On-The-Fly Verification of Discrete Event Simulations by Means of Simulation Purposes

Paulo Salem da Silva and Ana C. V. de Melo
University of São Paulo
Department of Computer Science
São Paulo – Brazil
salem@ime.usp.br acvm@ime.usp.br

Keywords: formal method, approximate verification, testing, experimentation.

Abstract

Discrete event simulations can be used to analyse natural and artificial phenomena. To this end, one provides models whose behaviours 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 in modelling the simulation as a 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. The precise nature of simulation purposes, as well as the corresponding verification algorithm, are largely determined by methodological considerations important for simulations.

1. INTRODUCTION

Discrete event simulations can be used to analyse natural phenomena. To this end, one provides models whose behaviours are characterized by discrete events in a discrete timeline. By running such a simulation, one can then observe its properties.[1] Our particular interest is in the simulation of multi-agent systems, in which the simulation model is a set of *agents* who interact within an *environment* [6]. A number of tools exist for performing such simulations (e.g., [14; 10]), and the need for methods for analysing them has already been recognized (e.g., by [11]). However, compilation of statistics over multiple simulation runs is often the only implemented and imagined analysis mechanism, and formal verification is seldom considered.

We believe that 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.

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, 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 correction 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. This construction, moreover, assumes the existence of a simulator with a certain interface, which is used to connect the formal analyses to the actual simulation. This interface is very simple, and therefore can be easily incorporated into existing simulators.

There are several kinds of analyses that one could perform in this manner. In this presentation, however, we shall only consider the task of determining whether there exists a simulation that conforms to a specified simulation purpose. This reduces to searching for a particular path in the relevant synchronous product, which can be done through a depth-first search algorithm using a linear (in the depth) amount of memory.

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 transition systems, including the simulation purposes. Section 5. defines the synchronous product and the relevant criterion of correctness. Section 6., then, provides the verification algorithm. Section 7. gives a concrete example of how the ap-

[1] Notice that by "simulation" we do not mean the formal relation among two transition systems, such as what [12] 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.

proach can be useful. At last, Section 8. concludes. Appendix A provides, as ancillary material, some auxiliary procedures (and related explanations) used by the verification algorithm.

2. RELATED WORK

The inspiration for this work comes from the area of Model-Based Testing, specially from the approach used by TGV [7] 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 [16]. 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 [4].

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 paper 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 labelled states (and not only transitions), and that simulation purposes need not be input complete.

As we pointed out in the introduction, runtime verification of simulations are not usually done. Two notable exceptions are the network simulator Verisim [3] and a multi-agent modelling of food poisoning [13]. The approach taken by these works consists in running the simulation normally, but checking linear-time properties in the resulting execution traces. This kind of approach can be implemented by the usual runtime verification notion of a *monitor*, which is an extra component that is added to the system in order to perform the verification. Verisim [3], for instance, employs the MaC architecture [8] to provide such a monitor for its simulations. The precise nature of monitors vary according to the kind of property to be analysed. 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 logics, 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 by a number of works [5; 2; 9].

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 of expressing both success and failure in the same structure is another important point. Other requirements are examined in Section 3. below.

3. METHODOLOGICAL REQUIREMENTS

The formal approach that we present in the following sections is justified by the objective to which they should be applied, namely, the 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.

We are interested in simulation models of multi-agent systems. 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. As a working example, let us consider a model of an online social network, where several persons exist and can interact with each other through the features of a website.[2] Clearly, the behaviour of each individual person is likely to be very complex, and if a model is given to them, it probably won't be a simple one (e.g., see [15]). But the environment, on the other hand, can be described by some formalism that merely define relations among agents (e.g., a process algebra such as the π-calculus [12]), 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.

Notice 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's 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*;

[2]Current examples of such networks include popular websites such as `www.facebbok.com`, `www.orkut.com` and `www.myspace.com`.

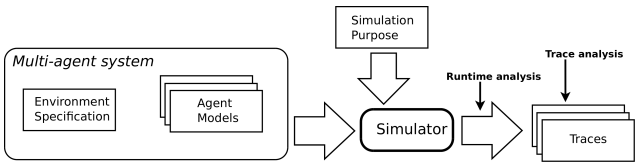

Figure 1. Overview of the verification architecture based on simulations of multi-agent systems. The simulator takes two inputs: (i) a multi-agent system, composed by agent models and an environment specification; (ii) a *simulation purpose* to be verified. The simulator then produces traces as outputs. Verification can be done at the simulation runtime level, as well as at the trace level (if the traces are recorded). In this paper, we only consider runtime verification, since we believe this is more fruitful.

