Check for updates

Simulation

Simulation: Transactions of the Society for Modeling and Simulation International
89(8) 977–1008
© 2013 The Society for Modeling and Simulation International
DOI: 10.1177/0037549713490439
sim.sagepub.com
SAGE SCS

On-the-fly verification of discrete event simulations by means of simulation purposes: Extended version

Paulo Salem da Silva and Ana C V de Melo

Abstract

Discrete event simulations can be used to analyze natural and artificial phenomena. To this end, one provides models whose behaviors are characterized by discrete events in a discrete timeline. By running such a simulation, one can then observe its properties. This suggests the possibility of applying on-the-fly verification procedures during simulations. In this work we propose a method by which this can be accomplished. It consists of modeling the simulation as a transition system (implicitly), and the property to be verified as another transition system (explicitly). The latter we call a *simulation purpose* and it is used both to verify the success of the property and to guide the simulation. Algorithmically, this corresponds to building a synchronous product of these two transitions systems on-the-fly and using it to operate a simulator. By the end of such an algorithm, it may deliver either a conclusive or inconclusive verdict. If conclusive, it becomes known whether the simulation model satisfies the simulation purpose. If inconclusive, it is possible to adjust certain parameters and try again. The precise nature of simulation purposes, as well as the corresponding satisfiability relations and verification algorithms, are largely determined by methodological considerations important for the analysis of simulations, whose computational characteristics we compare with empirical scientific procedures. We provide a number of ways in which such a satisfiability relation can be defined formally, the related algorithms, and mathematical proofs of soundness, completeness and complexities. Two application examples are given to illustrate the approach.

Keywords

Runtime verification, systematic exploration, formal method, testing, transition systems

1. Introduction

Discrete event simulations can be used to analyze natural and artificial phenomena. To this end, one provides models whose behaviors are characterized by discrete events in a discrete timeline. By running such a simulation, one can then observe its properties.[i] Our particular interest is in the simulation of multi-agent systems (MASs), in which the simulation model is a set of *agents* who interact within an *environment*.[2] A number of tools exist for performing such simulations,[3,4] and the need for methods for analyzing them has already been recognized.[5,6] However, compilation of statistics over multiple simulation runs is often the only implemented analysis mechanism, while formal verification is seldom considered.

Much more can be done in this respect. The fundamental insight is that simulations, by their very nature, ought to run and produce traces. It is very natural then to verify whether the traces being produced obey some temporal property. Runtime verification is particularly suitable for this task, for, as we shall explain, it allows not only the analysis of simulation runs, but also the on-the-fly choice of which runs are more pertinent to the property being considered. Thus, while full formal verification of simulation models is still a difficult problem, at least approximate formal verification can be performed. Let us expand this insight and shed further light on the rationale for incorporating formal verification ideas in simulation analysis.

University of São Paulo, Department of Computer Science, São Paulo, SP, Brazil

Corresponding author:
Paulo Salem da Silva, University of São Paulo, Department of Computer Science, São Paulo, Brazil.
Email: salem@ime.usp.br

Ana CV de Melo
Email: acvm@ime.usp.br

1.1. The case for combining formal verification and simulation

Formal verification and simulation methods are often seen as antagonists. The former answer questions with certainty either through formal proofs or by examining some model exhaustively, at the cost of either human effort (e.g. in writing proofs) or computational resources (e.g. in exploring exhaustively exponentially large state-space). The latter answers questions only in a partial and approximate manner, but at a much lower cost in terms of both human effort and computational resources. Moreover, formal verification is often only feasible in relatively small or abstract models (owing to the well-known *state-explosion problem*), whereas simulation methods can be applied to any system that can run at all, since it is by observing executions of the system (or a detailed model of it) that a simulation technique proceeds.

It is fair to say, however, that both approaches are seen as ways to automate the analysis of systems. The effort spent in building formal models or simulation models is worth because, once built, they allow one to investigate various properties of interest with the aid of a computer, instead of doing so manually. Therefore, they are not very different in their fundamental purposes, but only in their methods. With this in mind, let us sketch the fundamental elements of these approaches to see why and how they can be combined.

As stated above, both approaches aim at modeling and analyzing some system, call it M. Given such an M, the following are the main kinds of problems one may be interested in.

(i) Does M satisfy a property P?
(ii) What kinds of behaviors can be observed during M's execution? Does it conform with our expectations?
(iii) What inputs should be given to M to make interesting observations?
(iv) What inputs should be given to M in order to optimize some of its output value?

The first kind is strongly associated with formal verification, whereas the others most often belong to simulation. Indeed, formal verification has its roots on mathematical logic, in which one is traditionally interested in the truth value of *predicates* (such as a logic formula P) with respect to some object (such as a formal specification M). In this article, the particular kind of formal verification that concerns us is model checking,[7] where P is given as a formula of some *temporal* logic and M is a transition system whose states are labeled by atomic propositions. Each path in M is a possible evolution of the system. The objective is to check whether P holds considering all of these possible evolutions. This is achieved in various manners by different model checking techniques, but essentially they all systematically explore M having P as a guide.

As we said earlier, simulations necessarily produce traces of their execution, and thus one can easily imagine a transition system defined by all possible executions of some simulation model. In this manner, one can obtain an M similar to that which is used in model checking. What about P, can it be obtained from the problems relevant to the analysis of a simulation model? Let us examine the remaining items from the list of problems above:

(ii) Our expectations about the behavior of M can be characterized by making assertions about the value of variables in the several steps of a simulation. Thus, one can formally specify it through some formal property P. This matches closely the way formal verification works.
(iii) Part of what makes an observation interesting might be known *a priori*. For example, we may be concerned with what happens under a set of particular circumstances. This can be formally specified through some P by defining preconditions that determine which states in M are (potentially) interesting and which are not (and thus can be safely discarded from further simulations). This can be coupled with an algorithm to generate candidate inputs, which are then subject to simulation and whose traces can be examined by the user after the irrelevant (with respect to P) traces are removed.
(iv) The optimization of some variable can be defined by simple mathematical assertions, thereby easily providing a P to be used when assessing the output of a simulation. The actual optimization can be carried out by generating a set of candidate inputs, simulating all of them and finally selecting the inputs which produced the best output with respect to P.

Therefore, one can always obtain a P which helps in the job of assessing a simulation model. From this discussion, it is clear that one can theoretically see the exploration of a simulation model under a formal verification perspective. It remains to ask, though, whether it is actually worth doing so.

The answer which the present article supports is that the usefulness of this depends on how much information P carries. The more informative P can be made, the more useful it is to use such a hybrid approach. This arises from the fact that an informative P can be used to algorithmically discard many irrelevant simulations, thereby focusing the attention of the user on those simulations which are potentially interesting. In other words, instead of relying on a human operator to manually define inputs and run

a simulation, by adopting such a hybrid approach one can capture the operator's *a priori* knowledge and expectations (through P) and automate the whole process. The objective is not to *exhaustively* explore M (as it would be in a pure formal verification method), but merely to *systematically* and *automatically* explore M so that only promising simulations are performed. Hence, it is a way to address the already recognized problems of automating[5,6] and providing standard tools[2] to the exploration and analysis of simulations.

The use of formal verification in more empirically oriented areas is not a possibility unique to simulation. Software testing has seen considerable progress with the incorporation of ideas from formal verification, which resulted in the area of model-based testing.[8,9] The fact that such combinations are possible elsewhere suggests that there might be gains in applying them in the context of simulations. Indeed, as we explain later, the approach proposed here is inspired by a technique from model-based testing.

What was said above is a general account of the reasons for incorporating formal verification ideas, in particular those from model checking, in the analysis of simulation models. Our particular technique does not, of course, cover all possibilities that arise from the discussion. For example, we do not address optimization problems. However, we do offer a comprehensive method to address satisfiability problems, that is to say, to check whether, and how, our expectations about a simulation model hold or not. Our P is given as a *simulation purpose*, a structure that is used both to specify the property of interest and to guide the simulation executions, so that only the relevant simulations are performed. We hope that our particular application of the above principles serve not only as a direct technical contribution, but also as a compelling argument in favor of those general principles themselves.

1.2. Contribution of this article

This work presents a method that uses transitions systems in order to perform simulations orderly and verify properties about them. To this end, the possible simulation paths are described as a transition system (an *annotated transition system* (ATS), as we call it), and the property to be verified as another. Moreover, the property has some further particularities that make it a special kind of transition system, which we call a *simulation purpose*. Such simulation purposes not only give criteria for satisfiability but are also employed to guide the simulation, so that states irrelevant for the property are not explored. The verification is achieved by building, on-the-fly, a special kind of synchronous product between these two transition systems.

The ATS to be analyzed is given implicitly by the simulator. That is to say, at each simulation step the simulator provides choices concerning the next possible simulation events to take place, thus incrementally describing a transition system. Clearly, this on-the-fly construction assumes the existence of a simulator with a certain interface. However, this interface is very simple, and therefore can be easily incorporated into existing simulators. Furthermore, although we developed this method with the verification of MASs in mind, the resulting technique is actually applicable to a much larger class of discrete event simulations: those that satisfy the said simulator interface, which is tied to the notion of events, not agents.

Simulation purposes, in turn, are given explicitly by the user. They describe the desirable and undesirable simulation runs. Mathematically, they are used to select the relevant part of the ATS being analyzed, which results in their synchronous product. By investigating the structure of this product one can determine whether the simulation satisfies the simulation purpose in a number of precise senses. In this article, our focus is on four such satisfiability relations. On the one hand, *feasibility* (*refutability*) consists of checking whether some particular desirable (undesirable) path on the simulation purpose can actually be simulated. On the other hand, *certainty* (*impossibility*) requires that all desirable (undesirable) paths in the simulation purpose can actually be simulated.

Algorithmically, the verification of these four relations are all very similar, and it consists, essentially, in performing a depth-first search in the synchronous product of the two transition systems. In the case of the first two relations, this search seeks a run to show that the property holds, whereas in the last two it seeks a run to show that the property does not hold. Because it is a depth-first search, memory consumption is linear with respect to the search depth. Running time, however, in the worst case can be exponential with respect to the depth. While this difficulty exists, the fact that a part of the relevant simulations can be automatically and systematically chosen, run and judged is already valuable, and provides an approximate way of exploring them.

The present article is a substantially extended version of our initial presentation in da Silva and de Melo.[10] In particular, here we provide more detailed definitions, more satisfiability relations and related algorithms, and full mathematical proofs of soundness, completeness and complexity. This is the result of a PhD thesis.[11]

The text is organized as follows. Section 2 comments on the works that inspired our approach. Section 3 defines the objectives of our verification technique by showing how a systematic exploration of simulations can be thought of as the performance of scientific experiments. Section 4 presents the general form of our ATSs, while Section 5 defines simulation purposes. The whole technique is based on the synchronous product of these two entities, which is thus introduced in Section 6. There are more than one way in which an ATS can satisfy a

simulation purpose, and these several manners are considered in Section 7. Section 8, then, provides the verification algorithms themselves, along with an informal explanation (mathematical proofs of correctness and complexity are provided in Appendix A). Section 9 gives a concrete example of how the approach can be useful. Finally, Section 10 concludes.

2. Related work

The inspiration for this work comes from the area of model-based testing,[8] specially from the approach used by TGV[12] to generate test cases from specifications. There, a system under test is represented as a transition system, and a formal test purpose is used to extract test cases that satisfy the *ioco* conformance relation.[9] These test cases, then, can be executed against an actual implementation of the specification. TGV itself is based on a more general approach to the on-the-fly verification of transition systems, which can also be used to perform model-checking and to detect bisimulation equivalences.[13]

Our approach differentiates itself fundamentally from TGV because our objective is not the generation of test cases, and in particular we are not tied to the *ioco* conformance relation. Indeed, our simulation purpose is itself the structure that shall determine success or failure of a verification procedure (i.e. not some *a posteriori* test cases). As a consequence, different criteria of success or failure can be given, and then computed on-the-fly. In this article we consider the case in which one computed path terminates in a desirable state (which we call a *success* state), but we could also have the stronger criterion that requires all paths to terminate in such a desirable state. As we shall see in Section 3, a number of particular methodological considerations are at the heart of these definitions. Moreover, there are also other technical differences, such as the fact that we use labeled states (and not only transitions), and that simulation purposes need not be input-complete.

As we pointed out in the introduction, formal verification is not typically used together with simulation. There are, however, a few exceptions. Bosse et al.[14] present the Temporal Trace Language (TTL), which has an associated tool, designed to define simulation models (in a sublanguage called LEADSTO), as well as linear-time properties about such models. The approach is to execute the simulation model and check whether the resulting traces obey the specified linear-time properties. An example of this method is given by Bosse and Gerritsen,[15] where criminal behavior is modeled, simulated and analyzed. Our proposed method contrasts with this mainly in two aspects. First, only part of the input we require is formal (part is given as black-boxes to be simulated, which can be implemented in any programming language), whereas the method by Bosse et al.[14] depends on complete formal specifications, even though they are meant for simulation. Second, in the work of Bosse et al.[14] the properties to be checked play a passive role and are only considered *after* simulations, whereas the properties we check (simulation purposes) are also used to *guide* the simulations so that only relevant simulations are performed.

Two other similar exceptions are the network simulator Verisim[16] and a multi-agent modeling of food poisoning.[17] The approach taken by these works consists of running the simulation normally, but checking linear-time properties in the resulting execution traces. This kind of approach can be implemented by the runtime verification notion of a *monitor*, which is an extra component that is added to the system in order to perform the verification. Verisim,[16] for instance, employs the MaC architecture[18] to provide such a monitor for its simulations. The precise nature of monitors vary according to the kind of property to be analyzed. But it turns out that linear-time properties are more suitable to this task, and thus most approaches employ some variation of a linear-time logic, such as linear-temporal logic (LTL).

However, the traditional semantics for LTL assumes an infinite execution trace. Thus, it is unable to cope with cases in which only finite traces are available. To solve this problem, one may modify LTL to account for the case in which traces are finite entities. This approach is followed in a number of works.[19–21]

Our simulation purposes also express linear-time properties. However, they have particularities that are better defined in the form of transition systems instead of logic formulas. For example, the notion of events and state propositions are different, and explicitly so. The possibility to express both success and failure in the same structure is another important point. Other requirements are examined in Section 3.

Our technique, though quite different, nevertheless has common characteristics with model checking.[7] Most importantly, both assume the existence of an explicit set of states and transitions to be analyzed. In model checking this set is examined exhaustively, so that a conclusive verdict can always be given, provided that there are enough computational resources. In our case, by contrast, only a small part of the state-space is explored (i.e. those that are reached by the simulations performed), and one can never be sure of having explored every possible state, since the simulator is a black-box. Moreover, both methods allow the specification of a property of interest to be analyzed with respect to a system, thus establishing a difference between the model and the properties of the model. In model checking such a property is typically given in terms of some temporal logic, such as LTL. In our approach we use simulation purposes instead.

Despite operating on explicit models of systems (i.e. all possible transitions and states are considered), model checking remains a technique that manipulates purely

abstract, formal, structures. This allows, in particular, algorithms that do not check concrete executions, but abstract ones. This can be achieved with the technique of abstract interpretation,[22] through which one considers a super-set of all possible behaviors by ignoring selected details in the specification (e.g. instead of considering all particular integers for a variable x, only consider three *abstract* cases: $x < 0$, $x = 0$ and $x > 0$). However, it is not clear how such a technique could be applied in the verification of simulations as described in this article. While the formal descriptions employed could, in principle, be made more abstract, part of our method depends on black-boxes with interfaces whose internal details are not accessible. Hence, it would not be possible to really abstract over concrete executions, since at least these black-boxes must be simulated by concrete means.

Another technique used in model checking to reduce the state-space to be explored is partial order reduction.[23,24] It consists of using the fact that certain events are commutative and, therefore, that certain orderings are equivalent. With this realization, one then proceeds to examine only a representative execution among equivalent executions. In principle, it is possible to apply this technique to the method proposed in this article. Certain events are completely under control of the verification algorithm, which can thus select the order in which they happen. By allowing the user to add a specification of which events are commutative, one could then augment our algorithms to consider only one order among the equivalent orders. We have not pursued this idea, but it remains an interesting topic for further research.

It is also worth mentioning a particular model checking technique that limits itself to finite executions, called bounded model checking.[25–27] By limiting the length of the executions that are examined, it is possible to perform an efficient translation of the resulting problem to an instance of the general SAT problem. Profiting from recent developments in SAT solvers, this provides an efficient, although not complete, model checking approach. The state explosion problem, however, remains: the higher the bound, the worse it gets. In this manner, though also considering finite executions, the technique we propose here is fundamentally different. For example, the complexity to consider very long execution lengths, but only few of them, to us is polynomial, whereas to bounded model checking this would remain exponential (because it considers all executions of the given length, and not just a few).

Schruben[28] points out the simulation modeling and analysis are often seen as two entirely different activities, and argues that it would be more productive to design models considering how they are supposed to be analyzed. Our work, then, can be seen under this light, since our models and their analyses are closely related.

The Discrete Event System Specification (DEVS)[29] family of simulation formalisms provides conceptual frameworks to put simulation under rigorous definitions. In particular, DEVS defines the notion of *experimental frame* as an entity which provides inputs to a simulation model and judges its outputs. For the sake of uniformity, experimental frames can be expressed with the same formalism used to specify the simulation model itself. Experiments run in this fashion, though, have no control over the simulation once it is started, and can only evaluate its final result. This is sufficient to devise certain optimization techniques, by which several input parameters are tested in order to find those that generate the best output according to some optimization criteria.[30] This contrasts with our approach, since our technique allows simulations to be guided on-the-fly, and the property of interest is given declaratively (in the form of a simulation purpose), and not programmatically, as is the case with experimental frames.

Even though a DEVS model is meant for simulation, it can sometimes be subject to formal verification through model checking, provided that the model can be reduced to a particular subset of DEVS such as FD-DEVS.[31] Note, however, that this is not a verification of the simulation runs, but of the formal model itself, which is not our goal in this work.

In our verification algorithms we shall need a preprocessing procedure to calculate shortest distances from a certain vertex in the graph induced by the specified simulation purpose. To this end, we could use Dijkstra's algorithm.[32] However, the edges in our graph all have weight one (i.e. we count hops between a vertex and its successors), which permitted the development of a more specific algorithm. The reason is that, in this case, it is not necessary to keep a priority queue with unexplored vertices (ordered according to their current distances), which needs to be regularly re-ordered to account for updated entries. It suffices to explore the vertices in a depth-first manner.

