Verification & Validation of Agent-Based Simulations using Approximate Model Checking

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

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

Abstract. This paper focusses on the usefulness of approximate probabilistic model checking for the internal and external validation of large-scale agent-based simulations. We describe the translation of typical validation criteria into a variant of linear time logic. We further present a prototypical version of a highly customisable approximate model checker which we used in a range of experiments to verify properties of large scale models whose complexity prevents them from being amenable to conventional explicit or symbolic model checking.

Keywords: agent-based simulation, verification, validation, model checking

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. It is applied successfully to an ever-increasing number of real-world problems and could in many areas show advantages over traditional numerical and analytical approaches. Due to the high level of complexity, however, ABSs are difficult to understand, to verify and to validate. In order to deal with the large number of behaviours that a model can exhibit, random variance in the output and an often huge input parameter space, comprehensive experiments need to be conducted in all stages of the development process as well as during productive use. Insight into the dynamics is typically obtained by analysing the output (often by executing complex database queries), by statistical analysis and inductive reasoning. Similar to other software systems, correctness also plays a central role in agent-based simulation and questions of quality assurance become increasingly important [17]. In this context, it is important to distinguish between *verification* and *validation*. Whereas the former is targeted towards a system's correctness with respect to its specification, the latter ensures a sufficient level of accuracy with respect to the intended application domain, i.e. the real-world phenomenon in an ABS. Verification is typically associated with correctness of the *implementation* whereas validation is more targeted towards the system's *representativity*. The boundary between verification and validation is often blurred, particularly in the context of simulation models. In this paper, we are not concerned with the actual implementation of an ABS but rather with its conceptual correctness. We will employ the following terminology: By *validation*, we refer to the general process of assessing the conceptual correctness of an ABS. In order to distinguish between correctness checks

which concern the internal dynamics of the model only and those which include external (e.g. historical) data, we use the terms *internal* and *external validation*, respectively. By *verification* we refer to the technical process of answering a formalised question using a rigorous approach, i.e. a formal verification technique such as model checking. In short, we aim to use formal verification techniques for the purpose of ABS validation.

Due to their interconnected and emergent nature, ABSs typically exhibit a large semantic gap between their static (the code) and dynamic (the runtime behaviour) representation. In his seminal paper, Dijkstra points out a problem which will probably sound familiar to many of those working in agent-based modelling [6]: "*My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed*". Although he talks about the `goto` statement and its implications on code quality, the problem that we face in agent-based modelling is similar: what makes an ABS so hard to understand and control (and renders static analysis for its validation largely useless) is the fact that the global behaviour cannot easily be anticipated by scrutinising the behavioural logic of the individuals. Interaction between individual agents following simple (and well understood) behavioural rules may lead to positive or negative feedback loops which may either amplify, reduce or even cancel out certain effects entirely. It is exactly this element of surprise which makes ABSs powerful and complicated at the same time. Even the most experienced modeller will face situations in which the observed behaviour diverges significantly from what was intended or expected. It is the modeller's task to make sure that desired things happen and undesired things do not. It is of particular importance during this process to establish whether observed global behaviours are actually emerging from the individual's local rules or whether they are caused by an undesired mechanism, an *artefact* [10]. This, however, is a nontrivial task which requires deep understanding of the model's dynamics. The best way to understand the dynamics of an ABS is to build up a comprehensive set of validation criteria, i.e. a description of the desired behaviours, and test the system thoroughly and systematically against it. In order to accomplish that, we require two essential ingredients:

1. a language to describe validation criteria in a formal and unambiguous way
2. an automated mechanism to check if an ABS satisfies its validation criteria.

This paper addresses both points. Our contributions are (i) a demonstration of the usefulness of approximate model checking for ABS validation (Section 4); (ii) a description of how common ABS validation criteria can be formalised in a probabilistic variant of linear temporal logic (Sections 5 and 6); and, (iii) the description of a prototype infrastructure for ABS validation with approximate model checking (Section 7).

The paper starts with a brief overview of related work on formal ABS analysis and verification and its limitations in Section 2 followed by some theoretical background in Section 3. Our ideas have been evaluated in a range of experiments which are described in Section 8. Here we demonstrate the application of approximate model checking to an ABS of considerable size and show its benefits for the analysis as well as its advantage over conventional model checking.

2 Related Work

In the following, we describe related work in the area of formal verification for probabilistic agent-based systems and simulations. A good overview of statistical validation techniques (which is omitted here) is given in Kleijnen's and Sargent's papers [14, 18].

Despite the growing importance of ABS, dedicated formal verification techniques for their analysis are still largely missing. In recent years, probabilistic model checking has gained increasing importance for general multiagent systems (of which ABSs are a special type). An interesting approach to verify the emergent behaviour of robot swarms using probabilistic model checking has been presented by Konur *et al.* [15]. In order to tackle the combinatorial explosion of the state space, the authors exploit the high level of symmetry in the model and use *counter abstraction* [9]. In doing so, the authors manage to transform the problem which is originally exponential in the number of agents into one which is polynomial in the number of agents and exponential in the number of agent states. This is a significant improvement, yet it still remains limited to relatively small-scale systems.

Ballarini *et al.* [3] apply probabilistic model checking to a probabilistic variant of a negotiation game. They use PRISM [16], a probabilistic model checker, to verify PCTL (**P**robabilistic **C**omputation **T**ree **L**ogic) [2] properties referring to (i) the value at which an agreement between two agents bargaining over a single resource is reached and (ii) the delay for reaching an agreement. In this scenario, the overall state space is small and therefore combinatorial explosion is not an issue. According to the authors, probabilistic verification provides an interesting alternative to analytical and simulation methods and can provide further insight into the system's behaviour.

Dekhtyar *et al.* [5] describe a method to translate a multiagent system into a finite-state Markov chain and analyse the complexity of probabilistic model checking of its dynamic properties. Apart from mentioning the exponential complexity of both state space creation and verification, however, the authors do not present any ways to circumvent this problem. The verification of epistemic properties has also been addressed against the background of probabilistic agent-based systems.