- 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 require – 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 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 artefacts as transition systems.

In this way, the technique we present in this paper should be seen as a method to automate experiments undertaken employing simulations of either natural or artificial phenomena.

4. SEMANTIC MODEL

To formally describe the systems of interest (e.g., the environments presented in Section 3.), we define *annotated transition systems (ATSs)*, which essentially are transition systems with labels given to both states and transitions.[3] The properties to be checked, in turn, are given by *simulation purposes* (SPs), which are merely ATSs subject to some further restrictions.

In an ATS, events play a central role, and are further divided into *input events* (e.g., $?x$) and *output events* (e.g., $!x$). 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 – behaviour of the simulator). These events are complementary, and for a given event e we denote its complement by e^C. There are also two special events: (i) one called *other* and denoted by $\circledast$, which is a convenience to allow the definition of the set of events that have not been specified in a given state (i.e., $\circledast$ signals that something not specified happened).[4]; and (ii) another called *internal*, denoted by τ, which accounts for hidden (and thus anonymous) events.

Definition 1 (Annotated Transition System) *An* annotated transition system (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 primitive events.*
- P *is the finite set of primitive* propositions.
- $\rightarrow : S \times E \times S$ *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* labelling function.[5]
- *For all* $s \in S$ *and all* $p \in P$*, if* $p \in L(s)$*, then* $\neg p \notin L(s)$ *(i.e., the labelling function is consistent).*
- $s_0 \in S$ *is the* initial state.

[3] Our definition of ATS is very similar to what is merely called a transition system in [1]. We think, however, that it is worth to emphasize 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 Labelled Transition System (LTS) [12], in which only transitions are labelled. We also impose certain criteria of finiteness.

[4] As a technicality, we define that $\circledast^C = \circledast$.

[5] 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 $\{\neg p \mid p \in P\}$. This notation shall be used throughout the text.

Notice, in particular, that the labelling function associates literals[6], and not merely propositions, to the states. This allows the 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 modelling situations in which the truth value of a proposition cannot be assessed, as may it happen in experimental situations.

Thus, 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 *trace*.

A *simulation purpose*, in turn, is an ATS subject to a number of restrictions. In particular, it defines states to indicate either success or failure of the verification procedure (i.e., *verdict states*), and is deterministic to avoid contradictory verdicts.

Definition 2 (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) $L(q_0) = L(Success) = L(Failure) = \emptyset$.

(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., transitions are* deterministic*).*

(vi) For every $q \in Q$*, there exists a trace from* q *to either Success or Failure.*

A visual depiction of both a general ATS and a simulation purpose is given in Figure 2.

5. VERIFICATION TASK

We have defined both a way to describe a system of interest and the properties we wish to verify over such a system. Now we may describe precisely in what this verification consists.

The idea that a simulation purpose can select which traces to consider in another ATS is formalized by the notion of synchronization. Since both events and states contain relevant information for this selection, a particular definition for each case is first required, as follows.

[6]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*.

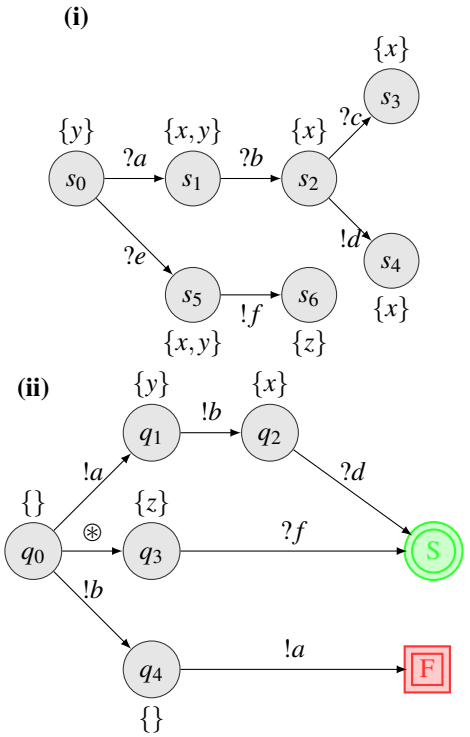

Figure 2. Examples of: (i) an ATS; and (ii) a simulation purpose. Transitions are annotated with events (i.e., $?a$, $?b$, $?c$, $?d$, $?e$, $?f$, $!a$, $!b$, $!c$, $!d$, $!e$, $!f$) and states are annotated with literals (i.e., x, y, z). The *Success* state is denoted by the green double circle, while the *Failure* state is denoted by the red double square.

