Teleo-Reactive Agents in a Simulation Platform

Vasileios Apostolidis-Afentoulis[a] and Ilias Sakellariou[b]
Department of Applied Informatics, School of Information Sciences, University of Macedonia, 156 Egnatia Str., GR-54636 Thessaloniki, Greece

Keywords: Agent Based Modeling and Simulation, Simulation Languages and Platforms, Teleo-Reactive Agents, NetLogo.

Abstract: Agent based modeling and simulation (ABMS) has been applied to a number of research areas including economics, social sciences, urban planning, epidemiology etc. with significant success. Agent simulation platforms have long been the principal tool, contributing to the wide adoption of ABMS, offering rapid model development for researchers in various fields. However, in most of the cases, agent behaviour in simulations is encoded employing standard or domain specific programming languages, with limited support for agent programming at a higher level. The present work contributes towards this direction, presenting an implementation of the Teleo-Reactive approach proposed by Nilsson for robotic control, to a well known ABMS platform, NetLogo. The aim is to allow modelers to encode complex agent models easily and thus to enhance the impact of ABMS to the respective fields.

1 INTRODUCTION

Agent-based Modeling & Simulation (ABMS) is a powerful technique with a wide range of applications in recent years, that advocates the idea of system modeling as a collection of autonomous decision-making entities. Each entity (agent), makes decisions based on its perception of the world, possibly an internal state and an executable specification of its behavior. Acting in a shared with other agents environment, leads to agent interaction resulting to the emergence of complex patterns, providing valuable insights into the dynamics of the actual system under consideration.

Although a large number of simulation platforms have been proposed, for instance 85 are reported by Abar et al. in (Abar et al., 2017), quite a few offering modeling of more complex (intelligent) agents, balancing between ease of use and modeling power of simulated entities is still an open research issue. For instance, Belief-Desire-Intention (BDI) agents (Rao et al., 1995), one of the most studied approaches in programming multi-agent systems, has been shown to provide significant benefits to social simulations (Adam and Gaudou, 2016), however, has not received a wider acceptance, possibly due to the fact that programming agents in such high level declarative abstractions is not a common skill among researchers of fields outside computer science (and in many cases not even withing the latter). This can be attributed to an extent to the lack of logic programming skills, that presents a major obstacle in the adoption of BDI and particular AgentSpeak as its one of the most common implementations.

An approach that provides such a balance in programming *goal directed* agents operating in dynamic environments, goal monitoring and replanning in a single elegant package, yet maintaining conceptual simplicity, is Teleo-Reactive (TR) programming (Nilsson, 1993; Nilsson, 2001). TR programs address issues such as gradually achieving the agent's goal, while gracefully responding to failures and environment changes, characteristics derived from their design target that was originally real world robotic systems. This makes TR an interesting agent behaviour modeling choice for agents in simulation platforms, that the current work introduces. Indeed, we support that robotic agents and agents in a simulation environment with a spatial dimension, share quite a few characteristics.

Thus, the main contribution of the current work is to present TR-TURTLES, an initial attempt of supporting Teleo-Reactive programming in a general purpose agent based simulation platform, i.e. NetLogo,

[a] https://orcid.org/0000-0002-4147-7011
[b] https://orcid.org/0000-0003-3522-6045

Apostolidis-Afentoulis, V. and Sakellariou, I.
Teleo-Reactive Agents in a Simulation Platform.
DOI: 10.5220/0011635300003393
In *Proceedings of the 15th International Conference on Agents and Artificial Intelligence (ICAART 2023) - Volume 1*, pages 26-36
ISBN: 978-989-758-623-1; ISSN: 2184-433X

through a meta-interpreter. We address issues such as syntactic support, belief store updates, rule action semantics, and provide a preliminary evaluation of the implementation.

To this end, the rest of the paper is organised as follows. Section 2 provides an overview of a subset of available ABMS platforms, focusing on their support for complex agent modeling. The Teleo-Reactive approach is briefly presented in Section 3. The introduction of the TR approach in NetLogo is presented in Section 4, and Section 5 presents an example of an agent simulation developed in TR-TURTLES. The execution overhead introduced by the meta-interpreter along with some preliminary results is discussed in Section 6. Finally, Section 7 concludes the paper and discusses future directions.

2 COMPLEX AGENTS IN ABMS PLATFORMS

A review of the all the available agent-based simulation platforms exceeds the scope of this paper. There are numerous review papers that address the subject and provide classifications of active and not so active platforms, as for instance (Abar et al., 2017; Kravari and Bassiliades, 2015; Railsback et al., 2006). Thus, the current section mainly reports on three, widely used, open source, *actively maintained* platforms, that are "general-purpose", i.e. do not target a specific application area. For instance, MATSim (Horni et al., 2016), has been extended to provide support for BDI (Padgham et al., 2014), however, given that it targets transportation simulations, it is not included. Thus, Repast, GAMA and NetLogo are briefly described focusing on their support in modeling complex agents.

The Repast Suite, (Repast Simphony (North et al., 2013), Repast HPC (Collier and North, 2013) and recently Repast for Python, are a set of agent-based modeling and simulation platforms, sharing some common concepts, but targeting different computational environments, i.e. workstations vs high performance computing (HPC). Agent models can be developed in mainstream general programming languages Java and Groovy, and ReLogo which can be considered as domain specific language following the syntax and semantics of NetLogo. To ease the development of complex agents, Repast Simphony supports encoding agent behaviour using statecharts (Ozik et al., 2015), with an automatic compilation of the latter to Java and integration to the simulation environment.

