. In *Towards Autonomous Robotic Systems*, pages 25–32, 2007.
24. W. Liu, A. Winfield, J. Sa, J. Chen, and L. Dou. Strategies for energy optimisation in a swarm of foraging robots. In volume 4433 of *LNCS*, pages 14–26. Springer, 2007.
25. A. Lomuscio, W. Penczek, and H. Qu. Partial order reductions for model checking temporal-epistemic logics over interleaved multi-agent systems. *Fundamenta Informaticae*, 101(1-2):71–90, Januar 2010.
26. A. Lomuscio, W. Penczek, and B. Woźna. Bounded model checking for knowledge and real time. *Artificial Intelligence*, 171(16-17):1011 – 1038, 2007.
27. I. Lorscheid, B.-O. Heine, and M. Meyer. Opening the 'black box' of simulations: increased transparency and effective communication through the systematic design of experiments. *Computational and Mathematical Organization Theory*, 18(1):22–62, 2012.
28. C. M. Macal and M. J. North. Tutorial on agent-based modelling and simulation. *Journal of Simulation*, 4(3):151–162, 2010.
29. R. E. Marks. Validating simulation models: A general framework and four applied examples. *Computational Economics*, 30:265–290, October 2007.
30. R. McCune and G. Madey. Agent-based simulation of cooperative hunting with UAVs. In *Proc. of the Agent-Directed Simulation Symposium*. Society for Comp. Sim. Int., 2013.
31. B. McKelvey. *The Blackwell Companion to Organizations*, chapter Model-centered organization science epistemology, pages 752–780. Blackwell, 2002.
32. M. Niazi, A. Hussain, and M. Kolberg. Verification and validation of agent based simulations using the VOMAS approach. In *Proc. 3rd Workshop on Multi-Agent Systems and Sim.*, 2009.
33. T. Pedersen and S. K. Dyrkolbotn. Agents homogeneous: A procedurally anonymous semantics characterizing the homogeneous fragment of atl. In *Principles and Practice of Multi-Agent Systems (PRIMA'13)*, volume 8291 of *LNCS*, pages 245–259. Springer, 2013.
34. D. Phan and F. Varenne. Agent-based models and simulations in economics and social sciences: From conceptual exploration to distinct ways of experimenting. *Journal of Artificial Societies and Social Simulation*, 13(1):5, 2010.
35. R. G. Sargent. Verification and validation of simulation models. In *Proc. 40th Winter Simulation Conference (WSC '08)*, pages 157–169, 2008.
36. S. Sebastio and A. Vandin. MultiVeStA: Statistical model checking for discrete event simulators. In *Proc. 7th Int. Conf. on Performance Evaluation Methodologies and Tools*, 2013
37. W. Wan, J. Bentahar, and A. Ben Hamza. Model checking epistemic and probabilistic properties of multi-agent systems. In *Modern Approaches in Applied Intelligence*, volume 6704 of *LNCS*, pages 68–78. Springer, 2011.
38. U. Wilensky. NetLogo. TR, Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL., 1999.
39. C. J. Wright, P. McMinn, and J. Gallardo. Towards the automatic identification of faulty multi-agent based simulation runs using MASTER. In *Multi-Agent-Based Simulation XIII*, volume 7838 of *LNCS*, pages 143–156. Springer, 2013.