Wan *et al.* [20] propose PCTLK, an epistemic, probabilistic branching-time logic which extends CTL (**C**omputation **T**ree **L**ogic) [2] with probabilistic and epistemic operators. In their paper, the focus of interest is rather on agent internals and thus complexity issues are not addressed.

A different formal approach to the analysis and verification of agent-based simulations has been proposed by Izquierdo *et al.* [13], who describe how simulations can be encoded into time-homogeneous Markov chains and analysed with respect to their transient and steady-state behaviour. Since the main focus of the paper is on the usefulness of Markov chain analysis for the understanding of complex simulation models, the authors do not provide any state space reduction techniques in order to circumvent the combinatorial explosion. However, they describe ways of analysing the limiting behaviour without having to represent the transition matrices by mere reasoning about the nature of the state space and deriving characteristics of the corresponding Markov chain. Despite advances, however, the verification of properties of large-scale ABSs with a focus on macro-level behaviour remains a largely unsolved problem.

3 Background

Probabilistic transition systems: A Probabilistic Transition System (PTS) is a tuple $\mathcal{M} = (S, P, I, AP, L)$ where S is a countable (but possibly infinite) set of states, $P : S \times S \rightarrow [0, 1]$ is a transition probability function such that for each $s \in S$: $\sum_{s' \in S} P(s, s') = 1$, $I \subseteq S$ is the set of initial states, AP is a set of atomic propositions and $L : S \rightarrow 2^{AP}$ is a labelling function. A path σ through a PTS is a finite or infinite sequence of states $(s_0, s_1, ...,)$ such that $P(s_i, s_{i+1}) > 0$ for all $i \geq 0$. By $\sigma[n]$ we refer to the n-th element of path σ and by $\sigma[j..]$ to the fragment of σ starting in $\sigma[j]$. The state space of an ABS can be modelled as a PTS in which each state represents a global state of the simulation. Due to the hierarchical nature of an ABS, each global state is itself comprised by n individual agent states plus the state of the environment. Each simulation run thus corresponds to one particular path σ through the underlying PTS.

Linear temporal logic: The treatment of time in temporal logic can be roughly subdivided into *branching time* (CTL, CTL*) and *linear temporal logic* (LTL) [2]. Branching time logics assume that there is a choice between different successor states at each time step and thus views time as an exponentially growing tree of 'possible worlds'. Linear time logic views time as a linear sequence of states. The approach described in this paper is based upon the analysis of individual finite paths representing simulation output. Since each path comprises a sequence of states, it is natural to assume linear temporal flow. We thus focus on LTL, the syntax of which is given below:

$$\phi ::= \text{true} \mid a \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg \phi \mid \mathbf{X}\phi \mid \phi \, \mathbf{U} \, \phi \mid \phi \, \mathbf{U}^{\leq k} \, \phi \qquad (1)$$

The basic building blocks are atomic propositions $a \in AP$, the Boolean connectives $\wedge$ ('and'), $\vee$ ('or') and $\neg$ ('not') and the temporal connectives $\mathbf{X}$ ('next') and $\mathbf{U}$ ('until') including a bounded variant. LTL formulae are evaluated over paths. For formula ϕ and state s, true always holds, a holds iff $a \in L(s)$, $\phi_1 \wedge \phi_2$ holds iff ϕ_1 holds and ϕ_2 holds, $\phi_1 \vee \phi_2$ holds iff either ϕ_1 or ϕ_2 holds, $\neg \phi$ holds iff ϕ does not hold and $\mathbf{X}\phi$ holds iff ϕ holds in the direct successor state of s. For formulae ϕ_1 and ϕ_2, $\phi_1 \, \mathbf{U} \, \phi_2$ holds in state s iff ϕ_1 holds in s and ϕ_2 holds at some point in the future. $\phi_1 \, \mathbf{U}^{\leq k} \, \phi_2$ holds iff ϕ_1 holds in s and ϕ_2 holds at some point within the next k time steps. Other logical connectives such as '$\Rightarrow$' or '$\Leftrightarrow$' can be derived in the usual manner: $\phi_1 \Rightarrow \phi_2 \equiv \neg \phi_1 \vee \phi_2$ and $\phi_1 \Leftrightarrow \phi_2 \equiv (\phi_1 \Rightarrow \phi_2) \wedge (\phi_2 \Rightarrow \phi_1)$. Additional temporal operators such as $\mathbf{F}$ ('finally','eventually'), $\mathbf{G}$ ('globally','always') and $\mathbf{W}$ ('weak until') can be derived as follows: $\mathbf{F}\phi \equiv \text{true} \, \mathbf{U} \, \phi$ (ϕ *holds eventually*), $\mathbf{F}^{\leq k}\phi \equiv \text{true} \, \mathbf{U}^{\leq k} \, \phi$ (ϕ *holds eventually within k time steps),* $\mathbf{G}\phi \equiv \neg \, \mathbf{F}(\neg \, \phi)$ (ϕ *holds always*) and $\phi_1 \mathbf{W} \phi_2 \equiv (\phi_1 \, \mathbf{U} \, \phi_2) \vee \mathbf{G}\phi_1$ (ϕ_1 may be succeeded by ϕ_2). It is often helpful to use certain combinations of operators. For example, $\mathbf{GF}\phi$ states that *"ϕ is satisfied infinitely often"*; $\mathbf{FG}\phi$ states that *"ϕ will eventually hold forever"*.