Adding programming constructs and agent concepts such as ACL and interaction protocols to Repast has also been considered in a number of projects like MISIA (García et al., 2011), Jrep (Gormer et al., 2011) and SAJaS (Cardoso, 2015). The approach in SAJaS, targets mainly a software engineering approach to assist JADE based systems to be tested before deployment, or modelers interested in FIPA compliant simulations latter turned to full MAS systems, bridging "*the gap between MAS simulation and development*".

Although there is no official BDI agent support in Repast, work in (Padgham et al., 2011) reports on a BDI model in JACK implemented in a modified version of Repast City ABM model, where authors explored synchronization issues between the BDI system and Repast, to allow a reliable simulation and modeling. The simulation framework consists of the ABM model (sensor-actuator agents) and the BDI model (reasoning agents) to provide high-level decision making. It supports interaction between the two models via actions, percepts, sensing actions and time-stepping, using a message passing protocol.

GAMA (Taillandier et al., 2019) is a platform that supports development of spatially explicit simulations with a wide variety of agents. It offers a complete development environment, 2D/3D visualizations, support for integrating Geographical Information System (GIS) data, and multi-level modeling. Modelers develop simulations in GAML, that supports agent control using finite state machines, task-based architectures, i.e. choosing a task on a number of strategies, reactive style rule-based architectures (i.e. reflex rules), and other control architectures that are available as additional plugins. Gamma supports BDI through BEN (Bourgais et al., 2020), explicitly handling beliefs, desires and intentions, however introducing also more advanced aspects of agency such as emotions and norms, integrating theories such as the OCEAN model for representing personality features, OCC model for emotions, explicitly representing social relations, thus leading to a rich representation of agent characteristics and interactions.

NetLogo (Tisue and Wilensky, 2004) is a versatile programming environment that enables the creation of complex system models, in an easy to use manner. An extensive library of models from various fields (economics, biology, physics, chemistry, psychology, etc.), combined with user friendliness, contributed to its wide adoption. Although NetLogo targets mainly simple agents, a number of approaches have been proposed towards supporting more complex behavioural models. Work reported in (Sakellariou, 2012) provides a domain specific language and an execution engine that allows the encoding of agents controlled by state machines. Other state based models, such as Stream X-Machines (Harwood, 1998), i.e. state-based machines extended with a memory structure and functions that guard transitions between states, have been introduced in NetLogo (Sakellariou et al., 2015), along with an approach to agent testing. A limited support for BDI agents and FIPA ACL message passing has been reported in (Sakellariou et al., 2008), that provides the necessary primitives through a set of procedures and functions. In all previous approaches, authors provided implementations in the NetLogo language. In contrast, (Luna-Ramirez and Fasli, 2018) integrates Jason (Bordini et al., 2007) (one of the most complete implementations of AgentSpeak in Java), with NetLogo to provide a disaster rescue scenario, by having the latter acting as the environment in the Jason platform and providing a link between the two.

Finally, it should be noted that the question of bridging the gap between simulation platforms and agent programming, by providing a framework to integrate "off-the-self" BDI systems and agent simulation platforms, has been addressed in (Singh et al., 2016). The latter describes a layered architecture to that allows any BDI programming language to "act as the brains" of simulated entities. In the same vein, an approach to multi-level simulation, integrating a cognitive BDI layer, a social layer based on diffusion theory in social networks and an agent based simulation platform, each corresponding to the cognitive, social and physical layers necessary, is described in (Bulumulla et al., 2022), that employees MATSim (Horni et al., 2016) as the simulation platform, as an example.

To our knowledge, an attempt to integrate the Teleo-Reactive paradigm in a "general purpose" simulation platform has not been reported before.

3 THE TELEO-REACTIVE APPROACH

The Teleo-Reactive approach was originally introduced by Nilsson (Nilsson, 1993), aiming to merge control theory and computer science notions, in order to offer an elegant way to encode agent behaviour in dynamic environments. Thus, a Teleo-Reactive sequence (TR) is an ordered list of production rules, guarded by environment conditions (K_i) that initiate actions (a_i).

$$K_1 \rightarrow a_1 \quad K_2 \rightarrow a_2 \quad \dots \quad K_m \rightarrow a_m$$

Rule ordering dictates a top-down *firing priority*, i.e. rules appearing early in the sequence have a higher priority. Thus, the condition K_m implicitly contains the negation of all previous rule conditions $\{K_j | j < m\}$.

Goal directed behaviour (*teleo*) is achieved by appropriately designing TR sequences to satisfy the *regression* property (Nilsson, 1993; Nilsson, 2001): each lower priority rule action is expected to achieve the condition of a higher priority rule, gradually leading to the achievement of the top-level goal, appearing as a condition of the first rule. Constantly monitoring the environment and evaluating rule conditions guarantees *reactivity* and ensures that the highest priority rule is always fired. Nilsson further defines *completeness*, as the property of TR sequences, in which the disjunction of all their rule conditions is a tautology. Sequences that satisfy both regression and completeness are *universal*, i.e. in the absence of sensing or action errors will eventually achieve their goal.

In TR programs, primitive actions are either *discrete*, i.e. are applied once the corresponding rule fires, or *durative*, i.e. actions applied while the corresponding rule condition is true. This distinction, that originates from the need to provide smooth action execution in continuous real world robotic environments, has some interesting implications in simulation environments, discussed in Section 4.2. Apart from primitive actions, the right hand side (RHS) of a rule can invoke a TR program. However, the latter does not follow the classical procedure invocation in programming languages, i.e. there is no "return" primitive: conditions are continuously assessed for the discovery of the next rule to be fired (Nilsson, 1993). Finally, TR programs can contain variables, bound at the execution time to offer generality in programming agent behaviour. The TR approach presents quite a few similarities with other reactive approaches, most notably that of Brooks (Brooks, 1986). An extensive comparison can be found in (Nilsson, 1993).