3. Automation of experiments

The formal approach that we present in the following sections is justified by the objective to which they should be applied, namely, the automated analysis of simulation models. Hence, in this section we examine the general nature of these models, and what kinds of questions are relevant for them.

Programs are usually designed in order to accomplish something. That is to say, they are supposed to obey a specification (even if the specification exists only in the mind of the programmer). If they indeed do so, they are deemed correct. Otherwise, they are considered incorrect. Verification, and formal verification in particular (where one has *formal* specifications of the expected behavior), is

thus concerned with determining whether or not programs satisfy specifications, from which one may infer the correctness of the program.

Yet one may have a slightly different point of view on the matter. In our case, we use programs as simulation models. From our perspective of modelers, the model is not necessarily supposed to accomplish something, for it is merely a representation of a certain state of affairs, which may be outside of our control. In investigating it, we are thus not necessarily concerned with whether it is doing its job correctly. Indeed, we may very well ignore why the model was designed in the way it was. We just want to discover *what* it can or cannot do. To this end, we may also employ a specification. But it is the specification of a *hypothesis* to be investigated, and which can be either true or false. Note the crucial difference: when verifying a program, the fact that the specification was violated indicates a problem in the program, and thus it is always undesirable; however, in our case, the fact that the hypothesis is violated is not, in principle, an indication of a problem either in the model or in the hypothesis itself. The judgment to be made depends of our objectives in each particular circumstance. Are we trying to discover some law about the model? In this case, if a hypothesis that represents this law turns out to be false, it is the hypothesis that is incorrect, not the model. Are we trying to engineer a model that obeys some law? In this case we have the opposite, a falsified hypothesis indicates a problem in the model. This view is akin to that found in empirical sciences, in which scientists investigate hypothesis and make judgments in a similar manner. In this respect, the main difference is that the empirical scientist studies the natural world *directly*, while we are concerned with *models* of nature.

More specifically, we are interested in simulation models of MASs. That is to say, those systems that can be decomposed into a set of *agents* and an *environment* in which these agents exist. Often, the description of environments is much simpler than that of the agents. When this is the case, we can give a formal model for the environment and treat the agents therein as black-boxes.[33]

In this article we employ two different examples that profit from this perspective. The first, very simple, models only one agent, a dog, whose training we want to assess through an environment which allows us to stimulate it in various ways. As it is well known, dogs can be trained to perform a number of things, and this example models some of them. This first example is constructed along the text, and is used to illustrate, in a simple and concrete manner, the several concepts that form our approach as we introduce them.

The second example is somewhat more complex and serves to shed further light on the approach afterwards, as well as to clarify details that might not have been fully appreciated before. It consists of a model of an online social network, where several persons exist and can interact with each other through the features of a website.[ii] By means of such a model, the operator of the website can experiment with changes using simulations before applying them to the real website. We confine the construction and analysis of this example to Section 9.

In both examples, the behavior of each individual agent is likely to be complex, and if a model is given to them, it probably will not be a simple one.[34] But the environment, on the other hand, can be described by some formalism that merely define what operations can be performed on agents, as well as relations among agents (e.g. using a process algebra such as the π-calculus[1]), providing a much more tractable model. The purely formal manipulations, then, can be restricted to the environment model. An overview of such an architecture is given in Figure 1.

Note that this is analogous to an experimental scientist working in his laboratory. The scientist is usually interested in discovering the properties of some agents, such as animals, chemicals, or elementary particles. He has no control over the internal mechanism of these agents: that is why experiments are needed. But he can control everything around them, so that they can be subject to conditions suitable for their study. These scientific experiments have some important characteristics:

- *inputs* should be given to agents under experimentation;
- *outputs* should be collected from these agents;
- sometimes it is not possible to make some important measurement, and therefore experiments often yield *incomplete knowledge*;
- the experiment is designed to either confirm some expectation (a *success*) or refute it (a *failure*);
- the experiment should last a *finite amount of time*, since the life of the scientist is also finite;
- the experiment should be as *systematic* as possible, though exhaustiveness is not required; the important thing is to try as many *relevant scenarios* as possible; in particular, the scientist may control how to continue the experiment depending on how the agents react;
- the experiment should define a clear course of action from the start;
- the experiment can be a way to find out how to achieve a certain end, after trying many things; therefore, it must be performed in a *constructive* manner, and not merely by deriving a contradiction;
- absence of a success does not necessarily mean that there is no way to achieve a desired effect; hence, it is convenient to know when something clearly indicates a failure.

A simulation purpose is like such a scientist: it controls the direction of the simulation and determines whether

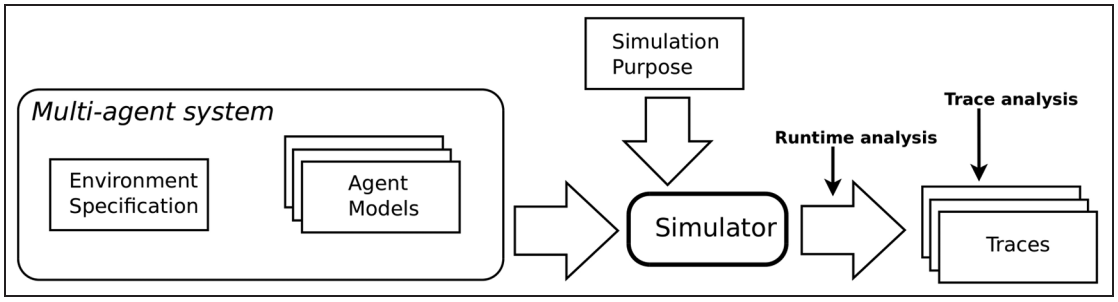

Figure 1. Overview of the verification architecture based on simulations of MASs. The simulator takes two inputs: (i) a MAS, composed by agent models and an environment specification; (ii) a *simulation purpose* to be verified. The simulator then produces traces as outputs. Verification can be done at the simulation runtime level, as well as at the trace level (if the traces are recorded). In this article, we only consider runtime verification, because this allows the simulations to be controlled in such a way that only those relevant to the specified simulation purpose are actually performed.

something constitutes a success or a failure by following similar principles. An environment model, in turn, is similar to the experimental setup, with its several instruments and agents. The simulation purpose interacts with the environment model in order to achieve its aims. All of this is accomplished by considering these two artifacts as transition systems.

In this way, the technique we present in this article should be seen as a method to automate experiments undertaken employing simulations of either natural or artificial phenomena. As Figure 2 shows, it depends on two main interacting layers, namely, one of verification and one of simulation. The details of the simulation layer are domain-specific (in our case, we approached the problem from a MAS perspective), but the verification method is general and applicable to any discrete event simulation. This article focus on the verification part and uses the MAS simulation merely as an application example.

4. ATSs

While there may be many ways to specify the systems and their properties (e.g. programming languages, process algebras, logic), it is convenient to have a simple and canonical representation to serve as their common underlying semantic model. Here, we employ *ATSs* to this end, which are nothing but transition systems with labels given to both states and transitions.[iii]

In an ATS, events play a central role, and are further divided into *input events* and *output events*. The former represent events that may be controlled by the verification procedure (i.e. may be given as an input to the simulator), and the latter events that cannot (e.g. because they are the output of some internal, and uncontrollable, behavior of the simulator). The following definition formalizes this.

Definition 1 (Events). *Let $\mathcal{N}$ be a primitive set of names. An* event *is one of the following:*

- *an* input event, *denoted by $?n$ for some $n \in \mathcal{N}$;*
- *an* output event, *denoted by $!n$ for some $n \in \mathcal{N}$;*
- *the* internal event, *denoted by τ, such that $\tau \notin \mathcal{N}$;*
- *the* other event, *denoted by $\circledast$, such that $\circledast \notin \mathcal{N}$.*

The $\circledast$ event above is a convenience to allow the specification of the complementary set of events possible in any state. A notion of event complementarity is also useful for later definitions.

Definition 2 (Complementary event). *Let $\mathcal{N}$ be a primitive set of names and e an event. Then its* complementary event, *denoted by e^c, is defined as*

$$e^C = \begin{cases} !n & \text{if } e = ?n \\ ?n & \text{if } e = !n \\ \tau & \text{if } e = \tau \\ \circledast & \text{if } e = \circledast \end{cases}$$

Finally, we may define an ATS.

Definition 3 (Annotated transition system). *An* ATS *is a tuple $\langle S, E, P, \rightarrow, L, s_0 \rangle$ such that:*

- *S is the set of primitive states;*
- *E is the finite set of events;*
- *P is the finite set of primitive* propositions;
- *$\rightarrow$: $S \times \mathrm{E} \times S$ is the* transition relation;
- *for any $s \in S$ and $e \in E$, there are only finitely many $s' \in S$ such that $s \xrightarrow{e} s'$ (i.e. finite branching);*
- *$L : S \mapsto \mathbb{P}(P \cup \neg P)$ is the* labeling function[iv];
- *for all $s \in S$ and all $p \in P$, if $p \in L(s)$, then $\neg p \notin L(s)$ (i.e. the labeling function is consistent);*
- *$s_0 \in S$ is the* initial state.

Note that the labeling function associates literals (for any proposition p, its associate literal l is defined either by $l = p$ or $l = \neg p$; in the former case, we say it is a *positive literal*, whereas in the latter we say it is a *negative literal*), and not merely propositions, to the states. This allows the

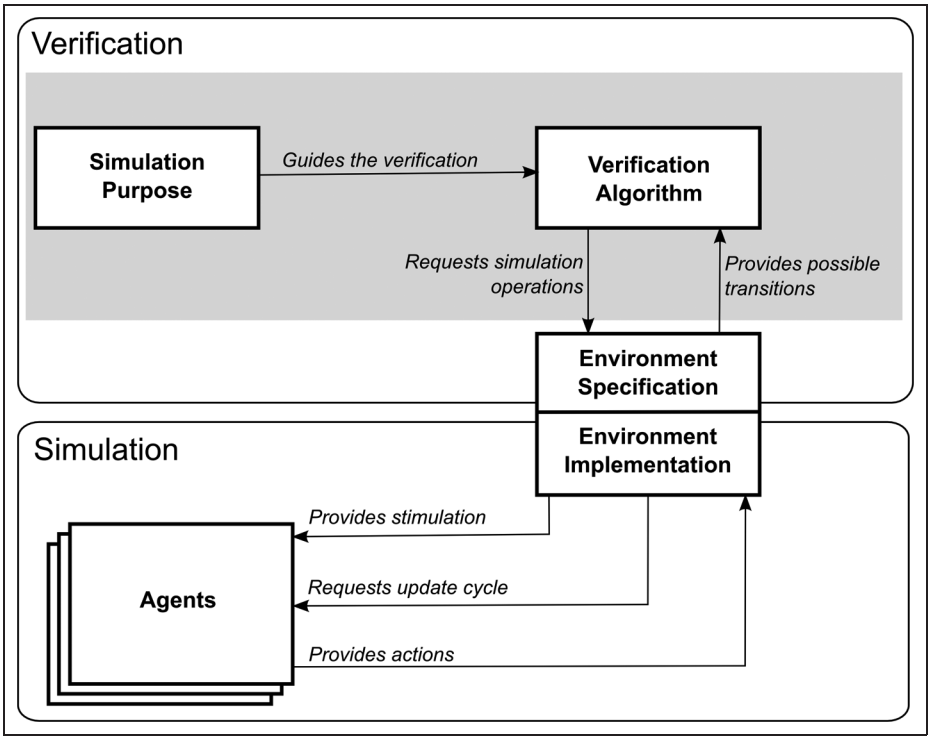

Figure 2. The approach depends both on a verification and on a simulation layer. In this article, we focus on the verification part (shaded on the diagram above), and use the simulation of MASs as an application example.

specification that some propositions are known to be false in a state (i.e. $\neg p$), but also that other propositions are not known (i.e. in case neither p nor $\neg p$ are assigned to the state). This last possibility is convenient for modeling situations in which the truth value of a proposition cannot be assessed, as it may happen in experimental situations.

Figure 3 shows an example of an ATS ($\mathcal{M}$) in the context of the first small example presented in Section 3. It models the actions that an agent can perform (i.e. bark, sit and salivate), the stimulation it can receive (i.e. the sound of a whistle or bell) and the ''clock'' that determines when time advances in the simulation (i.e. the !*commit* event, which is optional and has a role that will be better explained in Section 8.1). States are annotated with the literal h, meaning that the agent is hungry. From the verification point of view, these annotations suffice. Of course, each state in $\mathcal{M}$ is related to some simulator state, which is much more complex, but this is all abstracted away at the formal layer that we deal with in this article. The simulator state is merely supposed to exist and to be capable of receiving and providing information to the verification layer, mainly in the form of events and propositions.

An ATS represents some system that can be in several states, each one possessing a number of attributes, and a number of transition choices. The system progresses by choosing, at every state, a transition that leads to another state through some event. Given an ATS, any such particular finite sequence of its events and states is called a run.

Definition 4 (Run). *Let* $\langle S, E, P, \rightarrow, L, s_0 \rangle$ *be an ATS,* $e_0, e_1, \ldots, e_n \in E$ *and* $s_0, s_1, \ldots, s_n \in S$. *Then the sequence*

$$(s_0, e_0, s_1, e_1, \ldots, s_{n-1}, e_{n-1}, s_n)$$

is a run *of the ATS. Let us denote this sequence by* σ. *Then its length, denoted by* $|\sigma|$, *is* $n+1$. *Moreover, we also denote* σ *by* $\sigma'.e_{n-1}.s_n$, *where* σ' *corresponds to the subrun* $(s_0, \ldots, s_{n-1})$.

In the example of Figure 3, each run in $\mathcal{M}$ is a possible simulation: maybe the agent will hear a whistle and eventually salivate, maybe it will hear a bell and sit, or maybe it will follow some other sequence of events (not shown in the figure).

Let us also define the set of all possible runs of an ATS, which is used in later definitions.

Definition 5 (*runs*() function). *Let* $\mathcal{M}$ *be an ATS. Then the set of all runs of* $\mathcal{M}$ *is denoted by*

$$runs(\mathcal{M})$$

5. Simulation purposes

A *simulation purpose* is an ATS subject to a number of restrictions. In particular, it defines states to indicate either success or failure of the verification procedure, and requires that, from every other state, one of these must be reachable.

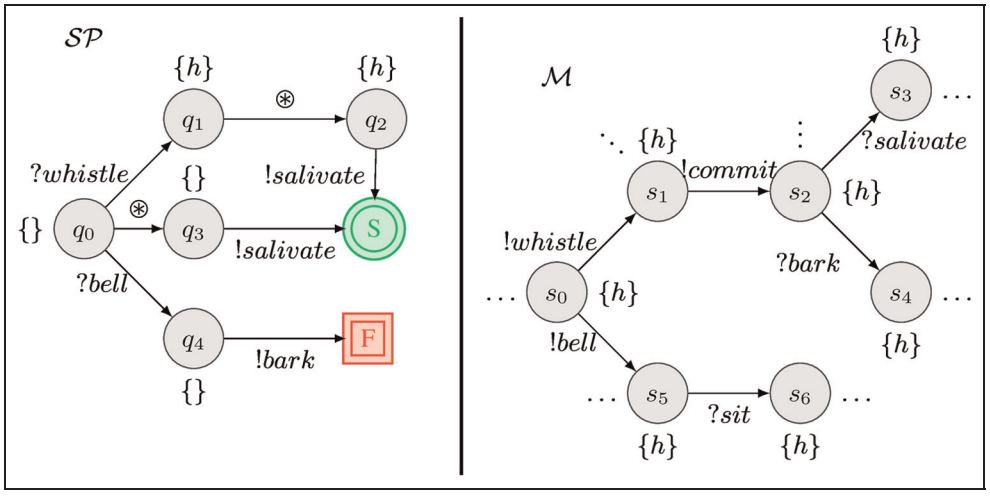

Figure 3. Examples of: (i) a general ATS, $\mathcal{M}$; and (ii) a simulation purpose, $\mathcal{SP}$. Transitions are annotated with events and states are annotated with literals. The state labeled with *S* is *Success*, and the one labeled with *F* is *Failure*. The dots (. . .) denote that $\mathcal{M}$ continues beyond the states shown (it is possibly infinite). In this example, $\mathcal{M}$ models the possible stimulation and actions of an agent (here, a dog simulation). The simulation purpose $\mathcal{SP}$, in turn, checks whether the dog has learned that when a whistle is blown, food is delivered to it (and therefore generates salivation).

Intuitively, it can be seen as a specification of desirable and undesirable simulation runs. Each possible path in a simulation purpose terminating in either success or failure defines one such run. Formally, we have the following.

Definition 6 (Simulation purpose). *A simulation purpose (SP) is an ATS* $\langle Q, E, P, \rightsquigarrow, L, q_0 \rangle$ *such that*:

(i) Q *is finite;*
(ii) $Success \in Q$ *is the verdict state to indicate success;*
(iii) $Failure \in Q$ *is the verdict state to indicate failure;*
(iv) $L(qo) = L(Success) = L(Failure) = \emptyset$;
(v) *for every* $q \in Q$, *if there are* $q', q'' \in Q$ *and* $e \in E$ *such that* $q \overset{e}{\rightsquigarrow} q$ *and* $q \overset{e}{\rightsquigarrow} q''$, *then* $q' = q''$ *(i.e. the system is deterministic);*
(vi) *for every* $q \in Q$, *there exists a run from* q *to either Success or Failure.*

These restrictions merit a few comments. First, requirement (iv) specifies that $L(q_0) = \emptyset$, which ensures that the initial state can always synchronize with the initial state of another ATS. This is a way to guide the simulation from the start. Second, condition (v) ensures that there are no two identical event paths such that one leads to *Success* and another to *Failure*. For example, if non-deterministic transitions were allowed, we could have both the run $(s_0, e_0, s_1, e_1, Success)$ and the run $(s_0, e_0, s_1, e_1, Failure)$, which is inconsistent (i.e. because identical situations must have identical verdicts). That this works shall be proven later on. Finally, restriction (vi) ensures that every state can, in principle, lead to a verdict. Nevertheless, an inconclusive verdict might be issued owing to the presence of cycles and the finite nature of simulation runs.

