

Time in discrete agent-based models of socio-economic systems

N. Botta^{*,1}, A. Mandel², and C. Ionescu¹

¹*PIK, Potsdam Institute for Climate Impact Research*

²*Centre d' Economie de la Sorbonne, CNRS et Université Paris 1 Panthéon Sorbonne*

Abstract

We formulate the problem of computing time in discrete dynamical agent-based models in the context of socio-economic modeling.

For such formulation, we outline a simple solution. This requires minimal extensions of the original untimed model. The proposed solution relies on the notion of agent-specific schedules of action and on two modeling assumptions. These are fulfilled by most models of practical interest.

For models for which stronger assumptions can be made, we discuss alternative formulations.

1 Introduction

Agent-based models, discrete dynamic models. In modeling socio-economic systems, the term agent has long been used informally to address, collectively, different socio-economic entities like for instance households, firms, shareholders.

More recently, the term agent-based model (multi-agent system, agent-based system) has been used – also informally – to describe *computational* models of a heterogeneous population of agents and their interactions <http://www.openabm.org/faq>.

In agent-based models, different types of agents are usually characterized by different *state variables*. A household, for instance, could be represented – in an oversimplified model – by a single positive real number giving the amount of “available labor” of that household.

In *discrete dynamical* models, the state variables of the agents – the agents states – evolve in discrete steps from some given initial values. Agents might be able to influence their evolution – the evolution of their state (variables) – by means of *controls*. A firm, for instance, might be able to increase its production by investing part of its resources in new equipment.

At any step, the set of admissible controls depends upon the agent’s *actual* state. The agent observes its actual state and selects an admissible control on the basis of a *policy*. This is a function that maps the agent’s state into admissible controls. Once the control is selected, the agent’s state is updated.

The update rules can be deterministic, non-deterministic or stochastic. They model idealized economic *actions* like trading, exchange of goods, production, consumption. Update rules for a given agent might depend on its state, on its control, on the states and controls of other agents and on model parameters. Update rules which model interactions between agents – for instance trading or exchange rules – usually imply simultaneous updates of the states of the interacting agents.

*botta@pik-potsdam.de

Discrete dynamic models, time. In discrete dynamic agent-based models, *time* plays a central role. Certain actions, for instance paying dividends or reporting profits or losses, take place at or between well defined temporal deadlines. Contracts, for instance to sell or buy a certain amount of a good at a certain price, explicitly contain or implicitly assume some shared notion of time. For instance, the time at which the contract expires.

Currently, there is no socio-economic theory which provides general, model-independent notions for the agents states, actions and update rules and which relates such notions to empirical observations¹: households, firms, shareholders and trading, goods exchange, production, consumption are, to the best of our knowledge, model-specific abstractions.

In particular, there is no general way of relating updates of the agents states to well defined time intervals or durations: the notion of *time* is model-specific.

Time in globally scheduled models. In certain models, a global schedule of actions is imposed on the agents from the outset [9], [10], [6]. For instance, consider a system of agents consisting of just firms and households. One can represent the agents by two lists: one of firms and one of households.

```
type Agents = ([Firm],[Household])
```

Throughout this paper, we introduce computational notions for agents, states, transition functions, etc. in Haskell. The syntax of Haskell is similar to standard mathematical notation [1], [7] and quite intuitive. Readers which are not familiar with functional programming languages might find appendix A useful.

Let `trade`, `produce` and `consume` be functions of type `Agents -> Agents`. Then the global schedule obtained by applying first `trade` then `produce` and, at last, `consume`

```
gs :: Agents -> Agents
gs = consume . produce . trade
```

can be used to evolve an initial system of agents `ags` step-by-step:

```
ags' = gs ags
ags'' = gs ags'
ags''' = gs ags''
...
```

Provided that `trade`, `produce` and `consume` satisfy suitable conditions, a shared time can be introduced by simply counting the number of trade-produce-consume steps. Informally, the conditions require that `trade`, `produce` and `consume` model the effects of trading, producing and consuming on the same time scale: a day, a month or a year, for instance. Of course, this time scale has also to be consistent with whatever (implicit or explicit) time scale appears in agents state “rate” parameters: investment rates, discount rates, etc.

Globally scheduled systems are straightforward to implement but have obvious drawbacks. For many socio-economic systems, imposing a global schedule could be a severe *overspecification*. It could force the system to behave in implementation specific ways and prevent the study of interesting phenomena, in particular self-organization and synchronization.