Although with a very appealing simplicity, TR has to overcome issues common in purely reactive architectures that arise due to complexity. Nilsson proposed the "Triple Tower" architecture (Nilsson, 2001), enhancing the simple TR approach, with a deductive rules component ("Percepts Tower") employed to populate the "Model Tower" with beliefs, used by the TR programs in "Actions Tower". Towards the same goal, Coffey and Clark (Coffey and Clark, 2006) proposed a hybrid architecture that combines BDI with TR, having the latter as plans in the former, yielding significant advantages, such as efficient intention suspension due to the stateless nature of TR programs.

TeleoR (Clark and Robinson, 2015) is an extension of the Nilsson's Teleo-Reactive paradigm, implemented in an extended logic programming language

Qu-Prolog (Quantifer Prolog) (Staples et al., 1989). TeleoR provides types and higher-order functions that create a safeguard for the correct instantiating of agent (robotic) actions to prevent run-time errors. Beliefs are handled via the QuLog (Robinson et al., 2003) (which runs on top of Qu-Prolog), offering a rich rule based language, to encode belief maintenance. Rules have been extended with *while* and *until* conditions, that allow rule commitment while a condition is true or until a condition becomes true (i.e. inferable from the belief base) respectively, within a TR procedure. The action part of TR rules has been extended to support parallel actions, time sequenced durative actions, i.e. an action sequence with predetermined time duration for actions, and *wait-repeat* annotations to allow repeating the application of an action in predetermined time intervals (or yield an error if the latter fails). Finally, message exchange between agents is supported via Pedro, a logic programming oriented message passing infrastructure.

There are numerous extensions to the TR paradigm, and also work that combines TR with other approaches. For instance, embedding statecharts in TR programs in order to overcome complexity issues is discussed in (Sánchez et al., 2017).

Formalising TR semantics has been also investigated in numerous approaches: in (Kowalski and Sadri, 2012) authors provide model theoretic semantics by representing TR programs in Abductive Logic Programming and (Dongol et al., 2010) employs temporal logic over continuous intervals to formalize TR semantics. A systematic literature review of 53 Teleo-Reactive-based studies until 2014 can be found in (Morales et al., 2014).

4 TELEO-REACTIVE AGENTS IN NetLogo

The requirements that underline the TR-TURTLES approach are as follows:

- offering modelers the ability to encode complex agent behaviour by embedding Teleo-Reactive rules in NetLogo syntax,
- low installation cost and an implementation approach that will survive the frequent evolutions of the NetLogo platform, with minimal to none changes,
- ease of use, small learning curve and compliance with the interactive style of model development in NetLogo and well as to its modeling approach.

Regarding the first requirement, the original TeleoR syntax has been modified slightly, while keeping the basic form of TR programs, as described the following sections. It has to be noted however, that the current implementation does not support a number of features that a full system like QuLog/TeleoR supports; some important features, such as supporting variables in rules, have still to be considered as their introduction involves some considerations and design choices.

The second requirement was addressed by developing the meta-interpreter in the modeling language of NetLogo. The latter has a functional flavour, supporting for instance anonymous functions, however provides a rather limited number of primitives for some common operations, such as strings. Although such an implementation choice appears to be an unconventional approach to that of extending the simulation platform's functionality via the provided JAVA/SCALA language API, we found it possible to develop the meta-interpreter using the facilities provided by the language. This decision might demand a higher programming effort, however, it presents two important benefits. The first is that the implementation is more likely to survive changes in the NetLogo platform as the latter evolves, since it is expected that newer versions will preserve backward compatibility in the NetLogo language or at least introduce minimal changes. Thus, it minimizes the number of dependencies. The second concerns the fact that it allows to encode agent sensors and actuators as NetLogo commands and reporters, the evaluation/execution of which is controlled by the meta-interpreter. Thus, solves the problem of interfacing TR-TURTLES to the underlying simulation platform. Similar approaches have been used in the past for encoding complex agent behaviour as reported for instance in (Sakellariou et al., 2008).

The third requirement is also addressed by having the meta-interpreter as a NetLogo command that each "intelligent" turtle invokes to determine its next action, simulations involving TR programs controlling NetLogo agents can be developed incrementally, following the interactive style programming that the NetLogo platform offers. Additionally, TR-TURTLES follows the basic simulation approach commonly found in all NetLogo models, that is to proceed the simulation in *cycles*, each typically corresponding to one time unit (*tick* in NetLogo terms), meaning that the agent can perform (again typically) a limited number of actions in each simulation cycle.

The overall architecture of TR-TURTLES is presented in Fig. 1. Briefly, the modeler has to provide the simulation environment, the agent sensors and effectors as NetLogo reporters and procedures. In order to specify the behaviour of a set of agents (breed in NetLogo terms) using TR-TURTLES, the modeler (a) defines the former in a NetLogo procedure using the syntax described in Fig. 2 (populating the *Rule Store* with the TR rules), and (b) provides the belief update "callback" function translating sensory input to beliefs to update the *Belief Store*. The execution of the agent is then controlled by TR-TURTLES: in each simulation cycle the meta-interpreter (*TR Execution Engine*) invokes the "callback" function and executes the respective agent actions.