Figure 3 shows an example of a simulation purpose, $\mathcal{SP}$. It defines a number of experiments that we wish to perform on the agent modeled by $\mathcal{M}$. Note that the runs that lead to *Success* have as a common objective to make the agent salivate: we want to check whether and how this is possible. The simulation purpose $\mathcal{SP}$ also rules out a run that in which the agent hears a bell and sits, perhaps because we know that the dog has been trained to do this and we are not interested in this particular behavior.

The role of simulation purposes in our approach is similar to that of automata (especially deterministic finite automata (DFA)) in language recognition. Both are graphs used to detect certain properties in another object. Nonetheless, it is important to note what is different between them. In addition to the labeling of states, a simulation purpose is different from a DFA in two other ways: (a) in a simulation purpose, there are two different kinds of ''accept states'' (we call these verdict states), namely, *Success* and *Failure*; each one has a different role; (b) in a simulation purpose, every state must be able to reach a verdict state (otherwise it is a useless state for verification), which is not the case in a general DFA.[v]

A visual depiction of both a general ATS and a simulation purpose is given in Figure 3.

6. Synchronous product of an ATS and an SP

The idea that a simulation purpose can guide which runs to generate in another ATS is formalized by the notion of a synchronous product between them. Since both events and

states contain relevant information for this guidance, a particular definition of synchronization for each case is first required, as follows.

Definition 7 (Event Synchronization). *Let* $\mathcal{SP} = \langle Q, E_{sp}, P_{sp}, \rightsquigarrow, L_{sp}, q_0 \rangle$ *be a simulation purpose, and* $\mathcal{M} = \langle S, E, P, \rightarrow, L, s_0 \rangle$ *be an ATS. Moreover, let*

- $q_1 \overset{e_1}{\rightsquigarrow} q_2$ *be a transition from* $\mathcal{SP}$*; and*
- $s_1 \overset{e_2}{\rightsquigarrow} s_2$ *be a transition from* $\mathcal{M}$.

Then we define that events e_1 *and* e_2 synchronize *if, and only if, one of the following cases hold:*

- $e_1 = ?n$ *and* $e_2 = !n$ *for some name* n*;*
- $e_1 = !n$ *and* $e_2 = ?n$ *for some name* n*;*
- $e_1 = \circledast$ *and there is no* $q' \in Q$ *such that* $q1 \overset{e_2^c}{\rightsquigarrow} q'$*; or*
- $e_1 = e_2 = \tau$.

Moreover, we denote the fact that e_1 *and* e_2 *synchronize by*

$$e_1 \bowtie e_2$$

The synchronization of input and output events is quite natural and allows $\mathcal{SP}$ to choose specific events on $\mathcal{M}$. The other two cases merit a few more comments. The case in which $e_1 = \circledast$ specifies that it is a default transition, it can only happen if there is no concrete alternative. That is to say, if both $q_1 \overset{\circledast}{\rightsquigarrow} q_2$ and $q_1 \overset{?n}{\rightsquigarrow} q_3$ are in principle possible, only the latter will actually take place. Finally, the synchronization involving τ merely states that $\mathcal{SP}$ may observe that some internal, unknown, event took place.[vi]

State synchronization is simpler. It merely allows the simulation purpose to request certain literals to be present in specific states.

Definition 8 (State synchronization). *Let* $\mathcal{SP} = \langle Q, E_{sp}, P_{sp}, \rightsquigarrow, L_{sp}, q_0 \rangle$ *be a simulation purpose, and* $\mathcal{M} = \langle S, E, P, \rightarrow, L, s_0 \rangle$ *be an ATS. Moreover, let* $q \in Q$ *be a state from* $\mathcal{SP}$ *and* $s \in S$ *be a state from* $\mathcal{M}$*. Then we define that* q *and* s synchronize *if, and only if,*

$$L_{sp}(q) \subseteq L(s)$$

Moreover, we denote the fact that q *and* s *synchronize by*

$$q \bowtie s$$

Note, in particular, that if $L_{sp}(q)$ is empty, no demand is being made by $\mathcal{SP}$, and therefore q can synchronize with any state of $\mathcal{M}$.

We may then specify the overall synchronous product, which, by synchronizing events and states, selects only the runs relevant for the simulation purpose. The result of such a product, then, is an ATS that contains only the relevant runs.

Definition 9 (Synchronous product of a simulation purpose and an ATS). *Let* $\mathcal{SP} = \langle Q, E_{sp}, P_{sp}, \rightsquigarrow, L_{sp}, q_0 \rangle$ *be a simulation purpose, and* $\mathcal{M} = \langle S, E, P, \rightarrow, L, s_0 \rangle$ *be an ATS. Then their synchronous* product*, denoted as*

$$\mathcal{SP} \otimes \mathcal{M}$$

is an ATS $\mathcal{M}' = \langle S', E', P', \rightarrow', L', s_0' \rangle$ *such that:*

- $E' = E$;
- $P' = P$;
- S' *and* $\rightarrow'$ *are constructed inductively as follows*
  - **initial state** $s_0' = (q_0, s_0) \in S'$ and $L'(s_0') = L(s_0)$.
  - **other states and transitions** *built using the following rule*[vii]

$$\frac{q \overset{e_1}{\rightsquigarrow} q' \;\; s \overset{e_2}{\rightarrow} s' \;\; (q,s) \in S' \;\; e_1 \bowtie e_2 \; s' \bowtie q'}{(q,s) \overset{e_2}{\rightarrow}' (q',s')} \text{SYNCH}$$

- *if* $(q', s') \in S'$*, then* $L'((q', s')) = L(s')$.

This product defines the search space relevant for our algorithms. For this reason, we may refer to it as if it was completely computed. Nevertheless, algorithmically it can be built on-the-fly, and we shall profit from this in order to perform verification.

Figure 4 highlights the runs that synchronize in the example. Observe that while only one of the runs lead to success (shaded continuously), both are used to define the synchronous product. The $\otimes$ transition is never taken, because at that point the events of $\mathcal{M}$ can all synchronize with corresponding input and output events in $\mathcal{SP}$. Moreover, since $\mathcal{SP} \otimes \mathcal{M}$ is built on-the-fly, it is possible that during verification the ?*bell* event is also never considered (if the algorithm is looking for the continuously shaded run and happens to build it first).

We refer to runs of synchronous products as synchronous runs.

Definition 10 (Synchronous run). *A run in a synchronous product is called a* synchronous run.

In the next section we see that some such runs are special, because they convey particularly important information. For instance, in the example of Figure 3, the only synchronous run that can lead to a desirable simulation is the one shaded continuously, as shown in Figure 4:

$$((q_0, s_0), !whistle, (q_1, s_1), !commit, (q_2, s_2), ?salivate, (Success, s_3))$$

For the sake of illustration, let us consider briefly what it means to simulate this run. In the beginning, the simulator is in its initial state, at s_0. Then, the verification algorithm requests the scheduling of the event !*whistle*.

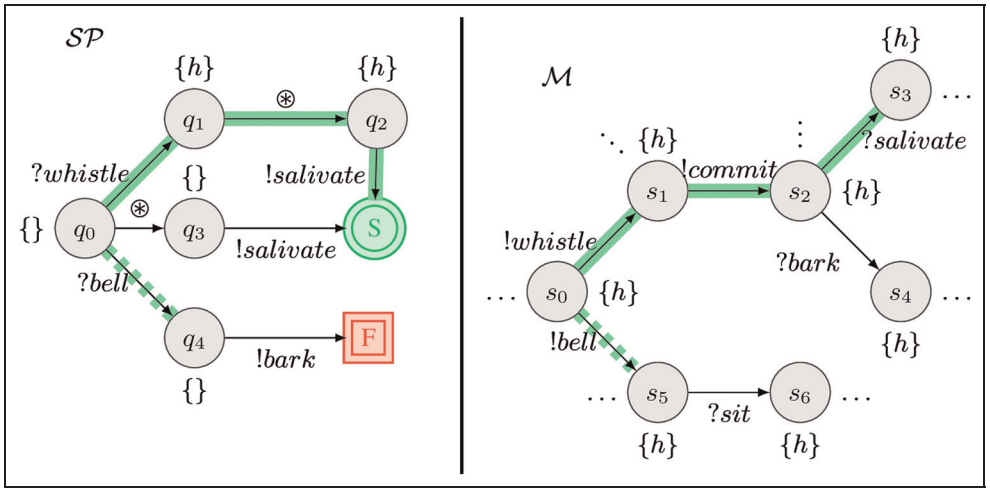

Figure 4. In this example, the simulation purpose $\mathcal{SP}$ guides the simulation so that a whistle stimulus is delivered to the agent, then anything can happen, and finally the organism salivates. There are other possibilities that follow from the $\mathcal{M}$ and $\mathcal{SP}$, when considered individually, but only the shaded runs can synchronize. There are rules that determine how events and states synchronize. The dashed shade (on transitions marked with ?*bell* and !*bell*) denotes runs that could not advance towards any verdict state, whereas the continuous shade (on the remaining shaded transitions) indicates runs that led to desirable simulations.

Afterwards the event !*commit* instructs the simulator to actually simulate the scheduled events, which leads the simulator to a new state in s_2. The verification algorithm then checks whether the agent can salivate through the event ?*salivate*. Because agents are autonomous, it could be the case that this event did not happen. In the example, though, we assume that the event did happen, which leads to the successful end of this particular simulation run.

7. Satisfiability relations

Given an ATS $\mathcal{M}$ and a simulation purpose $\mathcal{SP}$, one is interested in whether $\mathcal{M}$ satisfies $\mathcal{SP}$. There are a number of ways in which such satisfaction can be defined, and we call each one a *satisfiability relation*.[viii]

To begin with, we may be interested in whether the simulation purpose is capable of conducting to a state of either success or failure. This is to be interpreted as the possibility of constructing an experiment, which can be used as evidence either in favor or against some hypothesis. This can be done in either a weak manner or a strong manner. In the weak manner, at each step in the experiment one is concerned only with the possibility of proceeding to a next desirable step, without forbidding other courses of action. In the strong manner, on the other hand, at each step in the experiment one may forbid certain actions, so that if they are possible the step is considered useless. These notions are formalized as follows.

Definition 11 (Feasibility). *Let $\mathcal{SP}$ be a simulation purpose, $\mathcal{M}$ be an ATS and $R \subseteq runs(\mathcal{SP} \otimes \mathcal{M})$. Then we define that:*

- *$\mathcal{SP}$ is* weakly feasible *with respect to $\mathcal{M}$ and R if, and only if, there exists a synchronous run (called* weakly feasible run*) $r \in R$ such that its last state (q, s) is such that $q = Success$; otherwise, we call it* weakly unfeasible;
- *$\mathcal{SP}$ is* strongly feasible *with respect to $\mathcal{M}$ and R if, and only if, there exists a synchronous run (called* strongly feasible run*) $r \in R$ such that (i) its last state (q, s) is such that $q = Success$ and (ii) there is no state (q', s') in r such that $(q', s') \overset{e}{\rightsquigarrow} (Failure, s'')$ for some e and s''; otherwise, we call it* strongly unfeasible.

Moreover, if one of the two cases above hold, we say that $\mathcal{SP}$ is feasible with respect to $\mathcal{M}$, and the corresponding run is called feasible run.

In the example of Figure 4, $\mathcal{SP}$ is *strongly feasible* (as well as *weakly feasible*). The runs shaded continuously show how to compute the *strongly feasible run*.

Definition 12 (Refutability). *Let $\mathcal{SP}$ be a simulation purpose, $\mathcal{M}$ be an ATS and $R \subseteq runs(\mathcal{SP} \otimes \mathcal{M})$. Then we define that:*

- *$\mathcal{SP}$ is* weakly refutable *with respect to $\mathcal{M}$ and R if, and only if, there exists a synchronous run (called* weakly refutable run*) $r \in R$ such that its last state (q, s) is such that $q = Failure$; otherwise, we call it* weakly irrefutable;
- *$\mathcal{SP}$ is* strongly refutable *with respect to $\mathcal{M}$ if, and only if, there exists a synchronous run (called* strongly refutable run*) $r \in R$ such that (i) its last*

state (q, s) *is such that* $q = Failure$ *and (ii) there is no state* (q', s') *in r such that* $(q', s') \overset{e}{\rightsquigarrow} (success, s'')$ *for some e and* s''*; otherwise, we call it* strongly irrefutable.

Moreover, if one of the two cases above hold, we say that $\mathcal{SP}$ *is* refutable *with respect to* $\mathcal{M}$*, and the corresponding run is called* refutable run.

In the above definitions, the set R of runs to consider is left as a parameter because the notions of feasibility and refutability are applicable even in the case where not all possible runs (i.e. $runs(\mathcal{SP} \otimes \mathcal{M})$) can be known. Later we show how this can be used to perform verifications with respect to the incomplete observations obtained through a simulator.

A simulation purpose can be both feasible and refutable with respect to the same ATS. In such a case, it merely means that there are experiments that lead to different verdicts, which is not a contradiction.[ix]

It might be interesting, though, to know whether all of the experiments that follow from a simulation purpose lead to the same verdict. In this case, we are interested in establishing that all courses of action are either acceptable or unacceptable. This can be useful, for instance, if the simulation purpose is to model a protocol to be followed, and which, therefore, should always lead to some desirable state.

Definition 13 (Certainty). *Let* $\mathcal{SP}$ *be a simulation purpose,* $\mathcal{M}$ *be an ATS and* $R \subseteq runs(\mathcal{SP} \otimes \mathcal{M})$*. Then we define that* $\mathcal{SP}$ *is* certain *with respect to* $\mathcal{M}$ *if, and only if, every run in R terminates in a state* (q, s) *such that* $q = Success$.

Definition 14 (Impossibility). *Let* $\mathcal{SP}$ *be a simulation purpose,* $\mathcal{M}$ *be an ATS and* $R \subseteq runs(\mathcal{SP} \otimes \mathcal{M})$*. Then we define that* $\mathcal{SP}$ *is* impossible *with respect to* $\mathcal{M}$ *if, and only if, every run in R terminates in a state* (q, s) *such that* $q = Failure$.

Clearly, a simulation purpose only has value if it is capable of reaching some of its terminal states. Depending on the structure of the ATS to be verified, this might not happen in their synchronous product. In such a case the simulation purpose is incapable of providing information about the ATS. Therefore, there is an important notion of informativeness that we wish to attain.

Definition 15 (Informativeness). *Let* $\mathcal{SP}$ *be a simulation purpose and* $\mathcal{M}$ *be an ATS. Then we define that* $\mathcal{SP}$ *is* informative *with respect to* $\mathcal{M}$ *if, and only if, it is either feasible or refutable. Otherwise,* $\mathcal{SP}$ *is said to be* uninformative.

We claimed in Section 5 that simulation purposes avoid non-determinism in order to provide consistent verdicts. We shall now prove this.

Proposition 1 (Consistency). *Let* $\mathcal{SP}$ *be a simulation purpose,* $\mathcal{M}$ *be an ATS and* $Prod = \mathcal{SP} \otimes \mathcal{M}$*. Moreover, let* t_1 *and* t_2 *be runs of Prod which share a subrun t and an event e such that*

- $t_1 = t.e.(q_n^1, s_n^1)$;
- $t_2 = t.e.(q_n^2, s_n^2)$.

Then, we have that $q_n^1 = q_n^2$.

Proof. Let $n-1$ be the length of subrun t, and (q_{n-1}, s_{n-1}) be its last state. By hypothesis, $(q_{n-1}, s_{n-1}) \overset{e}{\rightarrow} (q_n^1, s_n^1)$ is a transition in *Prod*. Then, because of the determinism requirement on simulation purposes, it follows that there exists exactly one transition in $\mathcal{SP}$ from q_{n-1} using the event e, namely, $q_{n-1} \overset{e}{\rightsquigarrow} q_n^1$. Therefore, if $(q_{n-1}, s_{n-1}) \overset{e}{\rightarrow} (q_n^2, s_n^2)$ is also a transition in *Prod*, it must be the case that it arises from the synchronization with same transition in $\mathcal{SP}$, which implies that $q_n^1 = q_n^2$. □

Note that this result does not depend on determinism concerning $\mathcal{M}$, but only $\mathcal{SP}$.

8. Verification algorithms

The satisfiability relations presented in the previous section can be verified by analyzing the synchronous product given by Definition 9. The required algorithms, moreover, are all very similar: they all consist in a depth-first search on this product. For this reason, we first present and analyze extensively the algorithm for checking feasibility (Algorithm 1). By a trivial modification of the input, the same algorithm can be used to check refutability. We then define the algorithm for checking certainty (Algorithm 2), which is very similar to Algorithm 1, but do require some subtle adjustments. Again, by a trivial modification of the input, Algorithm 2 can also be used to check the remaining impossibility relation. It is therefore only necessary to provide these two algorithms to verify all the satisfiability relations defined previously. Both algorithms require the existence of a simulator interface to interact with, so we begin by introducing it.

8.1. Simulator interface

Before we proceed to the verification algorithms themselves, it is necessary to introduce a way for them to access the simulation infrastructure. We do this here by specifying a number of operations that the simulator must make available to the verification algorithms. This means that any simulator that provides this interface can be used to implement the verification technique presented here.

The required simulator interface is composed of the following operations.

- `GoToState` (*sim*): Makes the simulation execution return to the specified simulation state *sim*, which must have taken place previously.

- `CurrentState()`: Returns the current simulation state.
- `ScheduleStep(`e`)`: Schedules the specified event e for simulation.
- `Step()`: Requests that all scheduled events get simulated.
- `isCommitEvent(`e`)`: Checks whether e is an event that serves as a signal of when it is appropriate to call the `Step()` operation. Such an event can be thought of as a clock used by the simulator. We have seen this been used in the simple example constructed in the previous sections. If the simulator does not employ any such special event, then this operation always returns *true*. For variety, this will be the case in the example of the next section.
- `Successors(`*ATS, s*`)`: Calculates the finite set of all transitions in the specified ATS that can be simulated and that have the state s as their origin. This operation is necessary to allow the on-the-fly construction of the ATS.

8.2. Feasibility verification

Algorithm 1 implements feasibility verification. In addition to the simulator operations described above, it also assumes that the following simple elements are available.