¹The lack of general notions and empirical references has a number of implications. One is that agent-based models of socio-economic systems cannot be “applied” in the sense in which computational models in engineering, e.g., models for numerical weather prediction or material stress analysis, are routinely applied. In particular, agent-based models of socio-economic systems cannot be used for predicting the evolution of a given social-system in the sense in which numerical weather prediction models are used. At the present state, it is probably fair to view agent-based models as frameworks in which different hypotheses (for instance about the mechanisms of interaction between agents) can be formulated and by which questions like “What are the impacts of a more aggressive investment policy ?” can be studied under different such hypotheses.

One can of course weaken the effects of overspecification by introducing some randomness in the system. This can be done at different levels. For instance, one could apply, at different steps, the same basic action in different random orders. Often, actions like `trade` are expressed by folding some bilateral trade primitive on a random list of agent pairs [3]. Such lists can be randomly drawn every new step.

A more relaxed approach, however, is to simply avoid imposing any (deterministic, non-deterministic, stochastic) global step schedule from the outset and allow agents to act according to their own internal rules.

Time in models with internal transition functions. In models with internal transition functions, every agent is equipped with its own individual step function. This is part of the agent’s state. To distinguish between the step function of the whole system and that of an individual agent, we call the latter the agent’s *transition* function.

In models with internal transition functions, there is no model-specific global schedule. A step of the whole system is simply done by executing the transition function of each agent. This updates the agent’s state and, therefore, its transition function. Thus, next step, the same agent² might get updated according to a different rule: agents can learn.

For agents to be capable of interactions, the internal transition functions have to depend on some environment or input, e.g., to represent the state of other agents. A canonical approach is to represent an agent’s environment through a set of messages [15], [4], [11], [5]. Agents interact only by exchanging messages and the transition function of each agent only depends on the agent’s state and on a set of *incoming* messages. Each transition updates the agent’s state and generates a set of *outgoing* messages. The global schedule is model-independent and simply consists of iterating message exchange and internal update steps.

The notion of fully encapsulated agents interacting only through message exchange is a very popular one in computing science. It is at the core of the notion of process in models of parallel computation [2], multitasking [14] and distributed computing [13].

In agent-based modeling, the notion of fully encapsulated agents allows a clear distinction between model-specific aspects – the syntax of messages, the agent-specific transition functions – and model-independent problems, e.g., message parsing and exchanging. This allows developers to better understand, communicate, maintain, extend and refactor agent-based models. Model validation and comparison can be done in a more systematic and disciplined fashion.

Introducing a shared notion of time in models with internal transition functions, however, is not straightforward. Since there is no model-specific global schedule, there is no canonical way of associating a “duration” to a single step.

During an update step, for instance, different agents might be performing actions as different as accounting for the integral effects of “almost time-continuous” processes (for instance, accounting for one year’s consumption or production), or computing offers or demands in response to incoming messages. Depending on the specific application domain, the latter can be considered “almost instantaneous” actions. Attempts at introducing a global step duration by means of simple minded rules – e.g., by taking the duration of the transition of the “slowest” agent – can easily lead to inconsistent results, see section 4.

Similarly, in a message exchange step, different agents might be involved in message exchanges which can be thought of as of requiring a sizable duration (for instance message exchanges used to model goods transportation) and in exchanges of information which can be thought as of taking place almost instantaneously.

Thus, there seems to be no obvious way of computing time in agent-based models with internal transition functions.

²Of course, the agent’s state has changed in a step. Here we use the notion of *same* agent in the understanding that every agent has a state variable, e.g., an identity number or name, which is invariant under its transition function.

Outline. In the next section we present a simple approach for building models with internal transition functions. In section 3, we formulate the problem of introducing a shared notion of time in such models and present a simple solution. Two alternative formulations of the problem are outlined in section 4.

2 Models with internal transition functions

Agents in models with internal transition functions are defined in terms of a few general notions. These can be introduced from different viewpoints.

Here, we take the viewpoint of an agent’s observer. We look at the agent from the outside and make no specific assumption about what the agent represents – a firm, a household or maybe a component of the climate system. We focus on how we can interact with the agent generically. Minimal interactions are:

- Query the agent for its identity. This is a value of a type `Id` suitable to unambiguously identify the agent in a possibly large set, typically `Id` is a synonym of `Int`.
- Query the agent for its list of outgoing messages.
- Call the agent’s transition function with a list of incoming messages.

We assume that all outgoing messages come paired with the identity of the agent to which they are addressed. Similarly, incoming messages are paired with the identity of the agent that sent that message.

As we shall see shortly, the above interactions are enough to specify a function that collects the outgoing messages, dispatches the messages to the proper addresses and calls the internal transition function of all agents. This is a generic stepping function for models with internal transition functions. It can be used to iterate a set of agents independently of the number of agents, of their types, of the specific model, etc.

Agent data type. In order to define a generic stepping function for agent-based models with internal transition functions, we need to make the above notions operational.