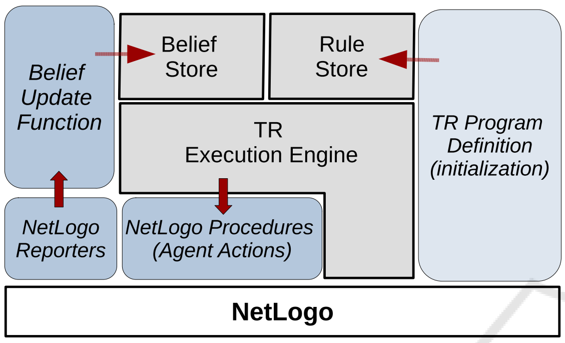

Figure 1: The overall TR-TURTLES architecture.

As mentioned in Section 3 a TeleoR program consists of a set of guarded action rules, organised in *procedures*. In TR-TURTLES we follow a syntax that is close to the one described in QuLog/TeleoR (Clark and Robinson, 2003; Clark and Robinson, 2015), and support many features introduced in the latter. The sections that follow present TR-TURTLES in more detail, starting with the guarded action rules description.

4.1 Guarded Action Rules

Probably the most important component of TR-TURTLES is the guarded action rule that has the following form:

```
# Condition --> Actions ++ [Side Effects] .
```

A `Condition` is a possible conjunction (&) of strings, each representing a potential match to a *belief* that exists in the Belief Store. *Actions* are string representations of NetLogo commands, i.e. the effectors of the agent, possibly arranged for parallel or sequential execution. Informally, an action can be either:

- A single action, e.g. `"move-forward"`.
- A list of actions enclosed in `"["` and `"]"`, executed in *parallel*, e.g `["blink" "move-forward"]`. Placing actions in such a list implies that the modeler considers that such actions can indeed be executed in parallel by the agent: this is a modeling choice.
- Time sequenced actions (Clark and Robinson, 2015), in which the sequential operator is `":"`, where each action is executed `for` N simulation steps (*ticks*), before proceeding to the next action in the list, e.g `"blink" : ["blink" "move-forward"] for 10 : "rotate" for 18 : "blink" : "move-forward" for 10`.
- *wait repeat* actions, that in the case that an action has not led to a change in the environment that leads to a different rule firing, are controllably repeated, e.g. `"ungrasp" wait-repeat 2 10`, imposes that the action `"ungrasp"` will be executed 10 times, with a 2 time units interval between tries. After that time has elapsed, TR-TURTLES will signal an error.
- a single procedure call.

For further clarification on how the above can be combined, the grammar of TR-TURTLES is provided in Fig. 2.

Side Effects are actions to be called at the end of the execution rule, and in the case of TR-TURTLES, can be any anonymous NetLogo command, even adding a belief in the store (see Section 4.3), that the agent would like to assert when an action is completed.

Procedures are collections of guarded action rules under a name, that can be called as the single action of a firing rule. Since, the latter do not terminate as in ordinary languages, i.e. by returning a value, a change in the belief base leads to a re-evaluation of the complete procedure stack. The modeler has to state the initial procedure to be activated, when the agent starts its execution via the `set-goal` keyword.

4.2 Discrete and Durative Actions

Given that the NetLogo simulation environment supports the notion of a time unit (*tick*), typically advancing in each simulation cycle, we assume that a *discrete* (or *ballistic*) action lasts for a single time tick, i.e. a single simulation circle. On the other hand a *durative* action can span its execution over multiple cycles, i.e. creating a sequence of "discrete instances" of the action that are executed until the rule is deselected: we follow the approach stated by Lee & Durfee (Lee and Durfee, 1994), where we consider as the "characteristic frequency" of the execution cycle and the environment to be equal, i.e. the simulation cycle frequency.

This can have a number of interesting implications. A list of parallel actions, can contain both discrete and durative actions. In the first execution of the corresponding rule, all actions get executed in a singe time step (thus in parallel), however, in subsequent executions only durative actions in the list are executed. Thus, an action list $[a_1^d, a_2, a_3^d, a_4 \ldots a_k^d]$ where

```
     ⟨TRProgram⟩  ::=  "tr-init" ⟨BeliefsD⟩ ⟨ActionD⟩ ⟨Procedures⟩
      ⟨BeliefsD⟩  ::=  "belief-update-function" ⟨NetLogoProc⟩
                       "beliefs" "[" ⟨Belief⟩* "]"
        ⟨Belief⟩  ::=  "string"
       ⟨ActionD⟩  ::=  "durative-actions" "["⟨ActionDescr⟩*"]"
                       "discrete-actions" "["⟨ActionDescr⟩*"]"
    ⟨Procedures⟩  ::=  ⟨Procedure⟩+ "set-goal" ⟨PName⟩
     ⟨Procedure⟩  ::=  "procedure" ⟨PName⟩ ⟨GuardedRule⟩* "end-procedure"
         ⟨PName⟩  ::=  "string"
   ⟨GuardedRule⟩  ::=  ⟨Guard⟩ "--->" ⟨Actions⟩ ("++" ⟨NetLogoProc⟩)? "."
         ⟨Guard⟩  ::=  ⟨Belief⟩ ("&" ⟨Belief⟩)* | "true"
       ⟨Actions⟩  ::=  ⟨AnotAction⟩ | ⟨PName⟩
    ⟨AnotAction⟩  ::=  ⟨Action⟩
                       | ⟨Action⟩ "wait-repeat" W R
                       | ⟨Action⟩ "for" N ( ":"⟨Action⟩( "for" N)?)*
        ⟨Action⟩  ::=  ⟨ActionDescr⟩ | "[" ⟨ActionDescr⟩+ "]"
   ⟨ActionDescr⟩  ::=  "string"
   ⟨NetLogoProc⟩  ::=  Anonymous NetLogo Procedure Call
```

Figure 2: TR-TURTLES grammar. Note that *W, R* and *N* are integers.

a_i^d, are durative actions, whereas a_j are discrete, is interpreted as a sequence of parallel actions of the form:

$$[a_1^d, a_2, a_3^d, a_4 \dots a_k^d]_{t1} : [a_1^d, a_3^d, \dots a_k^d]_{t_2} : \dots : [a_1^d, a_3^d, \dots a_k^d]_{t_m}$$

where $t_1 \dots t_m$ are the consecutive time points the corresponding rule is active, i.e. no changes happened in the belief base.