- `CanSynch`$(q \overset{f}{\rightsquigarrow} q',\ s \overset{g}{\rightarrow} s')$: Checks whether the two specified transitions can synchronize according to Definition 9.
- $depth_{max}$: The maximum depth allowed in the search tree. Note that since simulations are always finite (i.e. they must stop at some point), we can assume that $depth_{max}$ is finite.
- **max_int**: A very large integer (in most common programming languages, this can be the maximum integer available, thus the name), which is used to indicate distances that can be considered as infinite. It is assumed that in a simulation purpose all shortest paths from any state to verdict states are less than **max_int**.

The remaining procedures required for the algorithm are given explicitly after it.

Let us explain informally how the algorithm works. For a mathematically rigorous analysis of its correctness and complexity, the interested reader may consult Appendix A.

How the algorithm works First of all, a preprocessing of the simulation purpose is required. This consists of calculating how far from *Success*, the desired verdict state, each of the states in the simulation purpose is. By this provision, we are able to take the shortest route from any given simulation purpose state towards *Success*. The importance of such a route is that it avoids cycles whenever possible, which is crucial to prevent the algorithm from entering in infinite loops later on. For every simulation purpose state q, then, its distance to the desired verdict state is stored in $dist[q]$. However, if one is checking strong feasibility, $dist[Failure]$ is set to -1, so that later the algorithm will always find a successor that leads to *Failure* before any other successor (in order to discard it promptly, and thereby respect the strong variant of feasibility).

Once this preprocessing is complete, the algorithm performs a depth-first search on the synchronous product $\mathcal{SP} \otimes \mathcal{M}$. The central structure to achieve this is the stack *SynchStack*. Every time a successful synchronization between a transition in $\mathcal{SP}$ and one in $\mathcal{M}$ is reached, information about it is pushed on this stack. The pushed information is a tuple containing the following items:

- The state q of $\mathcal{SP}$ that synchronized.
- The state s of $\mathcal{M}$ that synchronized.
- The event e of $\mathcal{M}$ that synchronized. This will be used later to calculate a synchronous run.
- The state p of $\mathcal{SP}$ that came immediately before the one that has been synchronized (i.e. $p \overset{e}{\rightsquigarrow} q$). Again, this will be used later to calculate a run.
- The state of the simulation, *sim*, which can be extracted from the simulator using the `CurrentState()` function.
- The set of transitions starting at q (i.e. *Unexplored* = `Successors` $(\mathcal{SP}, q)$) which have not been explored yet.
- The depth in the search tree.

In the beginning, we assume that the initial states of both transition systems synchronize and push the relevant initial information on *SynchStack*. Thereafter, while there is any tuple on the stack, the algorithm will systematically examine it. It peeks the topmost tuple, $(q, s, e, p, sim, Unexplored, depth)$, and access its *Unexplored* set. These are simulation purpose transitions beginning in q which have not yet been considered at this point. The algorithm will examine each of these transitions while: (i) no synchronization is possible (i.e. the variable *progress* is *false*); and (ii) the search depth is below some maximum limit $depth_{max}$. In case (i), the rationale is that we wish to proceed with the next synchronized state as soon as possible, so once we find a synchronization, we move towards it. If that turns out to be unsuccessful, the set *Unexplored* will still hold further options for later use. In case (ii), we are merely taking into account the fact that there are situations in which the algorithm could potentially go into an infinite depth. For instance, $\mathcal{SP}$ could contain a cycle that is always taken because no other synchronizations are possible starting from the beginning of this cycle. The depth limit provides an upper-bound in such cases, and forces

Algorithm 1: On-the-fly verification of feasibility.

Input: A simulation purpose $\mathcal{SP} = \langle Q, E_{sp}, P_{sp}, \rightsquigarrow, L_{sp}, q_0 \rangle$, the initial state s_0 of an ATS $\mathcal{M}$ and a $variant \in \{weak, strong\}$.

Output: `SUCCESS` and a feasible run; `FAILURE`; or `INCONCLUSIVE`.

```
1  dist[ ] := Preprocess(SP, Success);
2  if variant = strong then dist[Failure] = −1;
3  let SynchStack be an empty stack;
4  let sim_0 := CurrentState() ;
5  let Unexplored_0 := Successors(SP, q_0);
6  Push (q_0, s_0, nil , nil , sim_0, Unexplored_0, 0) on SynchStack;
7  let verdict := FAILURE;
8  while SynchStack ≠ ∅ do
9      Peek (q, s, e, p, sim, Unexplored, depth) from SynchStack;
10     let progress := false;
11     while Unexplored ≠ ∅ ∧ progress = false ∧ depth < depth_max do
12         q -f-> q' := RemoveBest(Unexplored, dist[ ]);
13         let depth' := depth + 1;
14         let Succs := Successors(M, s);
15         while Succs ≠ ∅ do
16             Remove some s -g-> s' from Succs;
17             GoToState(sim);
18             ScheduleStep(g);
19             if isCommitEvent(g) then
20                 Step();
21             if CanSynch(q -f-> q', s -g-> s') then
22                 if q' = Failure ∧variant = strong then
23                     Pop from SynchStack;
24                     progress := true;
25                     Succs := ∅;
26                 else
27                     sim' := CurrentState();
28                     unexplored' := Successors(SP, q');
29                     Push (q', s', g, q, sim', unexplored', depth') on SynchStack;
30                     progress := true;
31                 if q' = Success then
32                     return SUCCESS and BuildRun(SynchStack, q', depth');
33     if depth ≥ depth_max then verdict := INCONCLUSIVE;
34     if progress = false then Pop from SynchStack;
35 return verdict;
```

the search to try other paths which might lead to a feasible run, instead of an infinite path.

In each iteration of this while loop, the algorithm selects the best transition $q \overset{f}{\rightsquigarrow} q'$ available in *Unexplored*. This selection employs the preprocessing of the simulation purpose, and merely selects the transition that is closer to the goal. That is to say, q' is such that there is no $q \overset{f}{\rightsquigarrow} q''$ such that $dist[q''] < dist[q']$. As we remarked above, this is intended to guide the search through the shortest path in order to avoid cycles whenever possible. Once such a transition is chosen, we may examine all possible transitions of $\mathcal{M}$ starting at s, the current synchronized state.

At this point, the simulator interface will be of importance. For each such transition $s \overset{g}{\rightarrow} s'$, we have to instruct the simulator to go to the simulation state *sim* in the peeked tuple. This simulation state holds the configuration of the structures internal to the simulator that correspond to the transition system state s. The algorithm may then request that the event g be scheduled. Then, if the event turns out to be a commit event, the simulator is instructed to perform one simulation step, which implies in delivering all the

Procedure Preprocess($\mathcal{SP}$, v)

Input: A simulation purpose $\mathcal{SP} = \langle Q, E, P, \rightsquigarrow, L, q_0 \rangle$ and a verdict state v.
Output: A function $dist : Q \longrightarrow \mathbb{Z}$ such that, for every $q \in Q$, $dist[q]$ is the minimal distance between q and v.

```
1  let visited[ ] be a map from states to Boolean values;
2  let dist[ ] be a map from states to either integers or nil ;
3  foreach q ∈ Q do
4  |  visited[q] := false;
5  |  dist[q] := nil ;
6  PreprocessAux(SP, q0, v, dist[ ], visited[ ]);
7  foreach q ∈ Q do
8  |  visited[q] := false;
9  PreprocessAux(SP, q0, v, dist[ ], visited[ ]);
10 return dist[ ];
```

Procedure PreprocessAux($\mathcal{SP}$, *source*, v, *dist*, *visited*)

Input: A simulation purpose $\mathcal{SP} = \langle Q, E, P, \rightsquigarrow, L, q_0 \rangle$, a $source \in Q$, a map $dist[\,]$, a map $visited[\,]$ and a verdict state v.

```
1  visited[source] = true;
2  if source = v then
3  |  dist[source] := 0;
4  else
5  |  let min := nil ;
6  |  if Successors(SP, source)= ∅ then
7  |  |  min := max_int ;
8  |  else
9  |  |  foreach source -f-> q' do
10 |  |  |  if visited[q'] = false then
11 |  |  |  |  PreprocessAux(SP, q', v, dist[ ], visited[ ]);
12 |  |  |  if dist[q'] ≠ nil then
13 |  |  |  |  if min = nil then
14 |  |  |  |  |  min := dist[q']
15 |  |  |  |  else if dist[q'] < min then
16 |  |  |  |  |  min := dist[q'];
17 |  |  if min ≠ nil ∧ min ≠ max_int then
18 |  |  |  min := min + 1;
19 |  dist[source] := min;
```

Procedure RemoveBest(*Unexplored*, *dist*)

Input: A set *Unexplored* of transitions of a simulation purpose and a map $dist[]$ from of states to integers.
Output: A transition in *Unexplored*.

```
1  let q -e-> q' ∈ Unexplored such that there is no q -e-> q'' ∈ Unexplored with dist[q''] < dist[q'];
2  return q -e-> q'
```

Procedure BuildRun(SynchStack, q^*, $depth^*$)

Input: A search stack *SynchStack*, the last simulation purpose state q^* to include in the run to be built, and the depth $depth^*$ of this state.

Output: A synchronous run.

```
1 let Run be a list initially empty;
2 while SynchStack ≠ ∅ do
3     Pop (q', s', f, q, sim, Unexplored, depth) from SynchStack;
4     if q' = q* ∧ depth* = depth then
5         Put (q', s') at the beginning of Run;
6         Put f at the beginning of Run;
7         q* := q;
8         depth* := depth* − 1;
9 return Run;
```

scheduled events. This will put the simulator into a new state, which will correspond to s' in $\mathcal{M}$. Then we may check whether $q \overset{f}{\rightsquigarrow} q'$ and $s \overset{g}{\rightarrow} s'$ can synchronize. If it is possible, then q' is *Success*, *Failure* or another state in Q. In the first case we have found the feasible run we were looking for and we are done. The second case only matters if we are checking the strong variant of feasibility, in which case we must discard the current synchronization state (by popping it from *SynchStack* and going to the next one) because it has led to a *Failure* and thus according to the definition of strong feasibility cannot be part of the strong feasible run. In the third case, we merely push the current (q', s', g, q, sim', $unexplored'$, $depth'$) tuple on *SynchStack* for later analysis and signal that the search can move on by setting *progress* to *true*.

If the algorithm abandons a search branch because its depth is greater than or equal to $depth_{max}$, the verdict in case of failure is set to be `INCONCLUSIVE`, since it is possible that there was a feasible run with length greater than $depth_{max} + 1$ that was not explored. Moreover, it is possible that after examining all transitions in *Unexplored*, none synchronized (i.e. the variable *progress* is still set to *false*). If this happens, the tuple is popped from *SynchStack* because by then we are sure that no feasible run will require it.

Finally, if *SynchStack* becomes empty, it means that no run in $\mathcal{SP}$ up to $depth_{max}$ leads to a feasible run. So we return a verdict which will be `FAILURE` if $depth_{max}$ was never reached, or `INCONCLUSIVE` otherwise.

How the algorithm handles cycles We have just said that a preprocessing of $\mathcal{SP}$ is required in order to deal with its cyclic paths. By this provision, we are able to determine at any state of $\mathcal{SP}$ which successor is closer to *Success*, the desired verdict state. Since any cyclic path from a state is longer than an acyclic one from the same state, this suffices to avoid cycles whenever possible. That said, let us see how this minimum distance is calculated by `Preprocess()`.

The calculation is divided between the main `Preprocess()` procedure and the auxiliary `PreprocessAux()` procedure. Indeed, `Preprocess()` merely: (i) initializes two maps, $visited[\,]$ and $dist[\,]$, which store whether a state has been visited and the distance from a state to the desired verdict, respectively; and (ii) call `PreprocessAux()` twice. In the first call, all of the acyclic paths are examined and have the corresponding $dist[\,]$ values set. In the second call, using this partial $dist[\,]$, `PreprocessAux()` is then capable of computing the distances for the states in cyclic paths as well.

`PreprocessAux()` is a function that for a given *source* state recursively examines all of its successor states q' to determine which one is closer to the desired verdict state v. Once the closer successor $q*$ is found, the function merely sets $dist[source] := dist[q*] + 1$. The recursion base takes place in three situations. First, when the *source* being examined is actually the verdict state v, and therefore its distance is $depth*$. Second, if the *source* being considered has no successors, and thus cannot get to v, which implies that $dist[source]$ is infinite. Third, when all successors of *source* have already been visited.

In the latter case it means that the procedure has found a cycle. Moreover, because of the recursive nature of the procedure, none of these successors q' will have their $dist[q']$ set yet, so that $dist[source]$ remains *nil*, which indicates that *source* is in a cycle. However, when `Preprocess()` calls `PreprocessAux()` a second time, these $dist[q']$ will be set, so that even in the case in which *source* is in a cycle, we are able to assign it a distance. This distance, indeed, is nothing but the sum of an acyclic path and the length of the corresponding cycle. That is to say, for any *source* located in a cyclic path, the procedure assigns it the shortest distance considering a way to get out of the cycle. Since by Definition 6 there is always an acyclic path towards a verdict state, it is always possible to calculate this distance. This does not prevent Algorithm 1 from taking such a cycle infinitely, but merely provides

guidance to avoid it whenever possible. This guidance is enforced by the `RemoveBest()` procedure used by Algorithm 1. However, it might be the case that $\mathcal{M}$ never gives the chance for $\mathcal{SP}$ to get out of the cycle. In this case, the algorithm proceeds until the search reaches the maximum depth $depth_{max}$. When this limit is reached, the particular search branch under examination is abandoned and the algorithm backtracks to try another branch. If no branches are left, the algorithm terminates and returns the `INCONCLUSIVE` verdict. This leads us to the next topic.

Hints on termination The complete treatment concerning termination is given in Appendix A. Let us nonetheless give some hints on this matter here. The termination of Algorithm 1 depends on whether $\mathcal{SP}$ is cyclic. If it is not, termination is always guaranteed because only finitely many states of $\mathcal{SP}$ are visited during the depth-first search (i.e. no previously visited state of $\mathcal{SP}$ would be revisited). However, if there are cycles in $\mathcal{SP}$, it would in principle be possible that infinitely many states were to be visited (since the same state in $\mathcal{SP}$ could synchronize infinitely many times with states in $\mathcal{M}$), which of course would compromise termination.

In such a cyclic case, the crucial factor that determines termination is the value of $depth_{max}$. Since $depth_{max}$ is assumed to be finite (i.e. the search is bounded), termination is guaranteed, because: (i) there are only finite many paths in $\mathcal{SP}$ of length $depth_{max}$; and (ii) each such path is used only once. If the search was not bounded in this manner, such a guarantee could not be given. But, as explained previously, by their very nature simulations must be finite, so the assumption of a bounded search is actually necessary.

Hints on complexity The exact analysis of the complexities is provided in Appendix A. Here it suffices to say that the complexity in space is polynomial with respect to $depth_{max}$ and other parameters, and the complexity in time is exponential with respect to $depth_{max}$. This exponential complexity in time arises partly from the fact that the algorithm do not keep track of the visited states of $\mathcal{M}$ and partly from the possibility that there are infinitely many states in $\mathcal{M}$. Note also that if the algorithm either employed a breadth-first search instead of a depth-first search or recorded visited states, the efficient use of space would be compromised.

8.3. Refutability verification

To verify refutability, one must look for a run leading to *Failure* instead of a run leading to *Success*. Therefore, it suffices to swap the *Success* and *Failure* states in the simulation purpose and then apply Algorithm 1.

8.4. Certainty verification

The verification of certainty can be achieved by a slightly modified version of Algorithm 1. Like for refutability, it should search for the *Failure* state. However, when it does find it, the final verdict is `FAILURE`, because by Definition 13 there should be no run leading to *Failure*. Moreover, for every visited transition $q \overset{f}{\rightsquigarrow} q'$ of $\mathcal{SP}$, there must be a synchronizable $s \overset{g}{\rightarrow} s'$. Otherwise, there would be a terminal state (q', s') such that $q' \neq Success$, which is forbidden by Definition 13. Finally, when the outer loop terminates, the verdict to be returned is either `SUCCESS` or `INCONCLUSIVE` (in case the search depth reached the maximum allowed), because no counter-examples have been found. Algorithm 2 incorporates these changes. The complexities remain the same, because these modifications do not change the arguments used to calculate them.

8.5 Impossibility verification

To verify impossibility, one must find that all runs in the synchronous product lead to *Failure*. It suffices then to swap the *Success* and *Failure* states in the simulation purpose and then apply Algorithm 2. This is similar to the verification of refutability, which can be accomplished by Algorithm 1 through such a swap.

9. Another example

Recall from Section 3 that we are interested in the simulation and verification of MASs. Let us then continue one of the examples suggested there, namely, that of modeling and analyzing an online social network.

There are, of course, a large number of events to account for in such a network. For simplicity, though, we will only consider the following:

- $!gui_i$: a graphical user interface i is chosen for the website;
- $!ad_i^{ag}$: advertisement i is delivered to agent ag;
- $?buy_i^{ag}$: agent ag buys product i;
- $?msg_{ag_1}^{ag_2}$: agent ag_1 sends a message to agent ag_2.
- $?app_i^{ag}$: agent ag runs application i; here, an application is any separate sub-program that the website makes available, such as a game.

Similarly, a number of propositions about the states can be defined. For the example, these suffice:

- m^{ag}: true if and only if agent ag likes music;
- y^{ag}: true if and only if agent ag is young.

From an implementation point of view, we require three different interacting parts:

Algorithm 2: On-the-fly verification of certainty.