As in the example given in the introduction, we are going to represent sets by lists³. In contrast to the example, however, we do not make any assumption about the types of the agents involved in the model. Thus, we cannot say that our agents are tuples of lists of whatever concrete types. In other words, we need a way to build collections of agents of different types and to treat agents of different types uniformly.

There are different ways to do so. Here, we follow the approach originally proposed in [8]. The data type `Agent`⁴ is polymorphic in the type of the messages `m`:

```
data Agent m = forall s . Agent (s -> Id)
                    (s -> [(m, Id)])
                    (s -> [(m, Id)] -> s)
                    s
```

Independently of the type `s`, an agent is constructed, by the data constructor `Agent` on the right-hand side of the equality, in terms of four data:

- A function of type `s -> Id`, the agent’s identity.
- A function of type `s -> [(m, Id)]`, the agent’s outgoing messages.

³We do so to keep the notation as simple as possible. In “real” applications, of course, lists have to be replaced by run-time efficient data structures, e.g., by arrays.

⁴We slightly abuse our notation and use the same type name of the example in the introduction.

- A function of type $s \rightarrow [(m, Id)] \rightarrow s$, the agent's transition function.
- A variable of type s .

This definition of agent is enough for agents to be able to update their internal transition function: this can be written in terms of s – a totally arbitrary type – which, in turns is updated by the agent's transition function. Of course, the same is true for the function that defines the agent's outgoing messages.

In fact, the agent's transition function and the agent's outgoing messages functions could be collapsed into a single function of type $(s, [(m, Id)]) \rightarrow (s, [(m, Id)])$. This would compute a new agent's internal state and its corresponding outgoing messages. This formulation of the data type `Agent` would be both simpler and more symmetrical but make the definition of the step function for a collection of agents more cumbersome.

The data type `Agent` is parametrized on the type of messages m . In most applications, messages can be thought of as sentences of domain specific languages. They represent application-specific modes of interaction between agents. In an oversimplified model of barter economies, for instance, a data type for messages could look like:

```
data Msg = Offer Trade | Accept Trade
```

and `Trade` could be a pair $((Good, Quantity), (Good, Quantity))$ representing the terms of a bilateral trade. In such context, the message

```
Offer ((water, 1.2), (wine, 0.3))
```

could be interpreted as a binding offer of 1.2 units of water for 0.3 units of wine.

Step function. With `Agent` defined as above, a generic stepping function for a list of agents can be defined in three steps. First, we introduce three helper functions

```
ident :: Agent m -> Id
ident (Agent i o t s) = i s

outs :: Agent m -> [(m, Id)]
outs (Agent i o t s) = o s

step :: Agent m -> [(m, Id)] -> Agent m
step (Agent i o t s) ins = (Agent i o t s')
  where s' = t s ins
```

to compute the identity (`ident`), the outgoing messages (`outs`) and to apply the transition function (`step`) of a generic agent. Second, we define helper functions `inMsgs` and `outMsgs` which collect and dispatch outgoing messages. Consider

```
inMsgs :: [Agent m] -> (Id -> [(m, Id)])
inMsgs ags = \i -> outs (fromJust (find ((i ==) . ident) ags))
```

`inMsgs` takes a list of agents and computes a function. This function returns, for any given agent identity i , the outgoing message of the agent in the list whose identity is i (if no such agent exists, the function fails).

The name `inMsgs` underlines the “observer's viewpoint” taken above. From the point of view of an external observer, the messages sent by the agents – their outgoing messages – are incoming. The observer's outgoing messages are those dispatched to the agents. These are computed by `outMsgs`:

```

outMsgs :: [Agent m] -> (Id -> [(m, Id)])
outMsgs ags = \i -> [(msg, i') | i' <- map ident ags,
                      (msg, i'') <- (inMsgs ags) i',
                      i'' == i]

```

Here the computation is slightly more complicated: for a given agent's identity i , the function `outMsgs ags` has to compute the list of messages sent to i and the identity of the sender. This is done as follows: for a given i , `outMsgs ags` goes through the list of all agents `ags`. For each agent in `ags`, it computes its identity i' . Of the outgoing messages of i' (computed by `(inMsgs ags) i'`), it selects those which are addressed to i (the condition $i'' == i$) and pairs them with the identity of the sender i' .