However, there is a crucial point that needs to be addressed in the above. If a change in the belief store did occur, and the same rule was selected once more, there are two options in order to handle action execution: either (a) continue with the current set of actions, as if no change occurred in the belief base, or (b) consider the rule as a new instance, and start the execution of the sequence, once more, including discrete actions. TR-TURTLES has adopted the first option (a) as discussed in (Clark and Robinson, 2020).

Thus, it is important that the modeler clearly defines the type of the corresponding action. This is achieved by having *action mode declarations* in the TR agent specification, listing each set of durative and discrete actions (Fig. 2). For instance:

```
durative-actions ["move-forward" "rotate"]
discrete-actions ["ungrasp" "grasp" "blink"]
```

4.3 Agent Perception, Beliefs and Updates

The handling of agent percepts must easily interface the agent with the NetLogo environment, while supporting the semantics of the TR approach. Thus, the modeler has to provide a belief update command, and include in the latter the detection of all the necessary environment "events" that the agent should observe (Fig. 1). Both the *belief update function* and the set of beliefs (for type checking) have to be declared in the TR-TURTLES program (Fig. 2). It should be noted that the meta-interpreter expects a NetLogo *anonymous* procedure in the declaration, in order to be stored in the appropriate structure and called in each cycle, automatically.

Upon the detection of such an environment event, the modeler has to update the store with the corresponding belief or inform the store that the specific belief should be removed. This is achieved by two commands that handle the belief addition/removal (`add-belief`, `no-belief` respectively) in the store and record whether a change occurred. For instance:

```
[add-belief "can-move-ahead"]
[no-belief "see-can"]
```

Currently, beliefs are represented as simple strings. Not supporting the versatility variables can offer, is indeed a major limitation of the current implementation, and is one of the main future extensions. However, adding variables poses a design choice: although introducing logical variables "ala Prolog", i.e. single assignment variables taking values via unification is the current most widely applied choice in agent languages (i.e. in Jason (Bordini et al., 2007) for instance), this certainly lies outside the programming vein that a NetLogo modeler is accustomed to and might present a challenge to the latter. We are currently investigating alternative design choices to support more general belief representations.

Beliefs form the guard part of each rule and can appear in conjunctions (Fig. 2). Currently, due to the lack of variables and the finite number of beliefs to be handled (due to the declaration) the meta-interpreter maintains in the store explicitly the truth value of each belief. However, with the introduction of variables, the approach adopted will most likely follow the closed world assumption, although the explicit negated beliefs such as those in Jason could be adopted.

4.4 Implementation

The TR-TURTLES meta-interpreter comes as a set of files (`*.nls`) to be included in any NetLogo model. This approach, allows users to deploy TR-TURTLES agents in parallel with agents following any other approach in the same simulation environment, as long as the model respects the semantics of the simulation cycle. In order to support the syntax described in Fig. 2, each punctuation symbol corresponds to a NetLogo reporter that builds the internal representation of the guarded rules, to be executed by the meta-interpreter. Currently, minimum type checking is performed in the belief update functions. The TR program that the modeler specifies is loaded *once* at the initialization of the agent: all guarded rules are translated to anonymous NetLogo procedures and stored for later execution in a turtle variable. In each simulation cycle, the meta-interpreter is invoked by "asking" the agent to execute the top-level command `execute-tr-rules`, provided by TR-TURTLES. The current implementation is publicly available in the GitHub repository[1].

5 COLLECTING CANS: AN ILLUSTRATIVE EXAMPLE

This section presents a simple simulation of an agent collecting cans in a room that originally appeared in (Dongol et al., 2010), slightly extended in order to demonstrate the application of more complex rules. In this simple setting, the agent collects cans in a room to deposit them to a number of bins (depots). In order to demonstrate the reactive features of the TR approach, depots move randomly inside the room and switch with a specified frequency between two states, those of accepting and not-accepting cans. However, this state is invisible to the robotic agent, i.e. the agent has no sensor to detect the state of the can, resulting in some cases to a failed action, i.e. one that does not change the environment. Robotic sensors for detecting the location of cans and depots have a limited range, implying that the robot has to engage in a wandering behaviour in order to find the respective positions. Thus, a robotic agent has to locate a can, pick it up, reach a depot, possibly follow it in case the depot moves away and try (multiple times) to drop the can.

The TR code specifying the behaviour of the agent is depicted in Fig. 3. All the TR-TURTLES code regarding single agent is inside a NetLogo procedure (named in this case `tr-code-of-robots`), that is to be called once during initialization of the respective turtle in NetLogo.