Definition 3 (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*

$s_1 \overset{e_2}{\rightarrow} s_2$ *be a transition from* $\mathcal{M}$.

Then we define that events e_1 *and* e_2 synchronize*, denoted by* $e_1 \bowtie e_2$*, if, and only if, one of the following cases holds:*

- $e_1 = ?n$ *and* $e_2 = !n$ *for some name* n*; or*
- $e_1 = !n$ *and* $e_2 = ?n$ *for some name* n*; or*
- $e_1 \neq \circledast$ *and* $e_2 = \circledast$*; or*
- $e_1 = \circledast$ *and there is no* $q' \in Q$ *such that* $q_1 \overset{e_2^C}{\rightsquigarrow} q'$*; or*
- $e_1 = e_2 = \tau$.

Definition 4 (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*, denoted by* $q \bowtie s$*, if, and only if,*

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

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

Definition 5 (Synchronous Product) *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 Environment 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:*

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

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

In the example of Figure 2, only the trace $(s_0, ?a, s_1, ?b, s_2, !d, s_4)$ would belong to the product of (i) and (ii).

Given this product, there are a number of properties that one might wish to analyse about it. In this paper, we shall merely consider the question of whether the simulation purpose is capable of conducting to a state of *success*. 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. We formalize this with the following notion of *feasibility*.

Definition 6 (Feasibility) *Let* $\mathcal{SP}$ *be a simulation purpose and* $\mathcal{M}$ *be an ATS. Then we define that* $\mathcal{SP}$ *is* feasible *with respect to* $\mathcal{M}$ *if, and only if, for some state* (q,s) *in* $\mathcal{SP} \otimes \mathcal{M}$*,* $q =$ *Success. Otherwise, we call it* unfeasible*. Moreover, the trace that leads to Success is called* feasible trace*.*

6. VERIFICATION ALGORITHM

We now present an on-the-fly algorithm to check the feasibility property defined previously. To do so, we first introduce a simulator interface, which abstracts the features of a simulator. We then present the algorithm itself, which employs this interface, and analyse its properties.

6.1. Simulator Interface

In order to work with our verification algorithm, a simulator must provide the following operations:

- `GoToState(`s`)`: Makes the simulation run return to the specified simulation state s, which must have taken place previously.
- `CurrentState()`: Returns the current simulation state.
- `CanStep(`e`)`: Checks whether the specified event e can be performed by the simulator. Note that this allows the simulator to influence the verification algorithm.
- `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. If the simulator does not employ any such clock, this operation always returns *true*.

6.2. Feasibility Verification

Algorithm 1 implements feasibility verification. Besides the simulator operations described above, it also assumes that the following simple functions and constants are available:

- `CanSynch(`$q \overset{f}{\rightsquigarrow} q'$, $s \overset{g}{\rightarrow} s'$`)`: Checks whether the two specified transitions can synchronize according to Definition 5.
- `Successors(`*ATS, s*`)`: Calculates the set of all transitions in the specified *ATS* that have the state `s` as their origin.
- $depth_{max}$: 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 as well.

The remaining procedures required for the algorithm are given in Appendix A.

Algorithm 1: On-the-fly verification of feasibility

Input: A simulation purpose $\mathcal{SP} = \langle Q, E_{sp}, P_{sp}, \rightsquigarrow, L_{sp}, q_0 \rangle$ and an ATS $\mathcal{M} = \langle S, E, P, \rightarrow, L, s_0 \rangle$.