Let's pause for a moment. The two functions `inMsgs` and `outMsgs` are simple but not trivial. They are designed to easily define a message exchanger. This is a function that takes the agents outgoing messages – each paired with the identity of the agent the message is sent to – and computes the agents incoming messages. For a given agent, each incoming message is paired with the identity of the agent that sent that message. This is essential for an agent to be able to reply to incoming messages. In fact, `inMsgs` and `outMsgs` fulfill the following *specification*:

$$(\text{msg}, j) \text{ elem } (\text{outMsgs ags } i) \Leftrightarrow (\text{msg}, i) \text{ elem } (\text{inMsgs ags } j) \quad (1)$$

We read the specification as follows: (msg, j) is an incoming message for agent i if and only if (msg, i) is an outgoing message of agent j . Notice that incoming and outgoing messages are computed by `outMsgs` and `inMsgs`, respectively. Again, the names given to these functions reflect the point of view of an observer. Notice also that the specification is a *mathematical* expression: \Leftrightarrow is not part of the Haskell language. With `inMsgs` and `outMsgs` fulfilling equation (1), a message exchanger can be defined straightforwardly:

```

exchMsgs :: [Agent m] -> [[(m, Id)]]
exchMsgs ags = [outMsgs ags i | i <- map ident ags]

```

The function takes a list of agents `ags` and computes a list of lists (of pairs (m, Id)). The latter contains, for each agent in `ags`, the list of all messages (message-identity pairs) sent to the agent.

With `exchMsgs` in place, a generic function for iterating a list of agents an arbitrary number of steps can be defined as follows:

```

iterateAgents :: [Agent m] -> Nat -> [Agent m]
iterateAgents ags 0 = ags
iterateAgents ags (n + 1) = iterateAgents ags' n
  where ags' = [step ag msgss | (ag, msgss) <- zip ags msgss]
        msgss = exchMsgs ags

```

The function is defined, as one would expect, recursively. When asked to compute zero iterations from an initial list of agents `ags`, `iterateAgents` simply returns the initial list. Otherwise, if the number of iterations is $n + 1$, `iterateAgents` calls itself for n further steps starting from a new list of agents `ags'`. This is obtained from `ags` as follows: first, the list `msgss` of lists of messages sent to the agents is constructed with `exchMsgs`. `msgss` has as many elements (lists) as `ags`. Thus, `ags` can be zipped with `msgss` to form a list of pairs of type $(\text{Agent } m, [(m, \text{Id})])$. For each element of such list, the helper function `step` is called with the correspondent agent and message list. This computes a new agent, one for each agent in `ags`.

Models with internal transition functions. The stepping function `iterateAgents` gives computational meaning to the notion of agent-based models with internal transition functions informally discussed in the introduction.

It is a generic, application independent component which plays an important role in building frameworks for agent-based modeling. For instance, `iterateAgents` could be applied for implementing a function which advances an initial list of agents an arbitrary number of steps and reports about the internal states of the agents at fixed or variable number of steps. This requires that `Agent` is extended with functionalities for querying the internal state of the agent. We do not discuss these functionalities here.

The stepping function `iterateAgents` forms the basis for formulating the problem of introducing a shared notion of time in agent-based models with internal transition functions. This is the subject of the next section.

3 Time in models with internal transition functions

With the stepping function `iterateAgents` of section 2, we are ready to formulate the problem of computing time in agent-based models with internal transition functions. In this section, we present one such formulations together with a solution. In section 4 we discuss alternative formulations.

A problem formulation. Consider a list of agents `ags :: [Agent m]` and assume that the transition functions of `ags` are driven by agent-specific, internal schedules. Think of an agent’s schedule as of a sequence of pending economic actions. Specifically, assume that:

1. All agents in `ags` can distinguish between internal states with pending actions (unaccomplished schedules) and internal states without pending actions (corresponding to accomplished schedules).
2. The internal schedules of `ags` are consistent. Informally, schedules are consistent if they represent sequences of economic actions on the same time scale: a day, a month or a year, for instance⁵.

For instance, a household agent could be driven, at some point of a simulation, by an internal schedule representing actions like “trade labor for wage”, “consume”, “save”. The head of the sequence – its first element – could represent the household’s *active* action: the action the household is actually trying to accomplish. While trading labor for wage, the household could react to a wage offer by accepting the offer and removing the schedule head from its schedule. This would make “consume” the new active action of the household.

Let’s call the time scale of the internal schedules of `ags` a time *period*. Assumptions 1) and 2) are too weak to associate a monotonically increasing number of periods to the agents states `iterateAgents ags 0`, `iterateAgents ags 1`, etc. They do not allow to meaningfully compute the duration of a step as a fraction of the time period.

However, 1) and 2) allow one to query `iterateAgents ags n` for pending actions at arbitrary `n`. This suggests that a shared notion of time can be introduced as an emerging synchronization condition: we start with `ags` at an arbitrary number of time periods. We advance `ags` step-by-step until we reach a state in which all agents are *idle* i.e., they have no pending actions. At this point, we say that a time period has elapsed and that the agents are *synchronized*. We can increment the number of time periods by one and start a new period.