Approximate model checking: Model checking [4] is a popular verification technique which uses a formal representation $\mathcal{M}$ of the system under consideration (usually a finite state model) together with a specification of the system's desired properties p, typically given in temporal logic. The verification of a system's correctness is then

done by checking whether a given property p holds (formally $\mathcal{M} \models p$) by examining all possible execution paths. In the case of violation, the model checker can provide a counterexample. In order to deal with inherently random systems, probabilistic extensions to model checking have been developed [16]. Despite impressive advances made in recent years, exponential growth of the underlying finite-state model (the so-called *state space explosion*) remains a central problem which makes the verification of non-trivial real-world systems difficult or even impossible. Apart from symbolic model checking, a number of techniques have been developed in order to tackle this problem, such as reduction, abstraction, compositional verification and approximation. In this paper, we focus on the latter idea. Approximate (also *statistical*) model checking (AMC) is based on the following principle: n paths from the state space underlying the system under consideration $\mathcal{M}$ are obtained through random sampling. It is then checked for each path σ whether σ satisfies a given linear-time property ϕ, denoted $\sigma \models \phi$. The verification of ϕ on path σ can be considered a Bernoulli trial with either positive or negative outcome. Let A denote the number of successes in a sequence of N Bernoulli trials. The overall probability of $\mathcal{M}$ satisfying ϕ, denoted $Pr(\phi)$, can then be approximated by A/N. Clearly, the number of samples – i.e. paths verified – has a significant impact on the accuracy of the result. By varying it, the *confidence* with which the resulting probability reflects the actual probability can be adjusted. Several approaches to address this problem have been proposed. We follow the work by Hérault *et al.* [11] who provide a *probabilistic guarantee* on the accuracy of the approximate value generated by using *Chernoff-Hoeffding bounds* on the tail of the underlying distribution. According to this idea, $\ln\left(\frac{2}{\delta}\right)/2\epsilon^2$ samples need to be obtained in order to achieve a result Y that deviates from the real probability X by at most ϵ with probability $1-\delta$, i.e. $Pr(|X - Y| \leq \epsilon) \geq 1 - \delta$. This results in the *Generic Approximation Algorithm ($\mathcal{GAA}$)* outlined in Alg. 1. It accepts four inputs: a path generator $pathGen$[1], an LTL property ϕ, the desired path length k, an approximation parameter ϵ and a confidence parameter δ. The algorithm obtains N samples where N is a function of ϵ and δ. Property ϕ is then evaluated separately for each path. Every successful evaluation increases a counter variable A. A/N then provides an estimation of the actual probability of ϕ.

Algorithm 1 Generic Approximation Algorithm $\mathcal{GAA}$

Input: $pathGen, \phi, k, \epsilon, \delta$
  $N := \ln\left(\frac{2}{\delta}\right) \cdot \frac{1}{2\epsilon^2}$
  $A := 0$
  for $i := 1$ to N **do**
    1. Generate a random path σ of length k using $pathGen$
    2. If ϕ is true on σ then $A := A + 1$
  end for
  return A/N

[1] In Hérault's paper, this parameter is referred to as $diagram$ [11].

4 Agent-Based Simulation and Approximate Model Checking

In agent-based modelling, conventional statistical analysis is the predominant approach to validation. To this end, simulation output is often written to a relational database, extracted with SQL and analysed with statistical methods. This is convenient for exploratory analysis, i.e. for summarising the main statistical characteristics of the output or for comparing the simulation output with historical data. Formulating complex behavioural expectations including temporal relationships, however, can become a manual and time-consuming process. Consider, for example, the following statement about the desired behaviour of an agent in a transmission simulation: *"it should hold for the majority of agents that, whenever an agent is susceptible, it must become infected within t time steps, otherwise it must recover again"*. Translating such a statement into a statistical test is nontrivial. The situation is further complicated in the presence of randomness, i.e. if transitions only occur with a certain probability. As a consequence, the nature of any validation effort is typically highly tailored to the project and characterised by the development of custom validation scripts which exacerbates scalability and reusability.

On the opposite side of the spectrum of quality control, formal verification – particularly model checking – has been used successfully in safety critical areas such as air traffic control, nuclear reactor protection or railway signalling. The mathematical rigour of model checking as well as its exhaustive nature make it a powerful verification tool. On the other hand, its applicability is hampered by the high level of expertise required as well as by well-known scalability issues. The high complexity of ABSs on one hand and their incompatibility with compositional verification techniques on the other hand renders conventional model checking unsuitable as a tool for their validation.

Approximate model checking (AMC) offers a nice balance between strict formal verification and statistical analysis. We believe it is of particular usefulness for ABS validation, for reasons given in the following. First, because of its non-exhaustive nature, AMC is interesting for systems whose complexity prevents the usage of conventional model checking. The parameters of the randomised algorithm can be used to approximate the actual verification result to an arbitrary degree of accuracy. Since the achievable accuracy is a function of the number of sample paths generated and this generation process may be expensive, AMC is particularly suitable for the verification of non-safety-critical systems for which an approximate result is sufficient. The vast majority of ABSs is extraordinarily complex but at the same time also non-safety-critical; a quick verification process that produces reasonably accurate results is thus mostly preferable over a highly precise, time-consuming one. Second, AMC allows for the separation of path creation (i.e. simulation) from actual verification. This makes it possible to verify blackbox systems from which only the output is available and whose logic cannot (or only with considerable effort) be translated into a more formal, lower-level language as required by conventional model checkers such as PRISM [16] or Spin [12]. The separation also allows for easy integration of the model checker into an existing simulation environment. Third, due to typically stochastic nature, repeated execution of an ABS provides a natural sampling of paths from the underlying probability space as required by AMC. Finally, the use of temporal logic allows for the formulation of complex behaviours in a succinct and descriptive way which makes it possible to formulate expectations about all observational levels (*micro*, *meso* and *macro*). Due to its

approximate nature and in contrast to other model checking techniques which focus on individual or small groups of agents, AMC is perfectly capable of dealing with huge populations. This makes it a suitable candidate not only for the analysis of individual or organisational but also for global, emergent behaviours of arbitrarily large ABSs.

It is important to note that, due to its focus on finite paths, AMC is not able to analyse the steady-state behaviour of the underlying system. Given the time-bounded nature of most ABSs and the typical focus of validation on the temporal behaviour along individual paths, however, this is uncritical. The time-bounded nature of ABSs also eliminates the need for adjusting the maximum path length dynamically according to the result of the verification of an LTL formula (which can be hard) or, alternatively, the need for monotone LTL formulae [11]. AMC is also interesting from an engineering perspective since, due to the independence of individual path samples, verification can be easily parallelised.