Input: A simulation purpose $\mathcal{SP} = \langle Q, E_{sp}, P_{sp}, \rightsquigarrow, L_{sp}, q_0 \rangle$ and the initial state s_0 of an ATS $\mathcal{M}$.
Output: `SUCCESS`; `FAILURE` and a refutable run; or `INCONCLUSIVE`.

```
1  dist[ ] := Preprocess(SP, Success);
2  let SynchStack be an empty stack;
3  let sim_0 := CurrentState() ;
4  let Unexplored_0 := Successors(SP, q_0);
5  Push (q_0, s_0, nil , nil , sim_0, Unexplored_0, 0) on SynchStack;
6  let verdict := SUCCESS;
7  while SynchStack ≠ ∅ do
8      Peek (q, s, e, p, sim, Unexplored, depth) from SynchStack;
9      while Unexplored ≠ ∅ ∧ depth < depth_max do
10         q -f-> q' := RemoveBest(Unexplored, dist[ ]);
11         let depth' := depth + 1;
12         let progress := false;
13         let Succs := Successors(M, s);
14         while Succs ≠ ∅ do
15             Remove some s -g-> s' from Succs;
16             GoToState(sim);
17             ScheduleStep(g);
18             if isCommitEvent(g) then
19                 Step();
20             if CanSynch(q -f-> q', s -g-> s') then
21                 sim' := CurrentState();
22                 unexplored' := Successors(SP, q');
23                 Push (q', s', g, q, sim', unexplored', depth') on SynchStack;
24                 progress := true;
25                 if q' = Failure then
26                     return FAILURE and BuildRun(SynchStack, q', depth');
27         if progress = false then
28             return FAILURE and BuildRun(SynchStack, q, depth);
29     if depth ≥ depth_max then verdict := INCONCLUSIVE;
30     Peek (q, s, e, p, sim, Unexplored, depth) from SynchStack;
31     if Unexplored = ∅ then Pop from SynchStack;
32 return verdict;
```

- an implementation the website model, which defines all of the operations that the website can perform on agents, as well as the reactions to their actions;
- an implementation of the first agent, which receives input from and produces output to the website;
- a similar implementation of the second agent.

That is to say, it is a MAS such that the website is an environment in which the agents exist and through which they interact. The exact manner in which these parts are built can vary, provided that their simulation ultimately produces discrete events and annotated states. For example, one could implement all of these as traditional programs (e.g. in C, Java) or as special purpose programs (e.g. executable formal specifications). In any case, we should obtain an $\mathcal{M}$ such as that of Figure 5.

Furthermore, let us assume that the objective of this model is to analyze marketing strategies for a product A targeted to young music lovers. In this way, a possible simulation purpose could be informally described as the following question: is there a way to setup the website and deliver advertisement so that users will buy the advertised product? Formally, we could have something like the simulation purpose shown in Figure 6. Note that the event $?msg_{ag_1}^{ag_2}$ is not present in this simulation purpose, although it is probably a frequent event in such a social network.

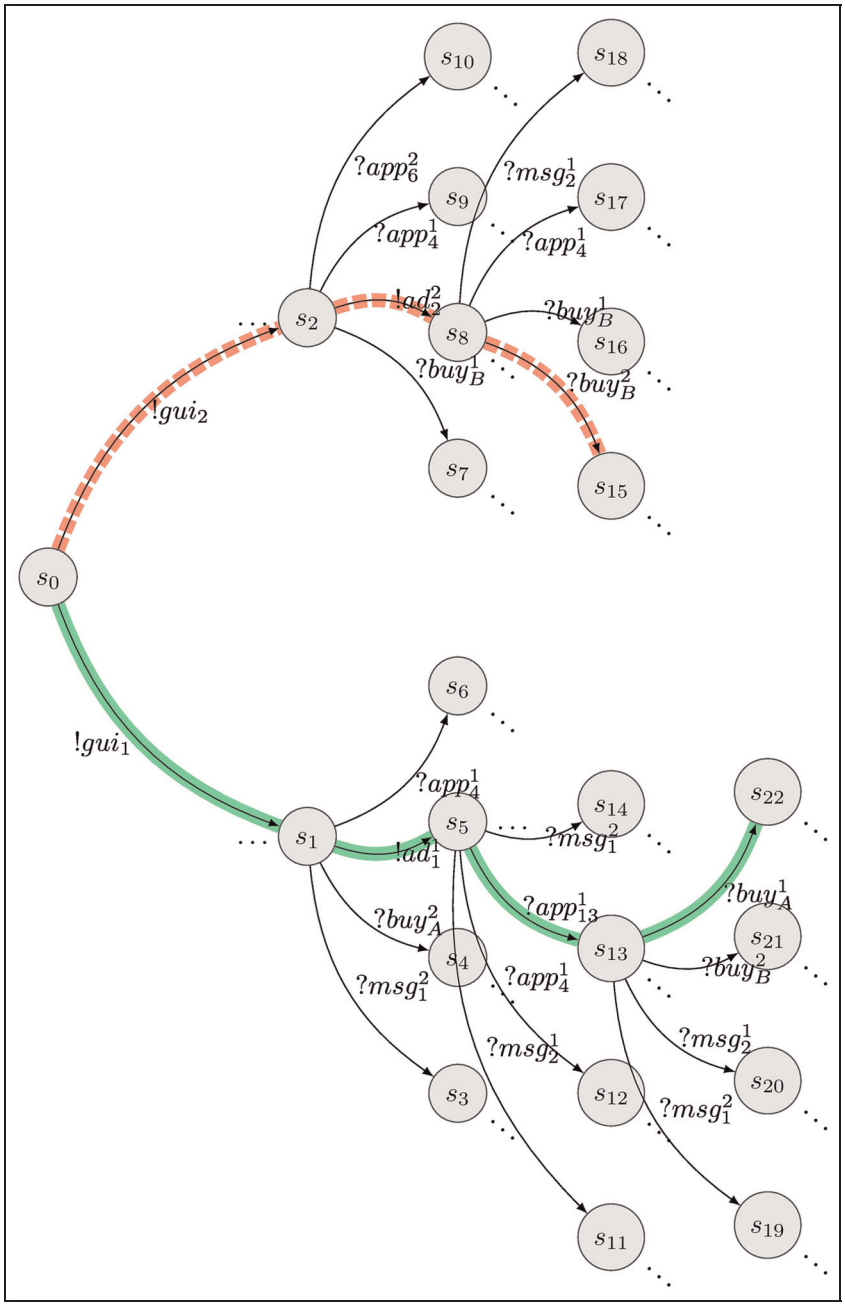

Figure 5. A partial view of the example's $\mathcal{M}$, which potentially has an infinite number of states and transitions (and even if finite, the branching nature of the structure makes the state-space exponentially large). Dots (. . .) denote that the structure continues beyond what is shown. The shaded runs are special: the continuous one (below s_0) is a successful simulation (the basis for a feasible run), whereas the dotted one (above s_0) is a failed simulation (the basis for a refutable run). All states are assumed to be annotated with $\{m^1, y^1, m^2, y^2\}$, which is not shown in the image (to avoid clutter).

This means that the event is not considered relevant for the task at hand, and thus a large part of the possible simulations can be safely ignored.

We may then investigate whether $\mathcal{SP}$ is feasible over $\mathcal{M}$. That is to say, whether it is *possible* to convince users to buy a product by following some marketing strategy. A positive verdict would not only suggest that it is possible, but would also provide one such strategy. The experimenter, then, could use this strategy in the actual social network, not the simulated one. As long as the model $\mathcal{M}$ is a sufficiently accurate representation of reality, such an approach would provide a way to evaluate several ideas *before* applying them in the real system, which not only would save resources, but would also prevent bad

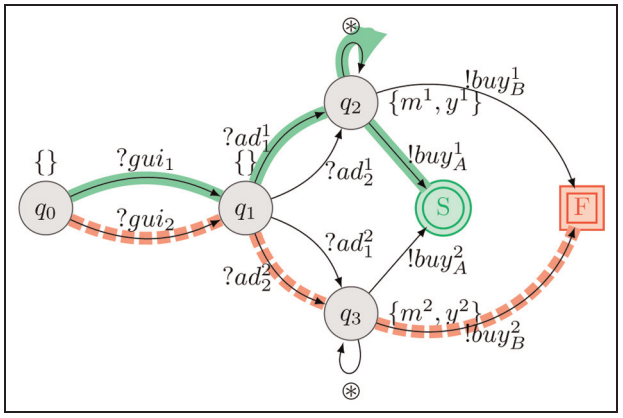

Figure 6. The example's simulation purpose . We assume only two agents for this example, and only one competing product, *B*. A possible run could be as follows: we implement GUI 1 ($q_0 \overset{?gui_1}{\rightsquigarrow} q_1$), deliver advertisement 1 to agent 2 ($q_1 \overset{?ad_1^2}{\rightsquigarrow} q_3$), and then realize that agent 2 indeed buys product *A*($q_3 \overset{?buy_A^2}{\rightsquigarrow}$ *success*).

strategies from causing damage (e.g. by submitting real users to an unbearable amount of advertisement).

The investigation of $\mathcal{SP}$ proceeds by building the synchronous product $\mathcal{SP} \otimes \mathcal{M}$ on-the-fly. That is to say, at each state of $\mathcal{SP} \otimes \mathcal{M}$, the simulator builds the next possible states and take the one which is relevant for the satisfiability relation being checked. Note that this implies, in particular, that it may not be necessary to build the *whole* synchronous product. In the example, since we are checking feasibility, the algorithm can safely terminate as soon as a feasible run is found. Figure 7 shows the full synchronous product. However, provided that one started the search by the $!gui_1$ event, only the continuously shaded path would be actually built and simulated. On the other hand, if we wanted to check for certainty, the whole of it would have been built, and the verification would actually fail, since the synchronous product also contains an refutable run.

This example exploits some of the key features of our technique:

- Social models are naturally complex and it would be unrealistic to consider exact and exhaustive analyzes. Hence, using simulations to partially explore them is preferable.
- Verdicts are calculated in a constructive manner, which allows the provision of instructions to achieve the verdict (and, therefore, replicate the result on the real system).
- Much of the model is irrelevant for some verification tasks. In the example, the event $!msg_{ag_1}^{ag_2}$ was not pertinent for the verification. It is convenient then that the property to be verified contains this information explicitly and is capable of avoiding some irrelevant simulations.
- Some forms of failure are explicit. In the example, if the agent buys the competitor product *B*, the marketing strategy is considered to have failed for that agent, and therefore one can immediately proceed to the analysis of another agent.

From the purely formal definitions given in previous sections, it is not clear what exactly is the difference between input and output events. This example helps to shed some light on this. First, it should be noted that the distinction between input and output events is a matter of

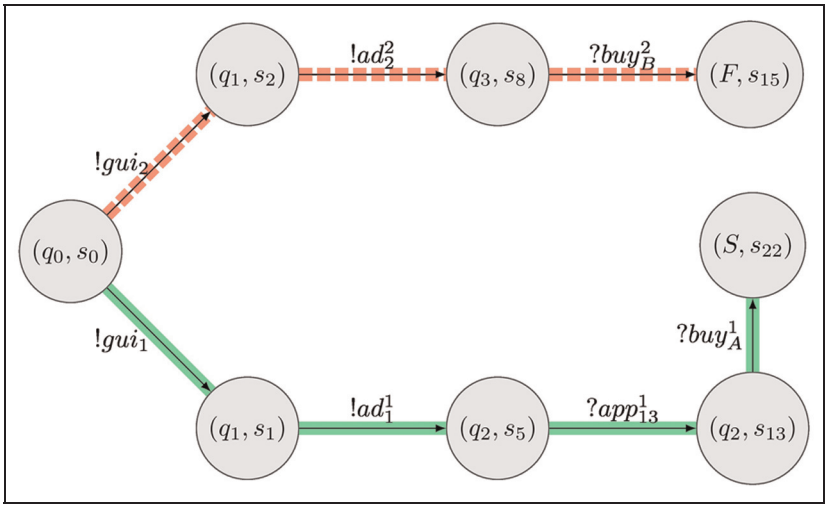

Figure 7. The synchronous product $\mathcal{SP} \otimes \mathcal{M}$. Note that this is can be seen as a sub-graph of $\mathcal{M}$ with the transitions recording the event synchronizations. The construction of the synchronous product is a way to select a finite set of runs on $\mathcal{M}$. All states are assumed to be annotated with $\{m^1, y^1, m^2, y^2\}$, which is not shown in the image (to avoid clutter).

convention: one could very well adopt the inverse terminology. The important thing is just that there have to be complementary classes of events, because they have different roles in $\mathcal{M}$. One of these classes of events is under the complete control of the experiment. In the example, this is the class of events corresponding to the website: we wish to be free to test new user interfaces, deliver ads to visitors, and so on. This can be seen in $\mathcal{M}$ when it requests (i.e. outputs an order) that the simulator performs, say, event $!gui_1$. The other class of events models the *possible* actions of agents outside our control, which may never happen, such as whether a user will buy the advertised product. Again, this is present in $\mathcal{M}$, where the event $?buy_A^2$ asks (i.e. waits for an input) the simulator whereas agent 2 bought product A. In practice, this means that: (i) the implementation of the simulated user cannot be forced to act (e.g. perhaps because it is a black-box implementation of user behavior), but only be stimulated and observed; and (ii) the implementation of the website simulation must be such that we can force it to take any of its possible actions at any time. The example mirrors real life: developers really have total control over the websites they program, but can only stimulate and observe their users. In the simulation, these differences must be enforced by the implementations of the `Successors()` and `ScheduleStep()` operations of the simulator interface, which generates $\mathcal{M}$ and handles the events as they are found there, respectively (see Section 8.1).

Note that the events found in $\mathcal{SP}$ do not give or receive orders to or from the simulator. They are there merely to guide $\mathcal{M}$, and this guidance is achieved by synchronizing both transition systems. However, for the sake of uniformity and simplicity, we defined event synchronization using the traditional idea of complementarity between input and output events.

We have chosen this simple example in order to stay focused on the simulation and verification techniques proposed. However, more complex and detailed examples can be found in the work of da Silva.[11] The examples shown there explicitly use agent and environment models to generate $\mathcal{M}$, whose formalisms are out of the scope of the present work, but can be found partly in da Silva and de Melo[33,34].

10. Conclusion

In this work we have seen how to check whether an ATS (describing a system of interest) conforms to a simulation purpose (describing a property of interest) in a number of different and precisely defined senses. In this way, our method operates on formal structures in order to verify something: it is thus a *formal* verification approach. Nevertheless, there is a crucial counterpoint to this formal aspect: the ATS under verification is, itself, representing the results of a simulation, whose internal details are not available for the verification procedure, and therefore, from our algorithmic perspective, can be seen as not formal. Verification and simulation interact by means of a simulator interface, which provides the required elements for the construction of transition systems, but do not reveal how they were generated.

This interplay between verification and simulation is a distinctive feature of our approach to the analysis of simulations. The required simulator interface is simple to implement: in essence, it merely requires that (i) simulations proceed from state to state by discrete events; and that (ii) simulation states may be saved and restored. Our technique, therefore, has clear practical applications. We are experimenting with it in our own simulator, but owing to this simple interface requirement, it should also be possible to incorporate it in other existing platforms.

There is also an interesting theoretical property in this approach: as long as it depends on a black-box (i.e. the simulator), the verification procedure can never provide a guarantee that some event cannot be observed. The reason is simply that any analysis of the black-box depends on extracting finite runs from it, and since nothing dictates the choices of the black-box, it can always be the case that a hitherto unobserved event takes place just after our last observation. This argument holds for any observation of finite length, however long. Note that this limitation is inherit to the *problem* of analyzing simulations, and not a difficulty pertaining to our specific *solution*.

One may then wonder what is the use of any such method of analysis. The answer, though, is equally simple. While one cannot guarantee the impossibility of an event, one *can* guarantee the possibility of an event, for to observe it is to prove that it is *possible*. Since this much can be done, it follows immediately that it is worth exploring simulations systematically in the search of what is possible. In fact, the whole point of simulating something is to do this; our contribution concerns the automation of such a search, which is often carried out manually by setting up initial conditions.

Furthermore, even if one cannot guarantee that a certain event will never happen, it is better to test whether it may or may not happen, for at least by doing this something can be discovered (e.g. that the event does not happen *always*, a case which is easy to detect). This judgment, the reader will recognize, is nothing but the common sense of empirical sciences, in which nothing can be proved definitively, but which nonetheless produce useful theories. To put this philosophical point of view in an formal framework is part of we set out to do in Section 3, and thus is also a contribution of this work.

The article dealt with low-level representations only, in the form of transition systems and algorithms to manipulate them. The reason for this is that this is the simplest way we could find to formalize the fundamental concept

of our approach: that simulations explore state-spaces and that these explorations can be systematically guided. Accordingly, our worry here is with simple semantic models, which provide the basis upon which various other abstractions could be built. It is assumed that whatever high-level abstraction is used, it can be translated to ATSs and simulation purposes.

Our implementation of the technique is out of the scope of this article, because it depends on various other abstractions to model MASs. It is worth to point out, though, that we employ the π-calculus[1] process algebra to formalize the environment in which agents exist, whereas the agents themselves are (from the simulation and verification perspective) black-boxes that implement certain interfaces (recall the general architecture shown in Figure 1). Thanks to the operational semantics of the π-calculus, it is possible to translate (on-the-fly) our high-level representation into an ordinary ATS and explore it using a simulation interface as described in Section 8.1. For more details, see da Silva[11] and da Silva and de Melo.[33]

Our particular choice of high-level abstraction, however, is not essential to the application of the technique. One may envisage the use of many other things, both formal and informal. On the formal front, other process algebras, such as CCS[1] and CSP,[35] could be used to generate a state-space; DEVS[29] could provide the basis for the generation of simulation traces as well; temporal logic formulas, in turn, could be used to generate simulation purposes, provided that they are properly adjusted to handle finite execution traces (see Section 2 for a discussion on the problems with traditional temporal logics). While a formal simulation purpose is necessary, formalisms are not essential for the definition of the simulation state-space. It would be perfectly possible to simply implement a simulation model using any ordinary programming language (e.g. C, Java) in such a way that transitions generate observable events and states. Existing platforms, such as RePast[3] and Mason,[4] could adopt this strategy to strengthen their analytical capabilities without much effort. Indeed, the lack of sophisticated experiment automation tools is a common shortcoming[6] in MAS simulators (which largely rely in usual programming languages to define simulation models) that could be partly addressed by adopting such an approach.

It is also possible to consider different satisfiability relations. The ones we provided (i.e. (weak and strong) feasibility, (weak and strong) refutability, certainty and impossibility) are those which we found most natural and useful, but one may devise others, more suitable to different applications. For example, instead of looking for a single feasible run, as required by the feasibility relation, one may be interested in finding a few of them. Or one may desire an even stronger notion of strong feasibility that would rule out any refutable run in the synchronous product. These new relations would require adjustments on the algorithms, though their fundamental mechanisms could remain the same (much like the two algorithms we gave, which are very similar).