Thus, the problem of computing a shared time can be formulated as the problem of detecting when the agents are synchronized. For agents to have their behavior depending on time, we also need a mechanism to keep track of the number of periods and communicate synchronization events to the agents.

⁵We have already introduced this notion of consistency in section 1 for models with a global schedule.

A solution. The problem of computing a shared time can be solved with minimal extensions of the set of messages and with the introduction of a “timekeeper” agent. Specifically, we extend the set of messages with a `AreYouIdle` rule for querying agents for absence of pending actions. Idleness queries are answered with `IdlePositive` or `IdleNegative` messages. Similarly, we introduce a `AreAllIdle` message for timekeeper queries. Such queries are replied by `AllIdlePositive` or `AllIdleNegative` messages. For instance, the data type `Msg` of section 2 would be extended in the following way⁶:

```
data Msg = Offer Trade
         | Accept Trade
         | AreYouIdle
         | IdlePositive
         | IdleNegative
         | AreAllIdle
         | AllIdlePositive
         | AllIdleNegative
```

With these new messages, we are ready to define a timekeeper agent. We are going to skip the details of the implementation and focus on the timekeeper’s internal state and transition function. The state of the timekeeper consists of six state variables:

```
type State = (Id, [Id], Time, Bool, Bool, [(Msg,Id)])
```

Thus, the timekeeper’s transition function, say `tf`, is a function of type `State -> [(Msg,Id)] -> State`. Let `(i, is, t, aif, tif, outs) :: State`. The first variable, `i :: Id`, is the timekeeper’s identity. As mentioned earlier, the internal transition functions of the agents preserve the agents identities. This means that `tf` fulfills the specification:

$$s' = \text{tf } s \text{ msgs} \Rightarrow \text{ident } s' = \text{ident } s \quad (2)$$

The second state variable, `is :: [Id]`, represents the set of agents timed by the timekeeper. For each timed agent, `is` stores the agent’s identity.

The third state variable, `t :: Time`, represent the number of time periods measured by the timekeeper. Typically, `Time` is a type synonym for `Nat`, the type of natural numbers. The state variables `aif`, `tif :: Bool` are flags. As we will see in short, `aif` is true at the beginning of a step if all agents timed by the timekeeper have been found to be idle in the previous step. Similarly, the “time incremented” flag `tif` is set to true whenever time is incremented. The sixth state variable, `outs :: [(Msg,Id)]`, is used to store the outgoing messages of the timekeeper.

At each iteration, the internal transition function of the timekeeper `tf` is responsible for updating `t`, `aif`, `tif` and `outs` according to the list of incoming messages. The transition function operates on an internal state `s :: State` and on the incoming messages `ins :: [(Msg,Id)]` as follows:

```
tf s ins = s6
  where allIdle = areAllIdle s ins
        s0 = s
        s1 = emptyMsgs s0
        s2 = if (allIdle && not (wereAllIdle s1))
              then incrementTime s1
              else s1
        s3 = setTimeIncremented (allIdle && not (wereAllIdle s1)) s2
        s4 = foldl f s3 [(AreYouIdle, k) | k <- peers s3]
        s5 = setAllIdle allIdle s4
```

⁶Again, with slight abuse of notation.

```

f s m = appendMsg m s
s6 = foldl g s5 ins
g s (AreAllIdle, k) = if allIdle
                      then appendMsg (AllIdlePositive, k) s
                      else appendMsg (AllIdleNegative, k) s

g s _ = s

```

First, the outgoing messages of $s_0 = s$ are emptied. The outcome is the new state s_1 . The function (command) `emptyMsgs` is simply:

```

emptyMsgs :: State -> State
emptyMsgs (i, is, t, aif, tif, outs) = (i, is, t, aif, tif, [])

```

Second, `tf` checks if all agents are idle *and* were not idle so far. In this case the time in s_1 is incremented and the “time incremented” flag is set to `True`. This is done in two steps with outcomes s_2 and s_3 , respectively. Testing if all agents are idle is done by the function `areAllIdle`. This is defined as follows:

```

areAllIdle :: State -> [(Msg,Id)] -> Bool
areAllIdle s ins = (is == js)
  where is = sort (map snd (filter isIdlePositive ins))
        js = sort (peers s)

```

`areAllIdle` takes a timekeeper’s internal state s and a list of incoming messages ins and computes a Boolean value. This is true if ins contains a positive answer to the `AreYouIdle` query for all the agents in the set timed by the timekeeper. This set of “peer” agents is computed by the helper function `peers`:

```

peers :: State -> [Id]
peers (i, is, t, aif, tif, outs) = is

```

The implementation of `areAllIdle` is straightforward but requires `Id` to be equality comparable and sortable. While the first requirement is natural, the second one is an overspecification only motivated by this implementation. In “real” applications it could be easily relaxed⁷.