The following section addresses the criteria formulation problem by describing PLTLa, a probabilistic variant of linear temporal logic, and its applicability for the formulation of ABS properties.

5 Towards a Temporal Logic for Agent-Based Simulations

In order to be verifiable by means of AMC, validation criteria need to be formulated in a linear temporal logic. In its basic form, LTL comprises only atomic propositions and temporal and Boolean operators. In order to verify statements like *"eventually the number of infected agents will exceed 100"*, it is thus necessary to introduce a 'helper property', e.g. $numInfGt100$, which is true in all states in which the number of infected agents is greater than 100. The statement can then be expressed by combining the atomic proposition with a temporal operator: $\phi = \mathbf{F}\ numInfGt100$. Although technically correct, this is not a very elegant solution. If we view the output of a simulation as a sequence of states, each of which is itself composed of multiple attribute-value pairs, it would be more convenient if we could refer to particular values in the simulation output and formulate arithmetic and comparison statements within the logic itself. Assuming the existence of a variable $numInfected$ in the output, the property above could then be reformulated as $\phi = \mathbf{F}(numInfected > 100)$ which is much closer to the statement in natural language.

The integration of numeric functions and simple arithmetic expressions into temporal logic is not new and has, for example, been investigated in the context of LTL with constraints (LTLc) [7]. Its usefulness for the verification of complex quantitative and qualitative properties involving external data in the domain of biochemical systems has been shown by Fages and Rizk [8]. We concentrate here on a subset of PLTLc (the probabilistic extension of LTLc) which is restricted to bound variables and thus releases the model checker from having to solve constraint satisfaction problems. We call this logic PLTLa (Probabilistic LTL with arithmetic expressions) and show how it can serve as a basic but powerful formal language for different types of typical ABS validation criteria in the following. The syntax of PLTLa is given below:

$$\phi ::= \text{true} \mid bFunc \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\, \phi \mid val \trianglelefteq val \mid \mathbf{X}\phi \mid \phi\, \mathbf{U}\, \phi \mid \phi\, \mathbf{U}^{\leq k}\, \phi$$
$$val ::= val \oplus val \mid val^{val} \mid nFunc \mid \mathbb{R}$$

Here, $\trianglelefteq \in \{>, <, \geq, \leq, =, \neq\}$, $\oplus \in \{+, -, *, /\}$, val^{val} is the power function, $bFunc$ is a placeholder for a Boolean function $bFunc : S \times 2^{Args} \rightarrow \{\text{true}, \text{false}\}$ which accepts a finite number of arguments drawn from an arbitrary set $Args$ and returns true or false, $nFunc$ is a placeholder for a numeric function $nFunc : S \times 2^{Args} \rightarrow \mathbb{R}$ that accepts a finite set of arguments and returns a real number, and $\mathbb{R}$ is a numeric constant. Note the interpretation of atomic propositions as Boolean functions and number variables as numeric functions, the usefulness of which is described further below. As opposed to LTLc, we assume that all variables are bound, i.e. their values are determined by the underlying model. In terms of satisfaction, the LTL fragment of PLTLa is dealt with in the same way as in basic LTL. For the remaining fragment, let $Eval : S \times val \rightarrow \mathbb{R}$ be an evaluation function which accepts any state $s : S$ and any val as input, evaluates val on s and returns a real number. Then $val \trianglelefteq val$ holds in state s iff $Eval(s, val_1) \trianglelefteq Eval(s, val_2)$ holds. We further define $Last \equiv \neg\, \mathbf{X}\,\text{true}$ (which only holds in the final state of a trace) and $\mathbf{F}^L(\phi) \equiv \mathbf{F}(\phi \vee Last)$. This variant of the $\mathbf{F}$ operator will be important for preserving the semantics of nested temporal operators (see Section 6). For the same reason, we also need to redefine the semantics of the $\mathbf{G}$ operator as follows: $\mathbf{G}\phi \equiv \phi\, \mathbf{U}\, Last$. As described above, we aim to employ a Monte Carlo approach to estimate the likelihood of a formula ϕ, denoted $Pr(\phi)$. In order to refer to the probability of any PLTLa formula ϕ, we use the expression $\mathbb{P}_{\trianglelefteq p}(\phi)$ which is true iff $Pr(\phi) \trianglelefteq p$ is true. We further use $\mathbb{P}_{=?}(\phi)$ to refer to the value of $Pr(\phi)$ itself.

Boolean and numeric functions play a central role in PLTLa. They allow for the integration of custom logic which facilitates the formulation of complex, multi-level properties. An important requirement for an ABS validation framework is the formulation of properties over different levels: we may want to make statements about single agents, about groups of agents or about the whole population. We may also want to make statements about values which are not even part of the simulation output, e.g. aggregations such as '*the average number of healthy agents*'. Instead of manually transforming the output prior to validation, it would be more convenient to specify the mapping of output values to aggregations of interest within the validation framework itself. Functions can facilitate this process by offering a way to hide custom logic behind simple variables. In this way, frequently recurring computations can be integrated without unnecessarily cluttering the actual logical property. This is best illustrated with an example. Imagine an ABS in which all agents have an attribute age which we assume can take values in $[1, 100]$. In order to formulate individual properties, we can now define a numeric function $age : \mathbb{N} \times Att \rightarrow \mathbb{R}$ which accepts the ID of an agent and returns the current value of age. $age(x)$ can then be used just like any other numeric constant within the temporal logic. Let us now look at groups of agents. When dealing with group-level properties, we are mostly interested in some sort of aggregation. We may, for example, want to formulate properties about the *average* number of infected agents, about the *minimum* age of an infected agent or about the *sum* of all agents infected agents at time

t. Again, all of these aggregations can be realised as numeric functions. For example, let $count : Att \times Value \rightarrow \mathbb{R}$ denote a function which accepts the name of an agent attribute $att : Att$ (e.g. age), a possible value $val : [1, 100]$ of the attribute and returns the number of agents for which $att = val$. Like the attribute access function defined above, $count$ can then be used just like any other numeric constant within the PLTLa formula. Boolean functions can be dealt with in the same way except that they are required to return a Boolean instead of a numeric value. The result of a Boolean function can thus be seen as a substitute for an atomic proposition in conventional logic.