Finally, it is worth emphasizing that the proposed verification framework is not limited to the simulation of MASs, which motivated our research. It was developed to address such a concern, but ATSs are general formal structures to represent states and transitions, and therefore can be used to formalize anything that can be put in their terms. This is an unexpected but fortunate consequence of our efforts.

Acknowledgements

The authors would like to thank: Professor Dr Marie-Claude Gaudel, from University Paris-Sud, for crucial suggestions that led to the present work; and the anonymous reviewers, who provided valuable feedback that improved the final version of this article.

Funding

This work was supported by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) and Conselho Nacional de Desenvolvimento Cientfico e Tecnológico (CNPq).

Notes

i The word ''simulation'' carries different meanings to different people. Owing to the different background in formal methods that readers may have, a clarification is in order. By ''simulation'' we do not mean a formal relation among two transition systems, such as what Milner[1] employs. As we explain, in this article a ''simulation'' refers, broadly, to an abstract reproduction of some target system by means of a detailed and executable model. That is to say, we use the term somewhat informally here. To avoid confusion, we do not refer to other meanings of the term in the text. Nevertheless, readers with a more formally inclined mind may recognize similarities between some of our formal definitions and formal simulation relations (e.g. that of Milner[1]). In particular, we define a number of satisfiability relations that depend on one transition system ''mimicking'' parts of another transition system: a clear reminder of formal simulation relations.

ii Current examples of such networks include popular websites such as www.facebook.com, www.twitter.com and plus.google.com.

iii Our definition of ATS is very similar to what is merely called a transition system by Baier and Katoen.[7] We think, however, that it is worth emphasizing that it is a special kind of transition system, in order to avoid confusion. In particular, an ATS is not what is usually called a labeled transition system (LTS),[1] in which only transitions are labeled. We also impose certain criteria of finiteness.

iv By $\mathbb{P}(P \cup \neg P)$ we mean the power set of $(P \cup \neg P)$, i.e. the set of all subsets of $(P \cup \neg P)$, and by $\neg P$ we mean the set $\{\neg p | p \in P\}$.

v Concerning the name of the structure, we chose to employ the term ''simulation purpose'' instead of some

modification of the more classic term ''automaton'' because our main direct inspiration comes from TGV.[12] As mentioned in Section 2, that is an approach to model-based testing which employs a similar structure named ''test purpose''. The name adopted reflects this connection and, of course, emphasizes what the structure is supposed to be used for.

vi To see this more clearly, consider that the purpose of τ is to allow the specifier to hide events (i.e. internal events) that he or she does not wish the $\mathcal{SP}$ to directly guide (e.g. setup the simulation model in some arbitrary initial configuration). One benefit of doing this is that many events that are irrelevant in some situation can simply be seen as τ. In this manner, the specification of $\mathcal{SP}$ becomes simpler: instead of defining what may happen to each such event, it suffices to just add *one* τ transition that captures what should happen to *all* of them. Thus, $\mathcal{M}$ should be able to produce τ and $\mathcal{SP}$ should be able to demand it, which implies synchronization on τ. Furthermore, although τ models ''hidden'' events, the synchronization is defined so that at least the amount of such events may be revealed: a τ in $\mathcal{SP}$ must be matched by exactly one τ in $\mathcal{M}$. To specify an unbounded sequence of τ in $\mathcal{SP}$, it suffices to specify a loop from a state to itself, with the transition labeled by τ. Finally, note also that what should and should not be seen as an internal event depends on the particular application and is up to the implementer.

vii As seen in Definitions 7 and 8, $a \bowtie b$ means that a and b synchronize.

viii Note that this notion of satisfiability is similar in its intent to what is found in model checking, since in both cases it indicates that some property ($\mathcal{SP}$ in our case) holds with respect to some system ($\mathcal{M}$ in our case). The difference here is that: (i) our simulation purposes are transition systems, not logical formulas; and (ii) there is more than one way in which we can say that an ATS satisfies a simulation purposes.

ix In contrast, as remarked earlier, a contradiction would follow if *the same* experiment was to lead to different verdicts. But owing to the definition of simulation purposes, this can never happen, as we show in Proposition 1.

x The division of correctness in soundness and completeness only makes sense if there is something external to the algorithms to used as a standard. The auxiliary procedures require no such standard, and therefore we refer to the fact that they work as specified merely as correctness.

References

1. Milner R. *Communicating and Mobile Systems: The π-calculus*. New York, NY: Cambridge University Press, 1999.
2. Gilbert N and Bankers S. Platforms and methods for agent-based modeling. *Proc Natl Acad Sci U S A* 2002; 99(Suppl. 3): 7197–7198.
3. North M, Collier N and Vos JR. Experiences creating three implementations of the Repast agent modeling toolkit. *ACM Trans Model Comput Sim* 2006; 16(1): 1–25.
4. Luke S, Cioffi-Revilla C, Panait L and Sullivan K. MASON: A new multi-agent simulation toolkit, 2004. http://cs.gmu.edu/~eclab/projects/mason/.
5. Marietto MB, David N, Sichman JS and Coelho H. Requirements analysis of agent-based simulation platforms: State of the art and new prospects. In *Multi-Agent-Based Simulation II: Third International Workshop (MABS 2002)*, Bologna, Italy, 15–16 July 2002, Revised Papers, pp. 183–201.
6. Railsback S, Lytinen S and Jackson S. Agent-based simulation platforms: review and development recommendations. *Simulation* 2006; 82(9): 609–623.
7. Baier C and Katoen J-P. *Principles of Model Checking*. Cambridge, MA: The MIT Press, 2008.
8. Gaudel M-C. Testing can be formal, too. In *Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development (TAPSOFT'95)*, London, UK. Berlin: Springer-Verlag, 1995, pp. 82–96.
9. Tretmans J. Model based testing with labelled transition systems. In Hierons RM, Bowen JP and Harman M (eds), *Formal Methods and Testing* (*Lecture Notes in Computer Science*, vol. 4949). Berlin: Springer, 2008, pp. 1–38.
10. da Silva PS and de Melo ACV. On-the-fly verification of discrete event simulations by means of simulation purposes. In *Proceedings of the 2011 Spring Simulation Multiconference (SpringSim'11)*. Philadelphia, PA: The Society for Modeling and Simulation International, 2011.
11. da Silva PS. *Verification of Behaviourist Multi-Agent Systems by means of Formally Guided Simulations*. PhD thesis, Universidade de São Paulo and Université Paris-Sud 11, November 2011. Available at: http://tel.archives-ouvertes.fr/tel-00656809/.
12. Jard C and Jéron T. TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Int J Softw Tools Technol Transf* 2005; 7(4): 297–315.
13. Fernandez J-C, Mounier L, Jard C and Jéron T. On-the-fly verification of finite transition systems. *Form Meth Syst Des* 1992; 1(2–3): 251–273.
14. Bosse T, Jonker CM, van der Meij L, Sharpanskykh A and Treur J. Specification and verification of dynamics in agent models. *Int J Cooperative Inf Syst* 2009; 18(1): 167–193.
15. Bosse T and Gerritsen C. Agent-based simulation of the spatial dynamics of crime: on the interplay between criminal hot spots and reputation. In Padgham L, Parkes DC, Müller JP and Parsons S (eds), *Proceedings of the Seventh International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'08)*, vol. 2. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2008, pp. 1129–1136.
16. Bhargavan K, Gunter CA, Lee I, et al. Verisim: Formal analysis of network simulations. *IEEE Trans Softw Eng* 2002; 28(2): 129–145.
17. Mysore V, Gill O, Daruwala RS, Antoniotti M, Saraswat V and Mishra B. Multi-agent modeling and analysis of the Brazilian food-poisoning scenario. In *Proceedings of Generative Social processes, Models, and Mechanisms Conference*, Argonne National Laboratory and The University of Chicago, 2005.
18. Kim M, Kannan S, Lee I, Sokolsky O and Viswanathan M. Java-MaC: a run-time assurance tool for java programs. In

Runtime Verification 2001 (*Electronic Notes in Theoretical Computer Science*, vol. 55). Amsterdam: Elsevier Science Publishers, 2001.
19. Geilen M. On the construction of monitors for temporal logic properties. In *Runtime Verification 2001* (*Electronic Notes in Theoretical Computer Science*, vol. 55(2)). Amsterdam: Elsevier Science Publishers, 2001.
20. Bauer A, Leucker M and Schallhart C. The good, the bad, and the ugly, but how ugly is ugly? In Sokolsky O and Tasiran S (eds), *Runtime Verification 2007* (*Lecture Notes in Computer Science*, vol. 4839). Berlin: Springer, 2007, pp. 126–138.
21. Lichtenstein O, Pnueli A and Zuck LD. The glory of the past. In *Proceedings of the Conference on Logic of Programs*, London, UK. Berlin: Springer-Verlag, 1985, pp. 196–218.
22. Cousot P and Cousot R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'77)*. New York, NY: ACM Press, 1977, pp. 238–252.
23. Peled D. All from one, one for all: on model checking using representatives. In *Proceedings of the 5th International Conference on Computer Aided Verification (CAV'93)*, London, UK. Berlin: Springer-Verlag, 1993, pp. 409–423.
24. Godefroid P. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. New York: Springer-Verlag, 1996.
25. Biere A, Cimatti A, Clarke EM and Zhu Y. Symbolic model checking without BDDs. In *TACAS*, 1999, pp. 193–207.
26. Biere A, Cimatti A, Clarke EM, Strichman O and Zhu Y. Bounded model checking. *Advances in Computers* 2003; 58: 118–149.
27. Clarke EM, Biere A, Raimi R and Zhu Y. Bounded model checking using satisfiability solving. *Formal Methods in System Design* 2001; 19(1): 7–34.
28. Schruben L. Simulation modeling for analysis. *ACM Trans Model Comput Simul* 2010; 20(1). doi: 10.1145/1667072.1667074.
29. Zeigler BP, Praehofer H and Kim TG. *Theory of Modeling and Simulation* (2nd ed.). New York: Academic Press, 2000.
30. Halim RA and Seck MD. The simulation-based multi-objective evolutionary optimization (SIMEON) framework. In *Proceedings of the 2011 Spring Simulation Multiconference (SpringSim11)*. Philadelphia, PA: The Society for Modeling and Simulation International, 2011.
31. Hwang, M-H. Generating finite-state global behavior of reconfigurable automation systems: DEVS approach. In *Proceedings of the IEEE International Conference on Automation Science and Engineering*. Piscataway, NJ: IEEE Press, 2005.
32. Cormen TH, Leiserson CE, Rivest RL and Stein C. *Introduction to Algorithms* (2nd ed.). Cambridge, MA: The MIT Press, 2001.
33. da Silva PS and de Melo ACV. A formal environment model for multi-agent systems. In Davies J, Silva L and Simao A (eds), *Formal Methods: Foundations and Applications* (*Lecture Notes in Computer Science*, vol. 6527). Heidelberg: Springer, 2011, pp. 64–79.
34. da Silva PS and de Melo ACV. A simulation-oriented formalization for a psychological theory. In Dwyer MB and Lopes A. (Eds.), *FASE 2007 Proceedings* (*Lecture Notes in Computer Science*, vol. 4422). Berlin: Springer-Verlag, 2007, pp. 42–56.
35. Hoare CAR. Communicating sequential processes. *Commun ACM* 1978; 21(8): 666–677.

Appendix A: Analysis of the algorithms

Let us now provide a more rigorous account of whether the algorithms are correct, and of how much resources they employ. To this end, we investigate their soundness, completeness (notions which must be defined) and worst-case complexities.

As noted previously, the presented algorithms are very similar to each other. Owing to this, here we provide a detailed analysis only of Algorithm 1, and then we explain how Algorithm 2 differs.

To reason about soundness and completeness, it is first necessary to establish two things: what entities are to be evaluated; and what is the standard against which this evaluation is to be made? In this work, the entities to be evaluated are Algorithms 1 and 2, and ideally the standard to be used would be the synchronous product $\mathcal{SP} \otimes \mathcal{M}$ that arises from a simulation purpose $\mathcal{SP}$ and an ATS $\mathcal{M}$. That is to say, an algorithm would be considered: *sound* if whenever it issues a `SUCCESS` or `FAILURE` verdict, the desired satisfiability relation holds or does not hold, respectively, in relation to $\mathcal{SP} \otimes \mathcal{M}$; and *complete* if whenever the desired satisfiability relation holds or does not hold with respect to $\mathcal{SP} \otimes \mathcal{M}$, it issues a `SUCCESS` or `FAILURE` verdict.

However, this ideal standard is too strong for the problems considered in this article, which are based on finite simulations on the presence of evolving and autonomous agent behavior. Thus, this ideal notion of completeness is out of the scope of the technique considered here. The same is true for soundness, for the fact that certain outcomes cannot be observed influences the evaluation of satisfiability relations of interest. These issues are inherent to the verification of the satisfiability relations with respect to autonomous systems (i.e. whose behavior cannot be fully predicted) by means of simulations, and not particular to the algorithmic solutions given in the present article.

This difficulty can be addressed in three ways: (i) by removing the autonomy of the systems being investigated; (ii) by making the properties to be verified independent of the autonomous parts of the system; or (iii) by using a weaker standard for soundness and completeness. Options (i) and (ii) are discarded, because autonomy and adaptation, even though they bring a number of problems, are fundamental characteristics of many of the systems we are concerned with (e.g. MASs). Thus, we are left with option (iii).

Let us then introduce the notions of observational soundness and observational completeness. To do so, it is first necessary to define the notion of observed runs.

Definition 16 ($runs^{\mathcal{A}}_{obs}()$ function). *Let $\mathcal{TS}$ be an ATS, and $\mathcal{A}$ an algorithm. Then the set of* observed runs *of $\mathcal{A}$ is denoted by $runs^{\mathcal{A}}_{obs}(\mathcal{TS})$ and such that:*

$$runs^{\mathcal{A}}_{obs}(\mathcal{TS}) \subseteq runs(\mathcal{TS})$$

This denotes the set of the actually encountered runs, among all possible runs.

Definition 17 (Observational soundness). *Let $\mathcal{SP}$ be a simulation purpose and $\mathcal{M}$ be a concrete environment ATS. Then an algorithm is* observationally sound *if whenever it issues:*

- *a* `SUCCESS` *verdict, the desired satisfiability relation holds in relation to $runs^{\mathcal{A}}_{obs}(\mathcal{SP} \otimes \mathcal{M})$;*
- *a* `FAILURE` *verdict, the desired satisfiability relation does not hold in relation to $runs^{\mathcal{A}}_{obs}(\mathcal{SP} \otimes \mathcal{M})$, even if the observed runs could be made longer;*
- *an* `INCONCLUSIVE` *verdict, the desired satisfiability relation does not hold in relation to $runs^{\mathcal{A}}_{obs}(\mathcal{SP} \otimes \mathcal{M})$, but it could perhaps hold if the observed runs could be made longer.*

Definition 18 (Observational completeness). *Let $\mathcal{SP}$ be a simulation purpose, $\mathcal{M}$ be an ATS and l the maximum length of runs in $runs^{\mathcal{A}}_{obs}(\mathcal{SP} \otimes \mathcal{M})$. Then an algorithm is observationally complete up to l if whenever the desired satisfiability relation holds or does not hold with respect to $runs^{\mathcal{A}}_{obs}(\mathcal{SP} \otimes \mathcal{M})$, it issues a* `SUCCESS`*,* `FAILURE` *or* `INCONCLUSIVE` *verdict.*

Each particular simulation execution provides a certain number of observations concerning the actions of the agents. Since one cannot know what all of the possible observations are, the next best thing is to be sound and complete with respect to the observations made. The `INCONCLUSIVE` verdict is added to account for the case in which a mere extension of the observations could have sufficed to find a run of interest.

Algorithms 1 and 2 are designed with this goal in mind. They systematically investigate all of the possible runs that arise from the observations made in the course of exploring the concrete environment ATS up to a certain length. In fact, both Algorithms 1 and 2 are observationally sound, are observationally complete, and terminate. It turns out, moreover, that when Algorithm 1 outputs a `SUCCESS` verdict, this verdict is also sound in the ideal sense, not only the observational one. The same is true with respect to the `FAILURE` verdict for Algorithm 2.

These results are shown to be true in Appendix A.1 (completeness), Appendix A.2 (soundness) and Appendix A.3 (termination). In Appendix A.4 these several analyses are grouped in order to establish the correctness of the algorithms. The worst-case complexities, in turn, are provided in Appendix A.5. In all of the analyses below, definitions, lemmas and propositions are given in the order that they are needed (i.e. bottom-up).

A.1. Justification of completeness

There are two problems that hinder the completeness of the algorithms. The first is that the search tree may have branches of infinite length, which forces the imposition of a maximum depth to guarantee termination, and that is why the `INCONCLUSIVE` verdict exists. The second is that the search can only be made with respect to observed synchronous runs, which may not contain all of the relevant synchronous runs. Nevertheless, both algorithms are still observationally complete up to $depth_{max} + 1$. This is carefully stated and proved below.

A.1.1. Justification of completeness of Algorithm 1

Proposition 2 (Observational completeness of Algorithm 1 up to $depth_{max} + 1$). *If there are feasible runs of length less than or equal to $depth_{max} + 1$ in $runs^{1}_{obs}(\mathcal{SP} \otimes \mathcal{M})$ (i.e. feasibility holds with respect to the observed runs), Algorithm 1 will find a minimal length one among them and return the verdict* `SUCCESS`.

Proof. Let n be the length of an arbitrary minimal length feasible run such that $n \leq depth_{max} + 1$. We prove by induction on n that the algorithm will find some feasible run t such that $|t| = n$.

Base ($n = 2$). Since the initialization phase of the algorithm pushes one tuple on *SynchStack*, and $q_o \overset{f}{\rightsquigarrow} success \in \texttt{Successer}(\mathcal{SP}, q_0)$ for some event f (because, by hypothesis, t is feasible), it follows that line 12 is reached. By the construction of `RemoveBest`, it always removes the transition which composes the shortest path to *Success*, or a transition that leads to *Failure* (in the case of strong feasibility). But in this case we are assuming that the feasible run exists, so the transition chosen is $q_o \overset{f}{\rightsquigarrow} Success$. Finally, since by Definition 8 *Success* synchronizes with any s', it follows that the if block of line 31 is reached. Therefore, the run $t = ((q_0, s_0), f, (Success, s'))$ is found and returned together with the `SUCCESS` verdict in line 32. And t is minimal because there is no feasible run of shorter length, as any run must contain at least q_0 and *Success*.