Testing whether all agent were idle (in the previous iteration) is simply done by reading the `aif` flag of the timekeeper’s state:

```

wereAllIdle :: State -> Bool
wereAllIdle (i, is, t, aif, tif, outs) = aif

```

In the next step, the list of outgoing messages is filled with `AreYouIdle` queries to all agents timed by the timekeeper. The outcome of this computation is s_4 . With s_5 , we set the “all idle” flag to the value computed by `areAllIdle`. The implementation of `setAllIdle` is again straightforward:

```

setAllIdle :: Bool -> State -> State
setAllIdle aif' (i, is, t, aif, tif, outs) = (i, is, t, aif', tif, outs)

```

One more computation is required to obtain the new state of the timekeeper. The incoming messages might contain `AreAllIdle` queries, e.g., from agents timed by the timekeeper. For agents to be able to adapt to synchronization events, – for instance by updating some private time period counter and starting a new period – such queries have to be answered properly. In the last computation, with outcome s_6 , all `AreAllIdle` queries contained in ins are answered and the answers are appended to the outgoing messages.

⁷But notice that comparing lists for identity can only be done efficiently if the list elements are sortable.

Example. As an example, we have introduced a timekeeper in a simple model of the dynamics of prices in an exchange economy. In this model, built upon [9], agents exchange goods on the basis of private barter prices so as to maximize the utility associated to the bundle of goods they eventually consume.

The main parameter of this model is a number of goods. At the beginning of a simulation, each agent is endowed with a private barter price for every good, randomly chosen from a uniform distribution. The schedule of the agents prescribes ten production, trading and consumption actions before the agents update their prices.

Each agent is characterized by a “production” good. This is the only good produced by that agent. Production simply consists in adding to the agent’s stock of goods a fixed amount of its production good. For an agent, a trading action involves a number of bilateral interactions with other agents. In each interaction, the agent offers a certain amount of its production good in exchange for an amount of the production good of the other agent. The agent’s demand and offer are computed on the basis of the agent’s private barter prices. Offers and demands are encoded and sent through **Offer Trade** messages. They can be replied by **Accept Trade** messages or ignored. The reply – and therefore the outcome of the interaction – is governed by the agent’s offer and demand and by the private barter prices of the responding agent. Thus, the private prices of the agents are policies (in the sense used in the introduction) or strategies in a game-theoretic sense.

During the consume action, the stock of goods is set to zero and determines a fitness which is computed on the basis of a utility function. Finally, the update of private prices consist in the implementation, via the exchange of messages, of a simple replicator dynamics algorithm [12].

We do not further discuss the basic model here. The code is available from the authors upon request and [3] can also be consulted. What is relevant for our example is that trading is a stochastic process. The agents (of a foreign sector) a given agent attempts to trade with, are selected randomly according to a given probability distribution. Thus, the number of iterations required for each agent to do ten trade actions (and, finally, to go through its schedule) is a random variable, too.

The following output shows the number of iterations, the number of periods (time) and the agents internal prices – their trading strategies – at those iterations at which the number of periods is increased. For readability, we only report the prices of the first two agents. For the first 10 periods, the number of iterations needed to complete a period varies between 89 and 97.

```

iter: 0; time: 0; prices: [[0.64,0.37,0.19], [1.00,0.28,0.91], ...]
iter: 90; time: 1; prices: [[0.55,0.91,1.00], [0.55,0.91,1.00], ...]
iter: 179; time: 2; prices: [[0.19,0.10,0.64], [0.19,0.10,0.64], ...]
iter: 272; time: 3; prices: [[0.55,0.91,1.00], [0.55,0.91,1.00], ...]
iter: 369; time: 4; prices: [[0.19,0.10,0.64], [0.19,0.10,0.46], ...]
iter: 462; time: 5; prices: [[0.19,0.10,0.64], [0.19,0.10,0.64], ...]
iter: 558; time: 6; prices: [[0.19,0.10,0.64], [0.19,0.10,0.64], ...]
iter: 650; time: 7; prices: [[0.19,0.10,0.64], [0.19,0.10,0.64], ...]
iter: 743; time: 8; prices: [[0.19,0.10,0.64], [0.19,0.10,0.64], ...]
iter: 837; time: 9; prices: [[0.19,0.10,0.64], [0.19,0.10,0.55], ...]
iter: 931; time: 10; prices: [[0.28,0.10,0.64], [0.19,0.10,0.64], ...]

```

Remarks. We have outlined a solution of the problem of computing time in agent-based models with internal transition functions. The solution requires minimal modifications of an untimed set of agents and the introduction of a special agent, the timekeeper.

In extending the set of messages from the example given in section 2, we have introduced messages for querying the timekeeper for synchronization events. It is straightforward to introduce messages to ask the timekeeper to report the number of time periods or other information.