The belief update function, that is called by the TR-TURTLES meta-interpreter automatically in each simulation cycle, is specified using the keyword `belief-update-function`, followed by the belief declarations in the next line. The meta-interpreter assumes that the modeler will provide a NetLogo procedure under the name *update-robot-beliefs*, responsible for updating beliefs. For instance, in the example, the agent detects within its field of view a depot and populates the store with the corresponding belief, as shown below:

```
to update-robot-beliefs
 ifelse any? depots in-cone
                view-distance view-angle
   [add-belief "see-depot"]
   [no-belief "see-depot"]
   ...
end
```

Given that the above procedure can contain any NetLogo code, the modeler can encode any agent sensors, thus providing full integration of TR-TURTLES with the simulation environment. There are two durative actions in the model, `move-forward` and `rotate` that model motion. Discrete actions `grasp` and `ungrasp`, control the robotic arm, while `blink` is a dummy discrete action that changes the color of the agent to indicate that it is in the process of delivering a can. Each action is expected to appear as a NetLogo procedure in the model.

Finally, TR rules in this simple example are included in two procedures, the top-level one (`clean-cans`) and `wander`, the latter implementing the wandering behaviour of the agent in the cases that it is holding a can or looking for a can. It is interesting to note two rules in the above TR program. The first is a `wait-repeat` rule that attempts to deliver the can by trying the action `ungrasp` in 2 ticks intervals for ten times, and the first rule of procedure `wander`, that is a timed sequence of two actions (`move-forward` and `rotate`) that generates a wandering behaviour. The

[1]https://github.com/isakellariou/teleoTurtles

```
to tr-code-of-robots
 tr-init
 belief-update-function [[] -> update-robot-beliefs]
 beliefs ["holding" "at-depot" "see-depot" "see-can" "touching" "can-move-ahead"]
 durative-actions ["move-forward" "rotate"]
 discrete-actions ["ungrasp" "grasp" "blink"]
 procedure "clean-cans"
  # "holding" & "at-depot" --> "ungrasp" wait-repeat 2 10
                        ++ [[]-> show "At-deport - Delivered"] .
  # "holding" & "see-depot" & "can-move-ahead" --> ["blink" "move-forward"]  .
  # "holding" --> "wander" .
  # "touching" --> "grasp" .
  # "see-can" & "can-move-ahead" --> "move-forward" .
  # "true" --> "wander" .
 end-procedure
 procedure "wander"
  # "can-move-ahead" --> "move-forward" for 2 : "rotate" for 1 .
  # "true" --> "rotate".
 end-procedure
 set-goal "clean-cans"
end
```

Figure 3: The TR-TURTLES code for the robotic agent.

model's code is publicly available [2].

6 TR-Turtles EXECUTION OVERHEAD

Naturally, introducing an execution layer above the programming layer of NetLogo yields some performance penalties. In order to evaluate these penalties and assess the scalability of TR-TURTLES, the "Shepherds" NetLogo library model in TR-TURTLES was implemented and compared with the implementation provided in the distribution.

There were two sets of experiments carried on the same models. One set is concerned with evaluating the "core" library in terms of execution speed. In this set all GUI elements (plots, environment updates) were turned off, measuring the time (wall time) it takes to execute 4000 time units (ticks) in the two approaches. Additionally, the first set evaluates through a measure existing in the original "Shepherds" model, whether the two models produce the same behaviour. The second set of experiments measures execution time under active GUI elements. The experiments were conducted on a desktop machine with 8GB of RAM, Intel i7-8700 CPU (4.60GHz) processor and NetLogo 6.3.0.

For the first set, a varying number of "shepherds" (TR agents) were introduced in the simulation and for each such population one hundred different initial environments were initialised, by setting the random generator seed to specific values (ranging from 10 to 100, with step 10) and a varying number of sheep (ranging from 50 to 500, with step 50). Each combination of environment/number of agents was executed three times, creating a total, of 3000 experiments.

[2]https://github.com/isakellariou/teleoTurtles

Results are presented in Table 1, where column "Mean" depicts the average execution time, along with its standard deviation (STD), whereas column "Median" depicts the median value for experiments, each for a different number of shepherds.

As expected, the TR-TURTLES meta-interpreter introduces an overhead. However, the execution time grows linearly with respect to the number of TR agents, as shown in Fig. 4. The current implementation is not optimised in a number of ways: for instance, type checking beliefs is done at runtime, introducing a time penalty in every addition of a belief.

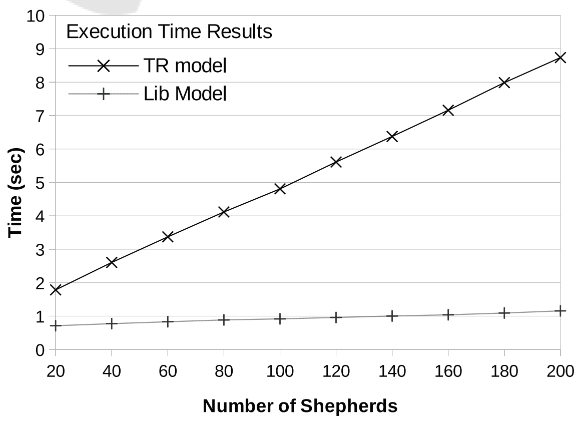

Figure 4: Execution time w.r.t. the number of agents in the Shepherds model and diverse seed.

In order to access how close the two models are in

Table 1: Execution time (wall time in sec) results (no GUI elements,4000 ticks, diverse seed values and number of sheep).