Output: `SUCCESS` and a feasible trace in $\mathcal{SP}$ if it is possible to show that $\mathcal{SP}$ is feasible over $\mathcal{M}$; `FAILURE` if no such feasible trace was found after trying all relevant synchronizations; `INCONCLUSIVE` if there was not sufficient resources to try all relevant synchronizations.

```
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 := FAILURE;
7  while SynchStack ≠ ∅ do
8      Peek (q, s, e, p, sim, Unexplored, depth) from
       SynchStack;
9      let progress := false;
10     while Unexplored ≠ ∅ ∧ progress =
       false ∧ depth < depth_max do
11         q -f-> q' := RemoveBest(Unexplored, dist[ ]);
12         let depth' = depth + 1;
13         foreach s -g-> s' ∈ Successors(M, s) do
14             GoToState(sim);
15             ScheduleStep(g);
16             if isCommitEvent(g) then
17                 Step();
18             if CanSynch(q -f-> q', s -g-> s') then
19                 sim' := CurrentState();
20                 unexplored' := Successors(SP, q');
21                 Push
                   (q', s', f, q, sim', unexplored', depth')
                   on SynchStack;
22                 progress := true;
23                 if q' = success then
24                     return SUCCESS and
                       BuildTrace(SynchStack, q',
                       depth');
25     if depth ≥ depth_max then
26         verdict := INCONCLUSIVE;
27     if progress = false then
28         Pop from SynchStack;
29 return verdict;
```

How the Algorithm Works First of all, a preprocessing of the simulation purpose is required. This consists in calculating how far from the desired verdict state each of the states in the simulation purpose is. By this provision, we shall be able to take the shortest route from any given simulation purpose state towards the desired verdict. 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 $dist[q]$.[7]

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 $\mathcal{SP}$ that synchronized. This will be used later to calculate a trace.
- 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 trace.
- 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

[7] Appendix A explains how this preprocessing is actually performed.

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 the search to try other paths which might lead to a feasible trace, instead 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 possible transition $s \xrightarrow{g} s'$, 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 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 $s \xrightarrow{g} s'$ can synchronize. If it is possible, then we either have the desired *success* state or we don't. In the first case we have found the feasible trace we were looking for and we are done. In the latter case, we merely push the current state of affairs on *SynchStack* for later analysis and register the successful synchronization by setting *progress* to *true*.

If the algorithm abandons a search branch because its depth is greater 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 trace with length greater than $depth_{max}$ 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 trace will require it.

At last, if *SynchStack* becomes empty, it means that no path in $\mathcal{SP}$ with up to $depth_{max}$ states led to a feasible trace. So we return a verdict which will be `FAILURE` if $depth_{max}$ was never reached, or `INCONCLUSIVE` otherwise.

Termination and Efficiency The algorithm is essentially a depth-first search. Since by hypothesis $depth_{max}$ is finite, each search branch will be finite. Moreover, since $\mathcal{SP}$ contains a finite number of states and events, and is used to choose the search branches, it follows that there are only finitely many such branches. Therefore, Algorithm 1 always terminates.

Concerning efficiency, since the search depth is bounded by $depth_{max}$, it should be clear that it consumes $O(b \cdot depth_{max})$ memory, where b is the maximal branching factor of $\mathcal{M}$ (i.e., the maximal number of successors that a state of $\mathcal{M}$ may have). Running time, however, is much worse. Let us suppose the worst case and say that $q \overset{f}{\rightsquigarrow} q'$ and $s \xrightarrow{g} s'$ can always synchronize. Then, for each point in a trace of the synchronous product, we shall have no more than either $|Q| \cdot b$ (for states) or $|E_{sp}| \cdot |E|$ (for events) possibilities. Let m be the greater of these two quantities. This results in $O(m^{depth_{max}})$ running time. Nonetheless, by crafting simulation purposes well, one may find practical ways to improve performance (e.g., by being as restrictive as possible).

7. EXAMPLE

Recall from Section 3. that we are interested in the simulation and verification of multi-agent systems. Let us then continue the example suggested there, namely, that of modelling and analysing 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 ones:

- $!gui_i$: 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.

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.