The next section describes the formalisation of typical validation criteria in PLTLa.

6 Validation Properties for Agent-based Simulations

An important part of the validation process is the formulation of a set of validation criteria, i.e. a description of the desired behaviours of the simulation. In conventional verification, it is common to classify different correctness questions as *reachability*, *safety* and *liveness properties* (among others which we shall not further discuss here). Since ABSs involve potentially large populations of heterogeneous agents, each equipped with arbitrarily complex internal behaviour, formulating a meaningful and sufficiently exhaustive set of criteria can be hard. It is helpful to use the aforementioned taxonomy as a means to guide and structure the formalisation process. In this section we show how the different property types correspond to typical ABS validation criteria. We further describe how basic questions that involve external (e.g. historical) reference data can also be formulated using the same technical framework. We give examples for each of these properties and show how they can be expressed in a formal way using PLTLa.

Reachability properties: Reachability questions ask whether, starting from an initial state, a particular state s of interest *can eventually be reached*. In a probabilistic environment, this corresponds to the question whether the probability of eventually reaching s is greater than 0. Reachability forms an important basis for the verification of more complex properties as described below. Consider, for example, an ABS simulating the transmission of diseases. Possible reachability questions include: *"what is the probability of reaching a state in which all agents are infected?"* or *"the probability of all agents becoming infected within 10 time steps is less than p"*. The temporal aspect of both properties can easily be expressed using the (bounded) **F** ('finally') operator. In order to formalise the statements in PLTLa, we first need to find a way to extract the number of infected agents from a system state in order for this number to be usable within a PLTLa formula. To this end, we define a function $numInfected$ which returns the number of infected agents. Remember that, since functions are an integral part of PLTLa, $numInfected$ can then be used just like any other numeric constant. Further let $numAgents$ denote a function returning the overall number of agents. We can now define the first property above as $\mathbb{P}_{=?}(\mathbf{F}(numInfected = numAgents))$ and the second property as $\mathbb{P}_{<p}(\mathbf{F}^{\leq 10}(numInfected = numAgents))$.

Safety properties: In addition to checking whether a *desired* state is reachable, we can also check *undesired* states for reachability. This leads to the definition of *unconditioned safety properties* which intuitively state that *"something bad will never happen"*.

Unconditioned safety properties are *invariants* since they must hold for all reachable states in the system[2]. In an epidemiological ABS, we might, for example, expect that a situation in which all agents are infected should never occur. We can define a corresponding safety property using the $\mathbf{G}$ ('always') operator: $\mathbb{P}_{>(1-\epsilon)}(\mathbf{G}(numInfected < numAgents))$. An interesting application for safety properties is to check for correct state transitions. Imagine again an epidemiological ABS in which agents have an attribute $health : \{0, 1, 2\}$[3] and are only allowed to transition between those states in the following order: $S \rightarrow I \rightarrow R$. Assuming numerical function $health(x)$ which returns the value of attribute $health$ for agent x, we can now formulate the following safety property: '*if agent x is in state 0 (Susceptible) then it will never transition directly into state 2 (Infected)*'. In PLTLa, this property can be expressed as follows: $\mathbb{P}_{>(1-\epsilon)}(\mathbf{G}(\neg\ (health(x) = 0 \wedge \mathbf{X}(health(x) = 2)))$[4].

Conditioned safety properties impose restrictions on finite paths and thus cannot be considered invariants [2]. Using an example from a different domain, we could assert that *"the number of sales should never exceed a given threshold* ***unless*** *a marketing campaign has been started"*. This property can be expressed using the $\mathbf{W}$ operator. We first need to define a Boolean function $campaignStarted$ which is true in those states during which the campaign is running. We also need a numeric function $numSales$ which calculates and returns the overall number of sales in a particular state of the simulation. The corresponding PLTLa property is $\mathbb{P}_{\geq(1-\epsilon)}((numSales \leq t)\ \mathbf{W}\ campaignStarted)$.

Liveness properties: Safety properties are always satisfied if the underlying system does nothing at all, which is clearly not a desirable situation. It is thus important to complement them with *liveness properties* which state that *"something good will eventually happen"*. An important subclass are *repeated reachability* or *progress properties* which state that something will *always eventually* happen – a pattern which is often used in validation criteria. In our example, we could, e.g., require that *"it is always possible to return to a state in which at most 10% of the agents are infected"*. This corresponds with the expectation that, whatever happens, the population is always able to recover. In order to formulate this property in PLTLa, we can reuse the functions $numInfected$ and $numAgents$ defined above. Progress properties typically use the nested $\mathbf{GF}$ operator. However, the fact that we are dealing with finite traces has some implications on the semantics of nested temporal operators which, for space limitations, shall not be further described here. In order to ensure that the formal property correctly reflects our requirement, we thus need to use operator $\mathbf{F}^L$ instead of $\mathbf{F}$. This results in the following formula: $\mathbb{P}_{\leq p}(\mathbf{GF}^L(numInfected = (0.1 * numAgents)))$. Progress properties are particularly useful on the individual level. Imagine an ABS in which an agent is expected to perform regular actions, e.g. product purchases. Given a Boolean function $purchase(x)$ which returns $true$ whenever agent x made a purchase in a given tick, we can formalise the progress expectation as follows: $\mathbb{P}_{\geq p}(\mathbf{GF}^L(purchase(x)))$.

[2] Note that, in the presence of finite traces, the duality between invariants and negated reachability properties does not hold.

[3] We assume the following mapping: $0 \rightarrow Susceptible, 1 \rightarrow Infected, 2 \rightarrow Recovered$.