	Lib Model			**TR Model**		
Shepherds	Mean	STD	Median	Mean	STD	Median
20	0.71	0.13	0.71	1.78	0.23	1.79
40	0.77	0.13	0.77	2.60	0.27	2.61
60	0.83	0.12	0.83	3.37	0.29	3.37
80	0.88	0.13	0.88	4.11	0.32	4.13
100	0.91	0.12	0.91	4.80	0.36	4.81
120	0.95	0.12	0.95	5.61	0.37	5.62
140	1.00	0.12	1.00	6.37	0.40	6.40
160	1.03	0.12	1.03	7.15	0.46	7.19
180	1.09	0.12	1.08	7.98	0.50	8.01
200	1.15	0.13	1.14	8.73	0.58	8.76

terms of agent behaviour, we are using a measure that exists in the original simulation, *herding efficiency*, that is computed *by counting the number of patches that have no sheep in their neighborhood.* As shown in Table 2, herding efficiency results are very close, an indication that the two models produce the same behaviour. The large standard deviation reported in experiments with a small number of agents, is attributed to the fact that a small number of shepherds in the model cannot efficiently handle a large number of sheep. It is interesting to note that the standard deviation is very close in both versions of the model as shown in the table.

Table 2: Efficiency (%) results in simulations with no GUI elements (4000 ticks), diverse seed values and number of sheep.

	Lib Model		**TR Model**	
Sheph	Mean	STD	Mean	STD
20	84.19	6.97	84.17	7.06
40	90.46	3.80	90.14	3.88
60	92.83	2.48	92.98	2.55
80	94.06	1.81	94.18	1.81
100	94.53	1.56	94.30	1.48
120	94.55	1.30	94.51	1.28
140	94.69	1.23	94.55	1.21
160	94.68	1.34	94.76	1.31
180	94.42	1.34	94.51	1.14
200	94.38	1.33	94.37	1.51

However, in an interactive simulation environment such as NetLogo, far more operations occur regarding visualization and experiment monitoring. To provide an estimate of how, in such cases, a modeler would perceive in terms of time penalty the use TR-TURTLES, we conducted experiments in the same simulation, however measuring time for a smaller number of ticks. Results are summarized in Table 3 and show that in such interactive simulations, the execution time difference is negligible, leading to the conclusion that in these cases the time spend in other operations taxes more the overall execution than the introduction of the meta-interpreter.

This second set, also evaluates how close the two models are, and as shown in Table 3, herding efficiency results are very close, an indication that the two models produce the same behaviour. The observant reader might notice that standard deviation is not listed for the herding efficiency results, in this case: all experiments were conducted with the same random seed, yielding exactly the same value, attributed to the "reproduciblity of experiments" property of NetLogo.

7 CONCLUSIONS AND FUTURE WORK

The introduction of new agent programming languages in ABMS simulation environments can be of great importance, since it enables modelers to enrich simulations with more complex agent behaviours. Such richer models can indeed prove to be valuable in certain fields, since they alleviate the need for model simplifications or increased programming effort of complex models. To the above end, the current work presents a first attempt to introduce the Teleo-Reactive paradigm in NetLogo, offering a meta-interpreter and extending the syntax of the underlying platform to encode TR rules.

We consider that robotic agents have a number of common characteristics with agents in a simulation environment that has a spatial dimension. They both share movement in a dynamic environment, possibly non deterministic actions, require replanning and goal monitoring, which is easily offered by the TR approach (and not so easily in a BDI approach to our opinion). As a first example of its wider applicability for general purpose case studies, we chose to re-

Table 3: Execution time (wall time in sec) results in simulations with GUI elements (1000 ticks).

Shepherds	Lib Av. Time	STD Lib	Efficiency	TR Av. Time	STD TR	Efficiency
50	34.26	0.034	71.35	34.32	0.026	71.06
100	34.29	0.026	82.43	34.27	0.036	81.4
150	34.39	0.037	85.57	34.33	0.055	87.07
200	34.29	0.015	88.44	34.24	0.009	86.37

implement the Shepherds model of the NetLogo library, however we plan to investigate on more models in the future.

The current implementation can be extended in a number of ways. The most important extension concerns the introduction of rule variables that will allow encoding more complex agent behaviour. Such an introduction demands extensive parsing machinery, variable operations such as unification with predicates in the belief base, in order to impose the necessary semantics regarding variable scope and perform type checking as the QuLog/TeleoR language. Another issue, concerns support for while/until rules which TR-TURTLES lacks. Reducing the execution overhead of the current TR approach, by identifying and introducing optimizations is also an important research direction, that can be significant when dealing with large scale simulations. Finally, exploring integration of the TR paradigm with other approaches, such as BDI, as in (Coffey and Clark, 2006), can lead to a number of interesting research results.

As a final note, ABMS platforms can provide an excellent testbed for investigating issues with respect to agent programming languages, since they offer rapid creation complex environments to test ideas on the latter.

REFERENCES

Abar, S., Theodoropoulos, G. K., Lemarinier, P., and O'Hare, G. M. (2017). Agent based modelling and simulation tools: A review of the state-of-art software. *Computer Science Review*, 24:13–33.

Adam, C. and Gaudou, B. (2016). Bdi agents in social simulations: a survey. *The Knowledge Engineering Review*, 31(3):207–238.

Bordini, R. H., Hübner, J. F., and Wooldridge, M. (2007). *Programming multi-agent systems in AgentSpeak using Jason.* John Wiley & Sons.

Bourgais, M., Taillandier, P., and Vercouter, L. (2020). Ben: An architecture for the behavior of social agents. *Journal of Artificial Societies and Social Simulation*, 23(4):12.

Brooks, R. (1986). A robust layered control system for a mobile robot. *IEEE Journal on Robotics and Automation*, 2(1):14–23.

Bulumulla, C., Singh, D., Padgham, L., and Chan, J. (2022). Multi-level simulation of the physical, cognitive and social. *Computers, Environment and Urban Systems*, 93:101756.