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 3. Notice that the event $?msg_{ag_1}^{ag_2}$ is not present in this simulation purpose, although it's probably a frequent event in

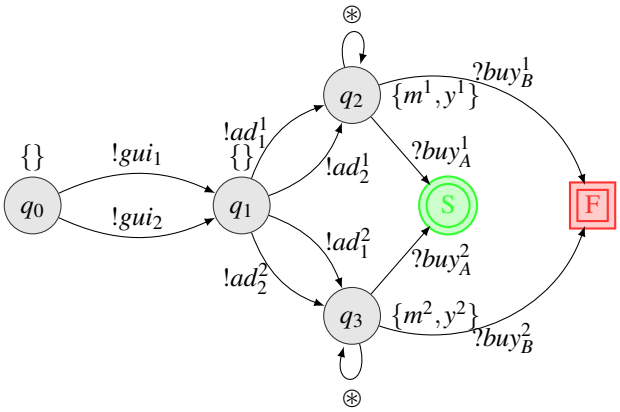

Figure 3. The example's simulation purpose. We assume only two agents for this example, and only one competing product, B. A possible trace could be as follows: we implement GUI 1 ($q_0 \overset{!gui_1}{\rightsquigarrow} q_1$), deliver advertisement 1 to agent 2 ($q_1 \overset{!ad_1^2}{\rightsquigarrow} q_3$), and then realize that agent 2 indeed buys product A ($q_3 \overset{?buy_A^2}{\rightsquigarrow} success$).

such a social network. 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 strategies from causing damage (e.g., by submitting real users to an unbearable amount of advertisement).

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 analyses. 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.

8. CONCLUSION

In this work, we have shown a way to perform an on-the-fly product of two annotated transition systems, $\mathcal{SP}$ and $\mathcal{M}$, in order to verify properties of interest. $\mathcal{M}$ is a model of the system to be verified. $\mathcal{SP}$ formalizes the property to be verified, is subject to some extra conditions, and for this reason receives the name of simulation purpose. The verification algorithm consist in using $\mathcal{SP}$ in order to partially explore $\mathcal{M}$. By this method, we allow the approximate verification of systems arising from discrete event simulation and, more specifically, the simulation of multi-agent systems, in which environments can be effectively represented as ATSs.

For brevity, we have considered only verifications that seek one feasible path. It should be clear, though, that one may adapt the provided approach in order to check other properties. For instance, one may be interested in whether all paths of a synchronous product are feasible.

We are experimenting with the proposed method in our own simulator, but it should also be possible to incorporate it in most other existing platforms, since the required simulator interface is very simple.

ACKNOWLEDGEMENTS

The authors would like to thank Prof. Dr. Marie-Claude Gaudel, from University Paris-Sud 11, for crucial suggestions that led to the present work.

This project benefited from the financial support of *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior* (CAPES) and *Conselho Nacional de Desenvolvimento Científico e Tecnológico* (CNPq).

REFERENCES

[1] Baier, Christel, and Joost-Pieter Katoen. 2008. *Principles of model checking*. The MIT Press.

[2] Bauer, Andreas, Martin Leucker, and Christian Schallhart. 2007. "The good, the bad, and the ugly, but how ugly is ugly?". In *RV*, ed. Oleg Sokolsky and Serdar Tasiran, vol. 4839 of *Lecture Notes in Computer Science*, 126–138. Springer.

[3] Bhargavan, Karthikeyan, Carl A. Gunter, Insup Lee, Oleg Sokolsky, Moonjoo Kim, Davor Obradovic, and

Mahesh Viswanathan. 2002. “Verisim: Formal analysis of network simulations”. *IEEE Trans. Softw. Eng.* 28(2): 129–145.

[4] Fernandez, Jean-Claude, Laurent Mounier, Claude Jard, and Thierry Jéron. 1992. “On-the-fly verification of finite transition systems”. *Form. Methods Syst. Des.* 1(2-3):251–273.

[5] Geilen, Marc. 2001. “On the construction of monitors for temporal logic properties”. *Electr. Notes Theor. Comput. Sci.* 55(2).

[6] Gilbert, Nigel, and Steven Bankers. 2002. “Platforms and methods for agent-based modeling”. *Proceedings of the National Academy of Sciences of the United States* 99(Supplement 3).

[7] Jard, Claude, and Thierry Jéron. 2005. “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.* 7(4):297–315.

[8] Kim, M., S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. 2001. “Java-MaC: a run-time assurance tool for java programs”. In *Runtime verification 2001 proceedings*, vol. 55 of *ENTCS*. Elsevier Science Publishers.