Induction hypothesis. Let σ be a subrun. If $t = \sigma.(p, s).e.(q, s').f.(Success, s'')$ is a feasible run of minimal length, then $r = \sigma.(p, s).e.(Success, s')$ *would* have been a minimal length feasible run if the transition $p \overset{e}{\rightsquigarrow} Success$ was added to $\mathcal{SP}$.

Inductive step ($n > 2$). By the inductive hypothesis, there could be a minimal length feasible run $r = \sigma.(p, s).e.(Success, s')$ such that $t = \sigma.(p, s).e.(q, s').f.(Success, s'')$ for events e and f, and states (p, s) and (q, s'), if $\mathcal{SP}$ was properly modified. This r could be obtained when the if block of line 31 was reached. In the present case, however, $q' \neq Success$, and as a consequence the algorithm continues running from that point on. Since $progress = true$, the while loop (lines 11–32) will not iterate again, and execution continues from line 8. The algorithm has just pushed a tuple on *SynchStack*, so it is not empty. Indeed, because of line 9, this tuple will be immediately analyzed by the while loop (lines 11–32). As in the induction basis, since `RemoveBest` picks the transition that leads to the shortest path to *Success*, it follows that there is an event f such that $q \overset{f}{\rightsquigarrow} Success$ will be chosen, and therefore the if block of line 31 will execute and return a feasible run together with the `SUCCESS` verdict. Moreover, since $|t| = |r| + 1$ and by hypothesis r was of minimal length, it follows that t is of minimal length.

A.1.2. Justification of completeness of Algorithm 2

Proposition 3 (Observational completeness of Algorithm 2 up to $depth_{max} + 1$). *If there are synchronous runs of maximal length less than or equal to $depth_{max} + 1$ in $runs^1_{obs}(\mathcal{SP} \otimes \mathcal{M})$ such that in their last state (q, s), $q \neq Success$ (i.e. certainty does not hold with respect to the observed runs), then Algorithm 2 will find the shortest among them and return the verdict* `FAILURE`.

Proof. Follows by a proof similar to that of Proposition 2. There are two differences: (i) Algorithm 2 is searching for the *Failure* verdict, and not *Success*; (ii) if no further synchronization is possible at some point, this suffices to provide a run whose last state (q, s) is such that $q \neq Success$. □

A.2. Justification of soundness

In what follows we investigate the soundness properties of Algorithms 1 and 2. This requires first the consideration of the auxiliary procedures that they employ.

A.2.1. Justification of correctness of auxiliary procedures

The verification algorithms depend on an important auxiliary procedure for their initialization, `Preprocess`, and therefore we investigate its correctness first.[x]

To begin, some auxiliary lemmas are required.

Lemma 1 (Correctness of `PreprocessAux()` for acyclic paths). *Let $\mathcal{SP} = \langle Q, E, P, \rightsquigarrow, L, q_0 \rangle$ be a simulation purpose, $source \in Q$, and v a verdict state. Moreover, for every $q \in Q$, let the following initial conditions hold: $dist[q] = nil$ and $visited[q] = false$. Then, after the execution of* `PreprocessAux` *$(\mathcal{SP}, source, v, dist[\,], visited[\,])$ we have that for every $p \in Q$ that is part of an acyclic path of $\mathcal{SP}$, $dist[p]$ will be the minimal distance between p and v.*

Proof. The proof is by induction on the possible distances towards v, denoted by n.

Base ($n = 0$). This can only be the case if $source = v$, which is a (trivial) acyclic path, and in which case the algorithm will correctly attribute 0 to $dist[source]$ in line 3.

Induction hypothesis. Let n be the minimal distance from *source* to v. Then, if *source* is in an acyclic path, there exists a successor q' such that $n > dist[q'] \in \mathbb{N}$ after executing `PreprocessAux`$(\mathcal{SP}, q', v, dist[\,], visited[\,])$.

Inductive step ($n > 0$). If *source* has no successors, it cannot possibly reach v. Hence, $dist[source]$ is made infinite in lines 7 and 19. On the other hand, if there are successors, then each such successor q' that has not yet been visited will be recursively subject to in `PreprocessAux()` line 11. By the induction hypothesis, this implies that if $dist[q'] \in \mathbb{N}$, then $dist[q']$ is the minimal distance from q' to v. Lines 12–16, in turn, select the minimal distance among all such successors and store it in *min*. Since the distance from *source* to any of its successors q' is 1, $dist[source]$ clearly must be $min + 1$, provided that *min* could be calculated at all (i.e. $min \neq nil$). Thus, line 19 correctly attributes the minimal distance to $dist[source]$, if it can be calculated. If, however, $min = nil$, it means, again by the induction hypothesis, that no successor of *source* is in an acyclic path, and therefore neither is *source*. So there is no need to attribute a number to $dist[source]$. □

Lemma 2 (Correctness of `PreprocessAux()` for cyclic paths). *Let $\mathcal{SP} = \langle Q, E, P, \rightsquigarrow, L, q_0 \rangle$ be a simulation purpose, $source \in Q$, and v a verdict state. Moreover, for every $p \in Q$ such that p is in an acyclic path of $\mathcal{SP}$, let the following initial conditions hold: $dist[p] \neq nil$ and $visited[p] = false$. Then, after the execution of* `PreprocessAux`*$(\mathcal{SP}, source, v, dist[\,], visited[\,])$, we have that for every $q \in Q$, $dist[q]$ will be the minimal distance between q and v.*

Proof. The proof is similar to that of Lemma 1. The only difference is that, in the inductive step, we know by hypothesis that for all $q \in Q$ in acyclic paths, $dist[q] \in \mathbb{Z}$. Thus, the only states that remain without an assigned distance are in a cycle. But every cycle eventually reaches a state q' of an acyclic path, so that now there is always a successor q' to *source* such that $dist[q'] \neq nil$ in line 12. Thus, even in the cyclic case, a distance is now assigned to *source*. □

We thus have the following proposition.

Proposition 4 (Correctness of `Preprocess()`). *Let* $\mathcal{SP} = \langle Q, E, P, \rightsquigarrow, L, q_0 \rangle$ *be a simulation purpose and* υ *a verdict state. Then* `Preprocess` $(\mathcal{SP}, \upsilon)$ *returns a function*

$$dist : Q \rightarrow \mathbb{Z}$$

such that, for every $q \in Q$, $dist[q]$ *is the minimal distance between* q *and* υ.

Proof. `Preprocess()` calls `PreprocessAux()` twice. In the first time, by Lemma 1, $dist[q]$ is correctly defined to all $q \in Q$ that belong to an acyclic path of $\mathcal{SP}$. In the second time, by Lemma 2, $dist[p]$ is correctly defined for the remaining $p \in Q$ that belong to cyclic paths of $\mathcal{SP}$.

The correctness of the `BuildRun()` auxiliary procedure will be assessed later, when it becomes necessary. Finally, we note that there is nothing to prove with respect to the `RemoveBest()` auxiliary procedure, since its very definition has the property that is used in later proofs (i.e. the capability of selecting the successor marked with a minimal distance).

A.2.2. Justification of soundness of Algorithm 1

Observational soundness of Algorithm 1 (for weak feasibility) The contents of the *SynchStack* stack of the algorithm is central to this analysis, so let us introduce a related concept and invariant now.

Definition 19 (run_{max}). *Let SynchStack be non-empty and as built by Algorithm 1. Then,* $run_{max}(SynchStack)$ *is a non-empty synchronous run of maximal length stored in SynchStack.*

Definition 20 (Main invariant). *If SynchStack is non-empty, then a non-empty synchronous run* $run_{max}(SynchStack)$ *can always be extracted from it.*

To show the observational soundness of Algorithm 1 for weak feasibility, a number of auxiliary lemmas must be given first. Let us begin by those concerning the invariants of the algorithm.

Lemma 3 (Inner while loop invariant maintenance). *Let* $t = run_{max}(SynchStack)$ *immediately before the inner while loop of lines 15–32,* (q, s) *its last element,* $q \overset{f}{\rightsquigarrow} q'$ *a transition in* $\mathcal{SP}$ *and* $t' = run_{max}(SynchStack)$ *during the loop. Then this loops maintains the following invariant:* t' *exists and either* $|t'| = |t|$ *or* $|t'| = |t| + 1$.

Proof. The only place within the inner while loop in which $run_{max}(SynchStack)$ is modified is in the if block of lines 21–32. This block is only executed if $q \overset{f}{\rightsquigarrow} q'$ synchronize $s \overset{g}{\rightarrow} s'$. By hypothesis, (q, s) is the last element of t. Therefore, the synchronization of these transitions imply that $t' = t.f.(q', s')$, which clearly is a synchronous run. Since the pushing of this synchronization is the only modification performed to *SynchStack*, it follows indeed that $t' = run_{max}(SynchStack)$ exists and that $|t'| = |t|$ (when the if block is not executed) or $|t'| = |t| + 1$ (when the if block is executed). □

Lemma 4 (Middle while loop invariant maintenance). *Let* $t = run_{max}(SynchStack)$ *immediately before the while loop of lines 11–32,* (q, s) *its last state, and* $t' = run_{max}(SynchStack)$ *during the loop. Then this loop makes one of the following statements true:*

- $|t'| = |t| + 1$ and $progress = true$;
- $|t'| = |t|$, $progress = false$ *and SynchStack is not modified*.

Proof. SynchStack is only modified in the inner while loop of lines 15–32, and therefore by Lemma 3 we immediately have the desired result. □

Lemma 5 (Main invariant maintenance). *The outer-most while loop of lines 8–34 maintain the main invariant.*

Proof. By hypothesis, the main invariant holds in the beginning of the loop. *SynchStack* is not changed (only examined) until the while loop of lines 11–32. By Lemma 4, this loop guarantees that after its execution $t = run_{max}(SynchStack)$ will either have its length increased by one or remain the same. In the first case, *SynchStack* is not changed further after this loop, since Lemma 4 guarantees that $progress = true$. Therefore, $run_{max}(SynchStack)$ is the same as before. In the latter case, though, the same Lemma guarantees that $progress = false$ and thus *SynchStack* has its top-most element popped in line 34. This implies that $run_{max}(SynchStack)$ is in a different run, t', such that $|t'| = |t|$ or $|t'| = |t| + 1$. But in both cases it exists, and therefore the main invariant is maintained. □

It is also necessary to show that the algorithm not only maintains the Main Invariant, but also establishes it in its initialization.

Lemma 6 (Initialization). *The algorithm initialization (lines 1 – 7) establishes Main Invariant.*

Proof. SynchStack begins as an empty stack. Since q_0, by Definition 6, is not labelled, it follows by Definition 8 that it synchronizes with s_0. Indeed, q_0 and s_0 are synchronized and pushed on the previously empty *SynchStack*. Hence, after the initialization we have that $run_{max}(SynchStack) = ((q_0, s_0))$. □

The last two lemmas guarantee correct results, provided that the algorithm terminates.

Lemma 7 (Construction of run_{max} by `BuildRun()`) *Let SynchStack be non-empty and as built by Algorithms 1 or 2. Then procedure* `BuildRun()` *can build* $run_{max}(SynchStack)$.

Proof. `BuildRun()` merely starts at the top of *SynchStack* and proceeds downwards in such a way that at every iteration a new element is added to the beginning of the run. This new element is chosen to be one whose *depth* is exactly one less than the previously examined element, and which is, by construction of *SynchStack* (line 29 of Algorithm 1, line 23 of Algorithm 2), an antecedent of this previously examined element. Therefore when the stack becomes empty, the procedure returns a synchronous run. □

Lemma 8 (Correct result upon termination). *Whenever the algorithm terminates, it returns the correct result.*

Proof. There are only two lines in which the algorithm returns, namely, lines 32 and 35.

When line 32 is executed, by Lemma 3 we have that the last element of $run_{max}(SynchStack)$ is q', which by line 31 must be `SUCCESS`. Therefore, $run_{max}(SynchStack)$ is actually a feasible run and by Lemma 7 it will be correctly built by `BuildRun()`. Hence, the returned values are correct.

Concerning line 35, clearly it can only be reached if no feasible run of length less than or equal to $depth_{max}+1$ was found. Owing to Proposition 2, this means that there is no such run, and therefore the result must be either `INCONCLUSIVE` or `FAILURE`.

If at some point it was not possible to synchronize $t = run_{max}(SynchStack)$ because $|t| = depth_{max}+1$, then clearly there could be some run of length greater than $depth_{max}+1$ that was not examined, and therefore the result in this case must be `INCONCLUSIVE`. Indeed, in such a case, line 33 would have set *verdic* = `INCONCLUSIVE` permanently, and therefore the returned result would be correct. On the other hand, if this situation did not arise, then it is plain that there is no possibility of finding a feasible run of any length, and therefore the default setting *verdict* = `FAILURE` is correct. □

Finally, we provide the proposition that guarantees the observational soundness Algorithm 1 for weak feasibility.

Proposition 5 (Observational soundness of Algorithm 1 for weak feasibility). *If Algorithm 1 terminates when checking weak feasibility, then one of the following cases is true.*

- *It returns both* `SUCCESS` *and a weak feasible run* $t \in runs^1_{obs}(\mathcal{SP} \otimes \mathcal{M})$ *such that* $|t| \leq depth_{max}+1$. *Weak feasibility holds with respect to* $runs^1_{obs}(\mathcal{SP} \otimes \mathcal{M})$.
- *It returns* `FAILURE` *and there is no weak feasible run in* $runs^1_{obs}(\mathcal{SP} \otimes \mathcal{M})$. *Weak feasibility does not hold with respect to* $runs^1_{obs}(\mathcal{SP} \otimes \mathcal{M})$, *and no extension of the runs therein would change this.*
- *It returns* `INCONCLUSIVE` *and there is no weak feasible run in* $runs^1_{obs}(\mathcal{SP} \otimes \mathcal{M})$. *Weak feasibility does not hold with respect to* $runs^1_{obs}(\mathcal{SP} \otimes \mathcal{M})$, *but an extension of the runs therein could change this.*

Proof. The algorithm is divided in an initialization part (lines 1–7), an outer-most loop (lines 8–34) and a last return statement (line 35). It suffices then to prove: (i) that the main invariant holds during the entire execution of the algorithm, and in particular during the outer-most loop; (ii) that this invariant is established in the initialization part, before the outer-most loop; and (iii) that if the algorithms terminates, its output complies with what is required by the proposition. Let us begin with (i) and (ii).

Initialization. By Lemma, 6, the initialization part establishes the main invariant.
Invariant maintenance. By Lemma 5, the outer-most loop maintains main invariant. Moreover, the last return statement (line 35) clearly does not violate it.

Finally, by Lemma 8 we have that upon termination it returns the correct result. This establishes (iii). □

Observational soundness of Algorithm 1 (for strong feasibility) The observational soundness with respect to strong feasibility follows from the fact that it suffices to eliminate elements that have transitions that lead to *Failure* from the search stack in order to verify this stronger variant. This merely strengthens the main invariant used to prove Proposition 5, by ensuring that the synchronous runs in the search stack are all strong feasible runs, and thus a similar result holds.

Proposition 6 (Observational soundness of Algorithm 1 for strong feasibility). *If Algorithm 1 terminates when checking strong feasibility, then one of the following cases is true.*

- *It returns both* `SUCCESS` *and a strong feasible run* $t \in runs^1_{obs}(\mathcal{SP} \otimes \mathcal{M})$ *such that* $|t| \leq depth_{max}+1$. *Strong feasibility holds with respect to* $runs^1_{obs}(\mathcal{SP} \otimes \mathcal{M})$.
- *It returns* `FAILURE` *and there is no strong feasible run in* $runs^1_{obs}(\mathcal{SP} \otimes \mathcal{M})$. *Strong feasibility does not hold with respect to* $runs^1_{obs}(\mathcal{SP} \otimes \mathcal{M})$, *and no extension of the runs therein would change this.*
- *It returns* `INCONCLUSIVE` *and there is no strong feasible run in* $runs^1_{obs}(\mathcal{SP} \otimes \mathcal{M})$. *Strong feasibility does not hold with respect to* $runs^1_{obs}(\mathcal{SP} \otimes \mathcal{M})$, *but an extension of the runs therein could change this.*

Proof. It suffices to show in what way the algorithm changes when checking strong feasibility, and how these changes affect the proof of Proposition 5. When checking strong feasibility, the only difference with respect to weak feasibility is that Algorithm 1 will set $dist[Failure] = -1$ (line 2) and, potentially, execute lines 22–25.

The only effect of setting $dist[Failure] = -1$ is that `RemoveBest()` will always return first a $q \overset{f}{\rightsquigarrow} Failure$ if one exists, since by construction no other q' can be such

that $dist[q'] \leq dist[Failure]$. This implies that if during an iteration of the while loop of line 11 there exists a $q \overset{f}{\rightsquigarrow} Failure$ and a $s \overset{g}{\rightarrow} s'$ that synchronize, the first call to *canSynch* will return true and lines 22–25 will be executed.

This execution will: pop *SynchStack*, thus eliminating (q, s) from the synchronous run; and set *progress* and *Succ* in such a way that the algorithm will proceed to the beginning of the outer while in line 8, without performing any further iteration of the other inner while loops, thus effectively ignoring (q, s) and its successors. So whatever synchronous run is stored in *SynchStack*, none of its elements can contain a transition to some $(Failure, s')$ state. This implies that if the algorithm finds a weakly feasible run, it will actually be a strong feasible run.

Finally, since the modifications induced by strong feasibility are very limited and do not compromise the proof of Proposition 5, it follows that similar guarantees hold for strong feasibility verification, but with the difference that they concern strong feasible runs instead of weak feasible runs. □

Special case of the `SUCCESS` verdict in Algorithm 1. As remarked before, the soundness analysis of Algorithm 1 can be strengthened with respect to the `SUCCESS` verdict.