[4] Alternatively, the property can also be expressed as follows: $\mathbb{P}_{=0}(\mathbf{F}(health(x) = 0 \wedge \mathbf{X}(health(x) = 2)))$

Comparing with a reference model: Many important and interesting internal validation questions can be formulated as reachability, liveness or safety properties. An important part of the validation of a simulation, however, is to assess external validity, i.e. how well the simulation output correlates with a given reference model, e.g. historical data. The reference model is often given in the form of *time series data*, i.e. sequences of time-stamped data points. The validation question then becomes to determine the correspondence between the reference model and the simulation output which is typically done using statistical analysis [14].

Fortunately, the same technical framework can be used to formulate properties that involve external data which fully integrates them into the model checking process. In order to exemplify that, let us consider two common metrics for time series comparison: *cumulative (CSE)* and *mean squared error (MSE)*. Let σ denote the simulation output and $\hat{\sigma}$ the reference model. We assume that both are of equal length. The CSE of state i describes the sum of all squared errors up to state i: $CSE(\sigma, \hat{\sigma}, i) = \sum_{i=1}^{n}(\sigma[i] - \sigma[\hat{i}])^2$. The MSE of state i is then calculated as follows: $MSE(\sigma, \hat{\sigma}, i) = 1/i \cdot CSE(\sigma, \hat{\sigma}, i)$. A typical validation property could be: *"The probability of the MSE w.r.t. the number of infected agents of any initial path fragment exceeding threshold t is less than or equal to p"*. Since the MSE can be easily calculated from the CSE, all we need is a numeric function CSE which, given state $\sigma[i]$ as input, calculates and returns the CSE up to this state. We also need a function $tick$ which returns the current time step. There are now several ways to formalise the property, one of which is $\mathbb{P}_{\leq p}(\mathbf{F}(((1/tick) * CSE) > t))$. Alternatively, we could wrap the entire calculation into a numeric function MSE which would then result in the following simplified version of the property: $\mathbb{P}_{\leq p}(\mathbf{F}(MSE > t))$.

7 (MC)^{2}MABS: Monte Carlo Model Checker for ABS

Algorithm 2 Outline of the function SAT

Input: Path σ, PLTLa formula ϕ

1: **if** ϕ is *bFunc* **return** $\{\sigma[i] \mid \phi \models call(\phi)\}$
2: **if** ϕ is $\neg\ \phi_1$ **return** $\{\sigma[i] \mid \sigma[i] \notin \mathtt{SAT}(\sigma, \phi_1)\}$
3: **if** ϕ is $(\phi_1 \wedge \phi_2)$ **return** $\{\sigma[i] \mid \sigma[i] \in \mathtt{SAT}(\sigma, \phi_1) \wedge \sigma[i] \in \mathtt{SAT}(\sigma, \phi_2)$
4: **if** ϕ is $(\phi_1 \vee \phi_2)$ **return** $\{\sigma[i] \mid \sigma[i] \in \mathtt{SAT}(\sigma, \phi_1) \vee \sigma[i] \in \mathtt{SAT}(\sigma, \phi_2)$
5: **if** ϕ is $(\phi_1 \trianglelefteq \phi_2)$ **return** $\{\sigma[i] \mid \sigma[i] \models \mathtt{Eval}(\sigma[i], \phi_1) \trianglelefteq \mathtt{Eval}(\sigma[i], \phi_2)\}$
6: **if** ϕ is $\mathbf{X}\phi_1$ **return** $\{\sigma[i] \mid \sigma[i+1] \in \mathtt{SAT}(\sigma, \phi_1)\}$
7: **if** ϕ is $(\phi_1\ \mathbf{U}\ \phi_2)$ **return** $\{\sigma[i] \mid \sigma[i] \in \mathtt{SAT}(\sigma, \phi_2) \vee (\sigma[i] \in \mathtt{SAT}(\sigma, \phi_1) \wedge \sigma[i] \in \mathtt{SAT}(\sigma, \mathbf{X}(\phi_1\ \mathbf{U}\ \phi_2))\}$

In order to evaluate our ideas, we developed (MC)^{2}MABS, an initial prototype of a Python-based Monte Carlo Model Checker for Multiagent-Based Simulations. It has full support for PLTLa and comprises the following components: (i) a PLTLa parser; (ii) an interface to a path generator; (iii) a labelling mechanism; and, (iv) a model checker. The PLTLa parser was implemented using the `pyparsing` library[5]. Properties can be

[5] http://pyparsing.wikispaces.com

formulated as conventional strings. Any property ϕ is parsed and translated into a tree structure which contains numerical values as well as Boolean and numeric functions as leaf nodes and ϕ as its root node. The interface to the path generator decouples the simulation from the verification process and allows to plug in different path generators. This facilitates the integration of $(\texttt{MC})^2\texttt{MABS}$ into existing simulation environments. It is also possible to use $(\texttt{MC})^2\texttt{MABS}$ retrospectively, for example to analyse a set of paths that have been generated elsewhere. This is particularly useful if the creation of paths is expensive (e.g. due to time-consuming simulation) and cannot be done repeatedly and on demand.

Model checking of a single output path in $(\texttt{MC})^2\texttt{MABS}$ is based on a recursive labelling function `SAT` similar to those used in explicit state model checking [4]. A (simplified) outline of the algorithm is shown in Algorithm 2. Given path σ and PLTLa property ϕ, it performs a depth-first search through the parse tree of ϕ and, for each subformula ϕ_s of ϕ, returns all states that satisfy ϕ_s. Path σ can thus be said to satisfy property ϕ iff its initial state satisfies ϕ, i.e. iff $\sigma[i] \in \texttt{SAT}(\sigma, \phi)$. Function names referenced in PLTLa formulae are expected to exist as callable functions (as indicated by line 1). Arithmetic expressions are evaluated using a special function `Eval` which was briefly mentioned in Section 5 and shall not be further explained here. The probability of ϕ being true in the entire state space is then approximated with the $\mathcal{GAA}$ algorithm proposed by Hérault *et al.* [11] and further described in Section 3.