[9] Lichtenstein, Orna, Amir Pnueli, and Lenore D. Zuck. 1985. “The glory of the past”. In *Proceedings of the conference on logic of programs*, 196–218. London, UK: Springer-Verlag.

[10] Luke, Sean, Claudio Cioffi-Revilla, Liviu Panait, and Keith Sullivan. 2004. “MASON: A new multi-agent simulation toolkit”. Http://cs.gmu.edu/ eclab/projects/mason/.

[11] Marietto, Maria B., Nuno David, Jaime S. Sichman, and Helder Coelho. 2003. “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, July 15-16, 2002. revised papers*, 183–201.

[12] Milner, Robin. 1999. *Communicating and mobile systems: the π-calculus*. New York, NY, USA: Cambridge University Press.

[13] Mysore V., Gill O., Daruwala R.S., Antoniotti M., Saraswat V.and, and Mishra B. 2005. “Multi-agent modeling and analysis of the brazilian food-poisoning scenario”. In *Proceedings of generative social processes, models, and mechanisms conference – argonne national laboratory & the university of chicago*.

[14] North, Michael, Nick Collier, and Jerry R. Vos. 2006. “Experiences creating three implementations of the Repast agent modeling toolkit”. *ACM Transactions on Modeling and Computer Simulation* 16(1):1–25. Http://repast.sourceforge.net/.

[15] da Silva, Paulo Salem, and Ana C. V. de Melo. 2007. “A simulation-oriented formalization for a psychological theory”. In *Fase 2007 proceedings*, ed. Matthew B. Dwyer and Antonia Lopes, vol. 4422 of *Lecture Notes in Computer Science*, 42–56. Springer-Verlag.

[16] Tretmans, Jan. 2008. “Model based testing with labelled transition systems”. In *Formal methods and testing*, ed. Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, vol. 4949 of *Lecture Notes in Computer Science*, 1–38. Springer.

A ANCILLARY MATERIAL

This appendix presents the auxiliary procedures used by Algorithm 1 that were not considered in the main text.

Preprocessing Procedure: Handling Cycles

Algorithm 1 employs a preprocessing procedure, which we give below.

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.

1 **let** $visited[\,]$ be a map from states to boolean values;
2 **let** $dist[\,]$ be a map from states to either natural numbers or nil;
3 **foreach** $q \in Q$ **do**
4 $visited[q] := false$;
5 $dist[q] := nil$;
6 `PreprocessAux` ($\mathcal{SP}$, q_0, v, $dist[\,]$, $visited[\,]$);
7 **foreach** $q \in Q$ **do**
8 $visited[q] := false$;
9 `PreprocessAux` ($\mathcal{SP}$, q_0, v, $dist[\,]$, $visited[\,]$);
10 **return** $dist[\,]$;

Let us now consider why it accomplishes what we claim. We have said that a preprocessing of $\mathcal{SP}$ is required in order to deal with its cyclic paths. By this provision, we shall be able to determine at any state of $\mathcal{SP}$ which successor is closer to 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()`.

Notice first that 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 stores 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 the acyclic paths shall be 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 procedure 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 0. Second, if the *source* being considered has no successors, and thus cannot get to v, which implies that $dist[source] = \infty$. And third, when all successors of *source* have already being visited.

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 := ∞;
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 then
18         min := min + 1;
19     dist[source] := min;
```

In the latter case it means that the function has found a cycle. Moreover, because of the recursive nature of the function, none of the successors q' will have their $dist[q']$ set yet, so that $dist[source]$ shall remain *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 shall be 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 assign it the shortest distance considering a way to get out of the cycle. Since by Definition 2 there is always an acyclic path towards a verdict state, it is always possible to calculate this distance. This provides guidance to avoid cycles whenever possible.

Successor Selection

The procedure for selecting the best among possible successors, which employs the results from the preprocessing, is given below.

Procedure `RemoveBest(`*Unexplored, dist[ ]*`)`

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

Feasible Trace Synthesis

Feasible traces, once found, are made explicit using the information available at the search stack. The procedure below accomplishes this.

Procedure `BuildTrace(`*SynchStack*, q^*, $depth^*$`)`

```
1  let Trace 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' at the beginning of Trace;
6          Put f at the beginning of Trace;
7          q* := q;
8          depth* := depth* − 1;
9  return Trace;
```