In implementing the transition function of the timekeeper, our main concern has been understandability. The function can (should) certainly be implemented more efficiently.

The notion of time on which problem formulation and solution are based is relative to a set of agents. An agent can be monitored by more than one timekeeper. However, timekeepers monitoring different agent sets will generally report different time measures.

4 Alternative formulations

In section 3 we have presented a formulation of the problem of computing time in agent-based models with internal transition functions.

The formulation is consistent with the notion of discrete time as an emerging property of a set of agents: the agents get started and keep on interacting – possibly according to some internal schedule of tasks – until all of them have no pending tasks. When this synchronization condition occurs, time is incremented and the agents are ready to start a new period.

Thus, time accounts, at an aggregate level, both for possibly very different modes of interactions – e.g. due to different conventions in effecting certain transactions – and for possibly very different schedules of actions of different agents.

From the modeling perspective, the formulation imposes two requirements: 1) the agents have to be able to distinguish between states corresponding to pending actions and states with no pending actions; and 2) the agents internal schedules have to be based on the same time scale.

The formulation does not require modelers to provide a well-defined measure for the duration of the actions performed by the internal transition function of the agents. There might be situations in which this information is actually available.

In these cases, the data type `Agent` defined in section 2 can be extended with a function of type `s -> [(m, Id)] -> Real` which measures the duration of the actions performed by the agent’s transition function. This function depends, in general, both on the agent’s internal state and on the incoming messages. Similarly, the set of helper functions `ident`, `outs` and `step` can be extended with a correspondent `dur :: Agent m -> [(m, Id)] -> Real` helper.

Formulations of the problem of computing time which require modelers to provide `dur` are more demanding than formulations which only assume 1) and 2). However, they do not require additional assumptions on the time scale of the agents internal schedules⁸.

In the rest of this section we shortly outline two such alternatives. We do not present ready solutions but discuss the logical consequences of the alternative formulations from a computational perspective.

Local time, iteration allocation, message time tagging. A straightforward way of introducing time in models for which `dur` is available is to equip agents with a local, agent specific time, say τ_i for agent with identity i . Initially, the internal time of all agents is set to some initial value, say t_0 , so that $\tau_i = t_0$ for all i . The internal time is incremented after every internal transition with the duration given by `dur`, and the resulting value is used to tag all outgoing messages. Thus, if agent i has internal time τ_i and performs an action whose `dur` value is d_i , then at the end of this action it will have internal time $\tau_i + d_i$ and all messages it sends will be tagged with this value. The message exchanger will then send the messages as described at the end of section 3 only if the internal time of the receiver is ulterior to the message tag, in order to prevent “time travel”. Undelivered messages are kept in a queue by the message exchanger, and used at the next message exchange.

As can be seen from this brief description, this approach involves extensions to the message exchanger, which has to have access to the internal time of each agent, going perhaps against the grain of the “fully encapsulated agents” paradigm.

⁸Since the duration of actions is perfectly known.

Global time (simple minded). If the duration of the actions performed by the internal transition function is known for all agents, we may well compute a global time by selecting, for each call of `iterateAgents` `ags 1`, the longest duration. This would correspond to the intuition that exchanging messages implies a synchronization barrier: “fast” agents have to wait for “slow” ones to be ready to send and receive messages.

Though apparently natural, this scheme for introducing time has a number of disturbing consequences. One is that effects of internal transitions which are tied to well defined time periods – for instance accounting for the interests accumulated over one year by a given agent – might actually get “diluted” over longer time intervals, depending on the duration of the internal transitions of other agents.

In fact, computing a global time by selecting the longest (shortest, average, etc.) duration over the durations of the internal transition functions is a too simple minded approach. It makes sense only if, at each call of `iterateAgents` `ags 1`, all internal transition functions of `ags` represent actions on the same time scale. From the modeling perspective, this is by far more demanding than 1) and 2).

5 Conclusions

We have presented a formulation of the problem of computing time in discrete dynamical agent-based models with internal transition functions in the context of socio-economic modeling.

Such models can be iterated generically. Each step of the iteration consists of two sub steps. In the first step, agents exchange application specific messages. In the second step, agents are updated according to their internal transition function and to the messages received.

Compared with models with a global schedule, models with internal transition functions depend on weaker assumptions and are better suited for modeling systems of heterogeneous agents.

For the formulation of the problem of computing time proposed, we have outlined a simple solution. This relies on the notion of agent-specific (internal, private) schedules of action and on two assumptions: 1) that the agents are able to distinguish states with pending actions from states with no pending actions; and 2) that the agents internal schedules are based on the same time scale.