Proposition 7 (Soundness of Algorithm 1 with respect to `SUCCESS`). *If Algorithm 1 ever outputs a `SUCCESS` verdict, $\mathcal{SP}$ is feasible with respect to $\mathcal{M}$ and $runs(\mathcal{SP} \otimes \mathcal{M})$.*

Proof. Algorithm 1 outputs a `SUCCESS` if, and only if, it finds a feasible run r. Since no further observations can change the fact that r exists in $runs(\mathcal{SP} \otimes \mathcal{M})$, it follows that $\mathcal{SP}$ is indeed feasible with respect to $\mathcal{M}$, since otherwise Definition 11 would be violated. □

A.2.3. Justification of soundness of Algorithm 2

Observational soundness of Algorithm 2 Algorithm 2 is merely a slightly modified version of Algorithm 1, which, instead of searching for a feasible run, searches for a run which is not feasible. Hence, the properties and proofs concerning both algorithms are very similar. Below we analyze Algorithm 2 in light of the proofs already given to Algorithm 1. In this way we avoid repeating most of what has already been said, and instead focus on the important differences that must be stressed.

Proposition 8 (Observational soundness of Algorithm 2). *Algorithm 2 terminates and one of the following cases is true.*

- *It returns `SUCCESS`, and all runs of maximal length in $runs^2_{obs}(\mathcal{SP} \otimes \mathcal{M})$ terminate in some state $(success, s)$. Certainty holds with respect to $runs^2_{obs}(\mathcal{SP} \otimes \mathcal{M})$, and no extension of the observed runs therein would change this.*
- *It returns both `FAILURE` and a run of maximal length in $runs^2_{obs}(\mathcal{SP} \otimes \mathcal{M})$ such that it terminates either in a state without a verdict or in some state $(failure, s)$. Certainty does not hold with respect to $runs^2_{obs}(\mathcal{SP} \otimes \mathcal{M})$.*
- *It returns `INCONCLUSIVE`. Certainty does not hold with respect to $runs^2_{obs}(\mathcal{SP} \otimes \mathcal{M})$, but an extension of the observed runs therein could change this.*

Proof. Both Algorithms 2 and 1 are searching for a kind of run. The proof follows by observing the few points in which they differ, from which we can put the former in terms of the latter and then use Proposition 5 to show that the former is sound as well.

By construction, the differences are as follows.

- In line 26, Algorithm 2 finds a run that terminates in some state $(Failure, s)$, whereas in the equivalent place of Algorithm 1 a state $(Success, s)$ is found. Since Algorithm 1 indeed finds such a state if it exists, if follows that Algorithm 2 will also find the state it is searching for if it exists. This establishes that it correctly returns `FAILURE` and an associated run.
- In line 28, Algorithm 2 reaches some state (q, s) such that $q \neq Success$ and $q \neq Failure$, which is indicated by $progress = false$. In Algorithm 1, such a state merely indicates that it is time to try another path in the synchronous product, and thus a check for it is located in line 11. However, in Algorithm 2 this state does indicate a result, that is to say, the fact that there is a run which terminates in a state that has no verdict, an undesirable condition. Like in the point above, then, in this case the algorithm also correctly returns `FAILURE` and a related run.
- In lines 30–31, Algorithm 2 pops from *SynchStack* only after examining all of the contents of the topmost *Unexplored* set. This is similar to what Algorithm 2 does, which pops only if $progress = false$ after trying all elements in *Unexplored*. The difference is that in Algorithm 2 all elements of *Unexplored* are expected to synchronize, and thus it suffices to check whether *Unexplored* is empty after the analysis to pop it, whereas in Algorithm 1 it may be the case that *Unexplored* is not empty and at the same time synchronization is no longer possible.
- If the search does not go beyond the depth $depth_{max}$ and none of the other conditions for `FAILURE` are met, it means that all runs of the synchronous product terminate in some state $(Success, s)$. Algorithm 2 correctly reports this by setting $verdict =$ `SUCCESS` in line 6, which under these conditions remains unmodified until line 32, when it is returned. □

Special case of `FAILURE` verdict in Algorithm 2. As remarked before, the soundness analysis of Algorithm 2 can be strengthened with respect to the `FAILURE` verdict.

Proposition 9 (Soundness of Algorithm 2 with respect to `FAILURE`). *If Algorithm 2 ever outputs a `FAILURE` verdict, $\mathcal{SP}$ is not certain with respect to $\mathcal{M}$ and runs($\mathcal{SP} \otimes \mathcal{M}$).*

Proof. Algorithm 2 outputs a `FAILURE` if, and only if, it finds an refutable run r before reaching the maximum search depth. Since no further observations can change the fact that r exists in $runs(\mathcal{SP} \otimes \mathcal{M})$, it follows that $\mathcal{SP}$ cannot be certain with respect to $\mathcal{M}$, since otherwise Definition 13 would be violated. □

A.3. Justification of termination

Both algorithms terminate, as the following results show.

A.3.1. Justification of termination of Algorithm 1

The following auxiliary lemma is necessary before showing that the algorithm terminates. It argues that, by the construction of the algorithm, similar tuples used in the search can only be pushed a finite amount of times on the search stack.

Lemma 9 (Finite consideration). *Let $(q, s, e, p, sim, Unexplored, depth)$ be an arbitrary tuple on SynchStack of Algorithm 1 at an arbitrary point of its execution. Then, only a finite amount of tuples with the same q, s, e, p, Unexplored and depth can be pushed on SynchStack (let us call this the property of being finitely considered).*

Proof. By induction on *depth*.

Base ($depth = 0$). Because of line 13, $depth'$ is always greater than zero in the outer-most while loop. Thus, the only time in which such a tuple can be pushed is in line 6, which is clearly executed only once, and therefore makes it finitely considered.

Induction hypothesis. The tuples $(q, s, e, p, sim, Unexplored, depth - 1)$ are finitely considered.

Inductive step ($depth > 0$). Take a tuple $tup = (q', s', f, q, sim', unexplored', depth)$ from *SynchStack*. It was either pushed on *SynchStack* after examining some tuple (in line 29) or resulted from a removal of an element from the set of unexplored transitions in a tuple already in *SynchStack* (in line 12).

Suppose that *tup* was pushed on *SynchStack*. Then there is a tuple tup_{pre} of depth $k = depth - 1$ that was peeked to generate *tup*. tup_{pre} can only be peeked once in line 9, because in line 12 one of its components is permanently modified (i.e. a transition is removed from its set of unexplored transitions). So once tup_{pre} is peeked, *tup* is calculated and pushed on *SynchStack* once. Since by hypothesis tuples marked with depth $k = depth - 1$ are finitely considered, this can only be repeated a finite amount of times. Moreover, by construction, if there are other $tup'_{pre} \neq tup_{pre}$ that can be used to generate *tup*, the only difference in tup'_{pre} with respect to tup_{pre} has to be the state s of $\mathcal{M}$. But because of the finite branching of $\mathcal{M}$, there are only finitely many such states up to $depth - 1$, so in all *tup* can only be pushed on *SynchStack* a finite amount of times, and is therefore finitely considered.

Suppose, on the other hand, that *tup* was obtained by reducing the set of unexplored transitions of another tuple already on *SynchStack*. In this case, clearly $unexplored' \neq$ `Successors` $(\mathcal{SP}, q')$. But tuples of depth greater than zero can be pushed only in line 29, in which, by construction, the equality $unexplored' \neq$ `Successors` $(\mathcal{SP}, q')$ must hold. So *tup* has never been pushed on *SynchStack*, and thus is finitely considered. □

The following proposition can now be proved.

Proposition 10 (Termination of Algorithm 1). *Algorithm 1 terminates.*

Proof. Termination follows from the fact that each state of $\mathcal{SP}$ is examined only a finite amount of times, after which either a feasible run is found and the algorithm terminates, or synchronizations are no longer possible and thus *SynchStack* becomes empty, thereby achieving termination as well. The former case arises if indeed there are feasible runs, because by Proposition 2 one of them will be found, and since this implies the execution of the return statement in line 32, the algorithm terminates. The latter case, however, require some more analysis.

Since the number of states in $\mathcal{SP}$ is finite, the possible search depths are finite (for $depth_{max}$ is finite), and the amount of events are also finite, there are only finitely many events g, sets of $unexplored'$ and natural numbers $depth'$ to be considered in line 29. Furthermore, although $\mathcal{M}$ may have an infinite number of states, for any finite depth there is a corresponding finite amount of reachable states s' (and, thus, of simulation states sim'), owing to the property of finite branching of ATSs. So there are only finitely many tuples to be considered in line 29. Moreover, by Lemma 9 all pushed tuples are finitely considered, which implies that each of these finitely many tuples can be pushed on *SynchStack* at most a finite amount of times. Hence, the size of *SynchStack* is bounded by a finite quantity.

Every time a tuple $(q, s, e, p, sim, Unexplored, depth)$ is chosen in line 9, *Unexplored* can only decrease, since the only statement that modifies it is the call to `RemoveBest()` of line 12. So for any set *Unexplored*, it will decrease until it reaches size zero and is popped. That this will indeed happen follows from these cases for the guard in line 11.

- If $Unexplored = \emptyset$, we have that $progress = false$, and in line 34 the tuple will be popped.

- If $Unexplored = \emptyset$, but $depth \geq depth_{max}$, it will be again the case that $progress = false$ and in line 34 the tuple will be popped.
- Otherwise, the while loop begins and `RemoveBest()` of line 12 is executed, thereby reducing its size.

Now, since there can be only finitely many tuples on *SynchStack*, and that each is guaranteed to be popped eventually, it follows that the outermost while loop eventually terminates, thus establishing that the algorithm itself terminates. □

A.3.2. Justification of termination of Algorithm 2

A similar result holds for Algorithm 2.

Proposition 11 (Termination of Algorithm 2). *Algorithm 2 terminates.*

Proof. Similar to the proof of Proposition 10, but taking in account the fact that Algorithm 2 is searching for a refutable run, and not a feasible run. □

A.4. Justification of correctness

Based on the results shown, we can now formalize their correctness with the following propositions.

Proposition 12 (Correctness of Algorithm 1). *Algorithm 1:*

- *is observationally sound for both weak and strong feasibility;*
- *is observationally complete up to* $depth_{max} + 1$ *for both weak and strong feasibility*;
- *gives a sound verdict in the classical sense (i.e. with respect to runs*$(\mathcal{SP} \otimes \mathcal{M})$*) whenever it outputs the* `SUCCESS` *verdict*;
- *terminates*.

Proof. Follows directly from Propositions 5, 6, 7, 2 and 10.

Proposition 13 (Correctness of Algorithm 2). *Algorithm 2:*

- *is observationally sound;*
- *is observationally complete up to* $depth_{max} + 1$*;*
- *gives a sound verdict in the classical sense (i.e. with respect to runs*$(\mathcal{SP} \otimes \mathcal{M})$*) whenever it outputs the* `FAILURE` *verdict;*
- *terminates.*

Proof. Follows directly from Propositions 8, 9, 3 and 11. □

A.5. Justification of worst-case complexities

Let us now provide the exact worst-case complexities of the auxiliary procedures and of the verification algorithms themselves. Owing to their similarity, Algorithms 1 and 2 actually have the same such complexities, and this follows from the same proof.

A.5.1. Worst-case complexities of auxiliary procedures

Lemma 10 (Worst-case time complexity of `Preprocess()`). *Let* $\mathcal{SP} = \langle Q, E, P, \rightsquigarrow, L, q_0 \rangle$ *be a simulation purpose. Then* `Preprocess()` *runs in* $O(|Q|^2)$ *time.*

Proof. In `Preprocess()` itself, it is clear that each state in Q must be visited twice for initializations, and that the rest of the running time is given by two invocations of the `PreprocessAux()` subprocedure. Because to each such state initialization a constant amount of operations is performed, these initializations together take time proportional to $m_0 = 2 \cdot |Q|$. `PreprocessAux()`, in turn, will recursively visit each state in Q accessible from q_0, which in the worst case are all of the states of Q. In each such visit, a state *source* is specified, all of its successors q' may be examined in order to determine whether they have or have not been visited yet, and a maximum constant amount of other operations are performed, without taking in account the recursive call. Since in the worst case there may be $|Q|$ successors, each such visit to a state takes time proportional to $m_1 = |Q|$,

In considering the successors q', two cases may arise. If q' has already been visited, then $visited[q'] = true$ (because this is the first thing `PreprocessAux()` sets when visiting a state), and no recursive call happens. On the other hand, if q' has not been visited, then $visited[q'] = false$ and a recursive call to `PreprocessAux()` takes place in line 11. Since in each visit the state is marked as visited, there can be at most $|Q|$ such recursive calls, hence at most $m_2 = |Q| + 1$ visited states (counting the initial visit of q_0).

Therefore, in all, `Preprocess()` has a time complexity of $O(m_0 + 2m_1 \cdot m_2) = O(2 \cdot |Q| + 2 \cdot |Q| \cdot |Q|) = O(|Q|^2)$. □

Lemma 11 (Worst-case space complexity of `Preprocess ()`). *Let* $\mathcal{SP} = \langle Q, E, P, \rightsquigarrow, L, q_0 \rangle$ *be a simulation purpose. Then* `Preprocess()` *consumes* $O(|Q|)$ *memory.*

Proof. This follows from two observations.

- The only data structures employed, $dist[]$ and $visited[]$, each consumes memory proportional to $|Q|$ by construction. This can be seen by noticing

that in these data structures only elements of Q are used as keys, and only numeric or Boolean elements are used as values.
- The recursive calls of line 11 of `PreprocessAux()` cannot induce a call stack larger than $|Q|$, since each state is visited at most once.

Therefore, in all, $O(2 \cdot |Q|) = O(|Q|)$ space is used. □

A.5.2. Worst-case complexities of Algorithms 1 and 2

The complexities of Algorithms 1 and 2 must be given considering the fact that they perform a depth-first search on a transition system with possibly infinitely many states that is built on-the-fly (i.e. $\mathcal{SP} \otimes \mathcal{M}$), without keeping track of the visited states, and only up to a maximum depth (i.e. $depth_{max}$). This means that the complexities must be given mainly in terms of $depth_{max}$ and the maximum branching factor (i.e. the maximum number of possible successors of any state).

The maximum branching factor of $\mathcal{SP} \otimes \mathcal{M}$ is calculated as follows.

Proposition 14 (Branching factor). *Let $\mathcal{SP} = \langle Q, E_{sp}, P_{sp}, \rightsquigarrow, L_{sp}, q_0 \rangle$ be a simulation purpose, and $\mathcal{M}$ an ATS in which any state has at most m successors. Then the branching factor (i.e. the maximum number of possible successors of any state) of $\mathcal{SP} \otimes \mathcal{M}$ is $O(|Q| \cdot |E_{sp}| \cdot m)$.*

Proof. Take an arbitrary state (q, s) of $\mathcal{SP} \otimes \mathcal{M}$. Let us first consider the number of possible transitions $q \overset{f}{\rightsquigarrow} q'$ in $\mathcal{SP}$. Certainly this number is less than or equal to $|Q| \cdot |E_{sp}|$, which corresponds to all possible such transitions from q. In the worst case, each of these transitions could synchronize with the m transitions from the states of $\mathcal{M}$, thereby generating a branch factor of $|Q| \cdot |E_{sp}| \cdot m$. Thus, the actual branching factor must be $O(|Q| \cdot |E_{sp}| \cdot m)$. □

It is now possible to give the complexities in space and time.

Proposition 15 (Worst-case space complexity of Algorithms 1 and 2). *Let $\mathcal{M}$ be the an ATS in which each state has a size less than or equal to k. Then, in the worst case, Algorithms 1 and 2 consume $O(|Q| \cdot |E_{sp}| \cdot m \cdot k \cdot depth_{max})$ space.*

Proof. Since the algorithm performs a depth-first search in a graph that is built on-the-fly and only up to a maximum depth (i.e. $depth_{max}$), it follows that the space complexity is given by the maximum size that the search stack may attain. This size, in turn, is given by the amount and size of elements that are put on the stack at each depth. The amount of these elements is no more than the maximum branching factor b, since it is a depth-first search. And the size of each element is proportional to k, since all that these elements contain is proportional to the current state of $\mathcal{M}$. Hence, the size of the search stack is less than or equal to $c_1 \cdot b \cdot k \cdot depth_{max}$, for some constant c_1.

By Proposition 14, $b \leq c_2 \cdot |Q| \cdot |E_{sp}| \cdot m$, for some constant c_2. Therefore, the size of the search stack is bounded by $c_1 \cdot c_2 \cdot |Q| \cdot |E_{sp}| \cdot m \cdot depth_{max}$, hence $O(|Q| \cdot |E_{sp}| \cdot m \cdot depth_{max})$. □

Proposition 16 (Worst-Case Time Complexity of Algorithm 1). *Let $\mathcal{SP} = \langle Q, E_{sp}, P_{sp}, \rightsquigarrow, L_{sp}, q_0 \rangle$ be a simulation purpose, $\mathcal{M} = \langle S, E, P, \rightarrow, L, s_0 \rangle$ an ATS, and $b = |Q| \cdot |E_{sp}| \cdot m$. Then, in the worst case, Algorithms 1 and 2 have $O(b^{depth_{max}})$ running time.*

Proof. At each new depth reached by the search, no more than b elements are put on the search stack, where b is the maximum branching factor. Since in the worst case all elements reached up to a maximum depth (i.e. $depth_{max}$) will be examined, it follows that the total quantity of examined elements is bounded by $c \cdot b^{depth_{max}}$, for some constant c. This results in $O(b^{depth_{max}})$ running time. □

These analyses concern only the verification procedure, and ignore the internal structure of the states in $\mathcal{M}$. In actual applications, this may be significant. In the case of the MASs we studied in da Silva,[11] for example, the maximum number m of successors of states in $\mathcal{M}$ is $O(n^2)$, where n is the size of the MAS environment under consideration. As in this case, other applications must refine the analysis of complexities to account for the inner working of the simulation model.

Author biographies

Paulo Salem da Silva received his BSc in computer science from the University of São Paulo (Brazil) and holds a PhD in the same area from both the University of São Paulo and the University Paris-Sud (France). His research interests center mostly on modeling and automated analysis of systems, which includes formal methods, multi-agent systems and programming language design. He is also an entrepreneur and is currently working independently on commercial World Wide Web technologies.

Ana CV de Melo received the BS and MS degrees in computer science from the Federal University of Pernambuco (Brazil) and PhD degree from the University of Manchester (UK), and was a postdoctoral fellow at the University of Oxford (UK). She is currently an associate professor at the University of São Paulo (Brazil). Her main research interests include static and dynamic analyses and coordination of concurrent systems through formal means to improve software quality and compositional behavior of systems, including applications to multi-agent systems, web services and business rules.