Similar approximate model checkers have been implemented before, e.g. APMC / PRISM [11, 16], and MC2(PLTLc) [7]. MC2(PLTLc) is closest to $(\texttt{MC})^2\texttt{MABS}$. Our decision to build yet another tool was motivated by the following facts: First, both PRISM and APMC require the specification of the model in the Reactive Modules (RM) language. Despite the power of the language, the translation of an ABS written in a higher-level language like Java or even a domain-specific language like NetLogo [21] can be challenging, especially for modellers with a nontechnical background. RM was not designed as a general purpose language and data structures and helper functions typically found in other languages are thus largely missing. MC2(PLTLc) is more flexible in this respect and allows loading external simulation output. However, output parsing is hard-coded and can thus not be adapted. Second, RM puts a strong emphasis on states and transitions as opposed to the modelling of more complex behaviour which is the focus of higher level languages. Nontrivial conceptual preprocessing of the model prior to the actual implementation, i.e. translation into a state transition model, is thus necessary. Third, all three tools are closed systems, i.e. none of them allows for the integration of custom logic which complicates or renders impossible the formulation of more complex properties such as the MSE. Fourth, as opposed to communication protocols, distributed randomised algorithms or game-theoretic problems (typical application domains of PRISM), ABSs are characterised by a large number of individual constituents (hundreds or thousands). In this context, properties like *"at least x % are in state y"* become crucial. PRISM's labelling mechanism is restricted to expressions formulated over individual components. Because of that, the size of the textual description of such aggregate propositions grows exponentially with the number of agents and the formulation becomes cumbersome and error-prone. Finally, in PRISM each component (agent) progresses individually and simulation of the population update as an atomic step is

thus not possible. As a consequence, properties need to be formulated in a less intuitive way and the path length to be analysed for bounded properties increases linearly with the size of the population. $(\texttt{MC})^2\texttt{MABS}$ is intended to be as customisable as possible by allowing to 'plug in' external logic formulated in a high-level programming language in order to satisfy ABS-specific requirements. Encapsulating complex custom logic into simple Boolean propositions and numeric variables helps to decouple the interpretation of system states from the actual verification logic which increases modularity and reusability of both functions as building blocks and properties as templates in different validation scenarios. As mentioned above, the development of $(\texttt{MC})^2\texttt{MABS}$ is still at a very early stage. Nevertheless, first experiments have shown promising results. As we shall describe below, we were able to verify properties for ABSs of considerable size efficiently.

8 Experiments

Algorithm 3 Outline of the agent update function

```
p := number of infected neighbours
n := total number of neighbours
if state = Susceptible then
    move to state Infected with probability p/n
    remain in state Susceptible with probability 1 − (p/n)
else {state = Infected}
    move to state Recovered with probability 0.3
    remain in state Infected with probability 0.7
else {state = Recovered}
    move to state Susceptible with probability 0.5
    remain in state Recovered with probability 0.5
end if
```

In order to assess the applicability of AMC to ABS validation, we designed two experiments. They were conducted on an Amazon EC2 [1] 64-bit instance with one virtual core comprising two EC2 computing units[6], 3.75 GB of memory and Ubuntu 12.10 Server as operating system.

The first experiment involved the comparison between AMC and conventional symbolic model checking. To this end, we wrote a simple (entirely unrealistic) epidemiological model in Python (see Algorithm 3). All agents are connected to each other and each agent can be in one of three states: $Susceptible$, $Infected$ and $Recovered$. Transitions between states are probabilistic and partly dependent upon the number of infected neighbours. We also translated this model into PRISM's RM language. Due to the characteristics of RM discussed in Section 4 and the resulting complications in translation, we restricted our comparison to the verification of a simple reachability property ($\mathbf{F}\,allInfected$) which states that eventually all agents will be in state *Infected*. We executed the Python model 1,000 times and checked the property on each path

[6] At the time of writing, each computing unit is equivalent to a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor.

	$(\texttt{MC})^2\texttt{MABS}$	PRISM		
#agents	Time	#states	#trans.	Time
10	0.08	$3.94 \cdot 10^4$	$2.93 \cdot 10^5$	0.13
15	0.09	$2.14 \cdot 10^6$	$1.76 \cdot 10^7$	7.80
20	0.09	$3.00 \cdot 10^8$	$3.23 \cdot 10^9$	o.o.m.
50	0.09	out of memory		
100	0.11	out of memory		

	Property									
	P1		**P2**		**P3**		**P4**		**P5**	
	#ticks		#ticks		#ticks		#ticks		#ticks	
#agents	50	100	50	100	50	100	50	100	50	100
100	0.09	0.17	0.10	0.23	0.65	1.33	0.4	0.78	3.7	13.72
500	0.09	0.17	0.09	0.17	0.64	1.25	0.4	0.76	3.73	13.67
1000	0.09	0.17	0.09	0.17	0.64	1.25	0.39	0.76	3.73	13.63

Table 1. Runtime comparison between $(\texttt{MC})^2\texttt{MABS}$ and PRISM's symbolic model checker (left) and time spent for the verification of NetLogo's Virus on a Network model (right). All times are given in seconds.

which results in $\delta = 0.01$ confidence at an approximation of $\epsilon \approx 0.05$. The population size was varied from 10 to 100. The results (see Table 1 left) indicate that $(\texttt{MC})^2\texttt{MABS}$ outperforms PRISM's symbolic model checker for which the exponential growth of the underlying state space soon became a limiting bottleneck. All checks were performed 100 times. The coefficient of variation lies between 0.8% and 3%, suggesting little variance in runtime.