These assumptions are reasonably weak and seem appropriate for a wide class of models of practical interest. For models for which stronger assumptions can be done we have discussed alternative formulations.

Acknowledgments. The work presented in this paper heavily relies on free software, among others on hugs, vi, the GCC compiler, Emacs, L^AT_EX and on the FreeBSD and Debian / GNU Linux operating systems. It is our pleasure to thank all developers of these excellent products.

A

As previously mentioned, Haskell notation differs sometimes from standard mathematical notation. To alleviate the potential confusion caused by this, we present in this section a summary of the most important differences, and a couple of definitions of the most important standard library functions we have used in this paper.

1. *Membership.* Haskell is a typed language, so in most cases the membership relation is replaced by the “of type” relation, for example $x \in X$ is written `x :: X`. We usually represent subsets of values of a given type by lists, and use the standard Haskell `elem` function to express membership of a value to a given list. The names of types are capitalized, thus we write `a :: A` for the membership of `a` to the type `A`. Haskell allows polymorphic type assignment, for example `[] :: [a]` which can be read “for any type `a`, `[]` denotes an element of type list of `a`, namely the empty list”. As

can be seen in this example, type variables are written in lowercase and are implicitly universally quantified.

2. *Functions.* Functions $f : A \rightarrow B$ are represented in Haskell as $f :: A \rightarrow B$. Function application, $f(a)$ is denoted by juxtaposition: $f\ a$, the brackets being omitted.
3. *List comprehension.* Just as sets are commonly represented by lists, so set comprehensions translate to list comprehension. The standard notation $\{ x \mid x \in A, P(x) \}$ for the set of all x in A satisfying property P is represented by $[x \mid x \leftarrow \text{as}, p\ x]$ if A is represented by the list as and P is represented by the boolean valued function p .
4. *Maps and folds.* Lists are the most widely used data structure in functional programming, and the most frequent operations performed on them are maps and folds. Given a function $f :: A \rightarrow B$ and a list $[a_0, a_1, \dots, a_n]$ of elements of type A , we have that

$$\text{map } f\ [a_0, a_1, \dots, a_n] = [f\ a_0, f\ a_1, \dots, f\ a_n]$$

Given some $e :: E$ and an operation $* :: E \rightarrow A \rightarrow A$, folding from the left produces

$$\text{foldl } *\ e\ [a_0, a_1, \dots, a_n] = ((e * a_0) * a_1) \dots * a_n$$

References

- [1] R. Bird. *Introduction to Functional Programming using Haskell*. International Series in Computer Science. Prentice Hall, second edition edition, 1998.
- [2] N. Botta and C. Ionescu. Relation based computations in a monadic BSP model. *Parallel Computing*, 33:795–821, 2007.
- [3] N. Botta, A. Mandel, C. Ionescu, M. Hofmann, D. Lincke, S. Schupp, and C. Jaeger. Computational patterns in exchange economy models. *submitted for publication to Applied Mathematics and Computation*, July 2009.
- [4] F. Cesarini and Thompson S. J. *Erlang Programming, A Concurrent Approach to Software Development*. O’Reilly Media, 2009.
- [5] C. Deissenberg, S. van der Hoog, and Dawid H. EURACE: A massively parallel agent-based model of the European economy. *Mathematics and Computation*, 204(2):541–552, 2008.
- [6] G. Dosi, G. Fagiolo, and A. Roventini. Schumpeter Meeting Keynes: A Policy-Friendly Model of Endogenous Growth and Business Cycles. Accepted for publication in the *Journal of Economic Dynamics and Control*, 2009.
- [7] S. L. Peyton Jones et al. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, 2003.
- [8] J. Gibbons. Unfolding abstract datatypes. In *Mathematics of Program Construction*, Lecture Notes in Computer Science, pages 110–133. Springer, 2008.
- [9] H. Gintis. The Dynamics of General Equilibrium. *Economic Journal*, 17:1280–1309, 2007.
- [10] C. Jaeger, A. Mandel, S. Fürst, W. Lass, and F. Meissner. Lagom generiC: A Minimal Description from an Economic Point of View. *submitted to the ECF-GSD Modeling Workshop on Agent-Based Modeling for Sustainable Development, Venice 2-4 April 2009*, 2009.

- [11] G. Laycock. *The theory and practice of specification based software testing*. PhD thesis, University of Sheffield, Dept. of Computer Science, 1993.
- [12] H. Peyton Young. The evolution of conventions. *Econometrica*, 61:57–84, 1993.
- [13] D. Sangiorgi and D. Walker. *The Pi-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.
- [14] W. Stallings. *Operating Systems: internals and design principles*. Prentice Hall, fifth edition edition, 2005.
- [15] M. Wooldridge. *An Introduction to Multiagent Systems*. John Wiley & Sons Ltd, 1966.