For the second experiment, we focussed on a more realistic and significantly larger example and used $(\texttt{MC})^2\texttt{MABS}$ to verify properties of *Virus on a Network* [19], an epidemiological model from the NetLogo [21] model library. Using the BehaviourSpace feature which allows simulations to be run repeatedly, we created 1,000 sample paths. The size of the population was varied from 100 to 1,000. In each experiment, we obtained the approximate probability of the simple reachability property from above (referred to as **P1**) and the following properties:

P2: ($\mathbf{F}(numInfected = (numAgents * 0.3))$): *"Eventually 30% of the agents will be infected"*
P3: ($\mathbf{FG}(numInfected \leq (numAgents * 0.01))$): *"The population will always recover*[7]*"*
P4: ($\mathbf{G}(sqError \leq x)$): *"The SE btwn. the simulation and a ref. model is always $< x$"*
P5: ($meanSqError \leq x$): *"The MSE btwn. the simulation and a reference model is less than x"*

The results (see Table 1 right) indicate that properties of even large-scale models can be verified efficiently. Similar to the first experiment, each check was performed 100 times. The coefficient of variation lies between 0.6% and 2.2%, indicating little variance in runtime. The influence of the population size on the verification time is negligible which suggests good scalability. On the other hand, the path length has a significant impact. This is both due to the labelling process and to additional computation necessary, e.g. for **P5**. It needs to be taken into account, however, that $(\texttt{MC})^2\texttt{MABS}$ is currently entirely unoptimised. We are confident that the numbers can be improved significantly by using more efficient data structures, by avoiding unnecessary looping in the labelling process[8] or by examining paths in parallel.

9 Conclusions and Future Work

In this paper we discussed the usage of approximate probabilistic temporal logic model checking for large-scale ABS validation. We described how PLTLa, an extension of

[7] By 'recovering' we mean returning to a state in which at most 1% are infected.
[8] For example, a path could be labelled with multiple atomic propositions in a single iteration.

LTL that allows for the formulation of arithmetic expressions and the integration of external logic by means of numeric and Boolean functions, can be used to encode common internal and external multi-level validation criteria in a formal and rigorous way. By interpreting atomic propositions and numeric values as function calls, even complex properties that involve aggregation over individual agents can be formulated in a succinct way and verified automatically. We further presented our initial version of $(\texttt{MC})^2\texttt{MABS}$, an approximate probabilistic Monte Carlo model checker tailored to the verification of ABS validation criteria. Since $(\texttt{MC})^2\texttt{MABS}$ concentrates on the simulation output and treats the simulation itself as a blackbox, it can be integrated into existing simulation environments without much effort. Given the interdisciplinary nature of ABS which involves people from various domains often with non-technical backgrounds, the encapsulation of the technical verification process represents a critical advantage for its practical adoption. Preliminary experiments showed promising results. Because of its customisability with respect to both expressiveness and accuracy, $(\texttt{MC})^2\texttt{MABS}$ can be tailored to the characteristics of different types of ABSs and used for the verification of arbitrarily large systems.

There are a number of open problems that we are currently working on. Due to its focus on finite path fragments, the complexity in AMC is shifted from model construction and verification to the creation of a sufficiently large number of path samples in order to guarantee a sufficient level of accuracy. Against the background of large-scale real world systems, this aspect may represent a critical bottleneck. Depending on the complexity of the original simulation, a single run may take a long time, which renders the generation of thousands of traces impossible. A common idea followed by most existing verification tools is to offer a high-level description language in which the system logic can be formulated. The high-level description can then be translated automatically into a highly performant 'path generator' which is able to sample a large number of paths efficiently. Similar to simulation itself, this process is also easily parallelisable and can thus be engineered efficiently. We are currently working on a more expressive temporal logic in which ABS-specific features like aggregation, selection and quantification over groups of agents can be expressed more naturally within the logic itself. Finally, we are investigating the possibility of adding 'on-the-fly' verification capabilities to $(\texttt{MC})^2\texttt{MABS}$ which would allow it to run as a monitoring process in parallel to the simulation. In this way, violations of validation criteria could be detected immediately when they happen and cause the simulation to stop. Given the often significant running time, this has the potential to reduce the time needed for verification significantly.

References

1. Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/.
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. E. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

5. 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.
6. E. W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.
7. R. Donaldson and N. Gilbert. A Monte Carlo model checker for probabilistic LTL with numerical constraints. Technical Report 282, Dept. of Computing Science, University of Glasgow, 2008.
8. F. Fages and A. Rizk. On the analysis of numerical data time series in temporal logic. In *Proc. Int. Conf. on Computational Methods in Systems Biology*, pages 48–63. Springer, 2007.
9. H. Fecher, M. Leucker, and V. Wolf. Don't know in probabilistic systems. In *Model checking software*, pages 71–88. Springer, 2006.
10. J. M. Galán, L. R. Izquierdo, S. S. Izquierdo, J. I. Santos, R. del Olmo, A. López-Paredes, and B. Edmonds. Errors and artefacts in agent-based modelling. *Journal of Artificial Societies and Social Simulation*, 12(1):1, 2009.
11. T. Hérault, R. Lassaigne, F. Magniette, and S. Peyronnet. Approximate probabilistic model checking. In *Proc. 5th Int. Conf. on Verification, Model Checking and Abstract Interpretation*, volume 2937 of *LNCS*, pages 307–329. Springer, 2004.
12. G. Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, first edition, 2003.
13. L. R. Izquierdo, S. S. Izquierdo, J. M. Galán, and J. I. Santos. Techniques to understand computer simulations: Markov chain analysis. *JASSS*, 12(1):6, 2009.
14. J. P. C. Kleijnen. Validation of models: statistical techniques and data availability. In *Proc. 31st Winter Simulation Conf.*, pages 647–654. ACM, 1999.
15. 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.
16. M. Kwiatkowska, G. Norman, and D. Parker. Stochastic model checking. In *Proc. 7th Int. Conf. on Formal Methods for Performance Evaluation*, SFM'07, pages 220–270. Springer, 2007.
17. D. Midgley, R. E. Marks, and D. Kunchamwar. Building and assurance of agent-based models: An example and challenge to the field. *Journal of Business Research*, 60(8):884 – 893, 2007. Complexities in Markets Special Issue.
18. R. G. Sargent. Verification and validation of simulation models. In *Proc. 40th Conf. on Winter Simulation*, WSC '08, pages 157–169. Winter Simulation Conference, 2008.
19. F. Stonedahl and U. Wilensky. NetLogo Virus on a Network model. Technical report, Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL, 2008.
20. 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.
21. U. Wilensky. NetLogo. Technical report, Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL., 1999.