Cardoso, H. L. (2015). Sajas: Enabling jade-based simulations. In Nguyen, N. T., Kowalczyk, R., Duval, B., van den Herik, J., Loiseau, S., and Filipe, J., editors, *Transactions on Computational Collective Intelligence XX*, pages 158–178. Springer International Publishing, Cham.

Clark, K. L. and Robinson, P. J. (2003). Qulog: A flexibly typed logic based language with function and action rules. Technical report, Imperial College London.

Clark, K. L. and Robinson, P. J. (2015). Robotic agent programming in teleor. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5040–5047.

Clark, K. L. and Robinson, P. J. (2020). Chapter 3: introduction to qulog. *Programming Communicating Robotic Agents: A Multi-tasking Teleo-Reactive Approach*.

Coffey, S. and Clark, K. (2006). A hybrid, teleo-reactive architecture for robot control. In *Proceedings of the 2nd International Workshop on Multi-Agent Robotic Systems - Volume 1: MARS, (ICINCO 2006)*, pages 54–65. INSTICC, SciTePress.

Collier, N. and North, M. (2013). Parallel agent-based simulation with repast for high performance computing. *SIMULATION*, 89(10):1215–1235.

Dongol, B., Hayes, I. J., and Robinson, P. J. (2010). Reasoning about real-time teleo-reactive programs. Technical report, The University of Queensland, QLD, 4072, Australia.

García, E., Rodríguez, S., Martín, B., Zato, C., and Pérez, B. (2011). Misia: Middleware infrastructure to simulate intelligent agents. In Abraham, A., Corchado, J. M., González, S. R., and De Paz Santana, J. F., editors, *International Symposium on Distributed Computing and Artificial Intelligence*, pages 107–116, Berlin, Heidelberg. Springer Berlin Heidelberg.

Gormer, J., Homoceanu, G., Mumme, C., Huhn, M., and Muller, J. P. (2011). Jrep: Extending repast simphony for jade agent behavior components. In *2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology*, volume 2, pages 149–154.

Harwood, C. (1998). *Correct Systems: Building a Business Process Solution*, page 206. Springer London.

Horni, A., Nagel, K., and Axhausen, K. W. (2016). *The Multi-Agent Transport Simulation MATSim*. Ubiquity Press, London, GBR.

Kowalski, R. A. and Sadri, F. (2012). Teleo-reactive abductive logic programs. In *Logic Programs, Norms and Action: Essays in Honor of Marek J. Sergot on the Occasion of His 60th Birthday*, pages 12–32. Springer Berlin Heidelberg, Berlin, Heidelberg.

Kravari, K. and Bassiliades, N. (2015). A survey of agent platforms. *Journal of Artificial Societies and Social Simulation*, 18(1):11.

Lee, J. and Durfee, E. H. (1994). Structured circuit semantics for reactive plan execution systems. In *Proceedings of the Twelfth AAAI National Conference on Artificial Intelligence*, AAAI'94, page 1232–1237. AAAI Press.

Luna-Ramirez, W. A. and Fasli, M. (2018). Bridging the gap between abm and mas: A disaster-rescue simulation using jason and netlogo. *Computers*, 7(2).

Morales, J. L., Sánchez, P., and Alonso, D. (2014). A systematic literature review of the teleo-reactive paradigm. *Artificial Intelligence Review*, 42(4):945–964.

Nilsson, N. (1993). Teleo-reactive programs for agent control. *Journal of artificial intelligence research*, 1:139–158.

Nilsson, N. J. (2001). Teleo-reactive programs and the triple-tower architecture. *Electron. Trans. Artif. Intell.*, 5(B):99–110.

North, M. J., Collier, N. T., Ozik, J., Tatara, E. R., Macal, C. M., Bragen, M., and Sydelko, P. (2013). Complex adaptive systems modeling with repast simphony. *Complex Adaptive Systems Modeling*, 1(1):3.

Ozik, J., Collier, N., Combs, T., Macal, C. M., and North, M. (2015). Repast simphony statecharts. *Journal of Artificial Societies and Social Simulation*, 18(3):11.

Padgham, L., Nagel, K., Singh, D., and Chen, Q. (2014). Integrating bdi agents into a matsim simulation. In *Proceedings of the Twenty-First European Conference on Artificial Intelligence*, ECAI'14, page 681–686, NLD. IOS Press.

Padgham, L., Scerri, D., Jayatilleke, G., and Hickmott, S. (2011). Integrating bdi reasoning into agent based modeling and simulation. In *Proceedings of the 2011 Winter Simulation Conference (WSC)*, pages 345–356.

Railsback, S. F., Lytinen, S. L., and Jackson, S. K. (2006). Agent-based simulation platforms: Review and development recommendations. *SIMULATION*, 82(9):609–623.

Rao, A. S., Georgeff, M. P., et al. (1995). Bdi agents: from theory to practice. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS95)*, volume 95, pages 312–319. Association for the Advancement of Artificial Intelligence.

Robinson, P. J., Hinchey, M., and Clark, K. (2003). Quprolog: An implementation language for agents with advanced reasoning capabilities. In *Formal Approaches to Agent-Based Systems*, pages 162–172, Berlin, Heidelberg. Springer Berlin Heidelberg.

Sakellariou, I. (2012). TURTLES AS STATE MACHINES - Agent Programming in NetLogo using State Machines:. In *Proceedings of the 4th International Conference on Agents and Artificial Intelligence*, pages 375–378, Vilamoura, Algarve, Portugal. SciTePress - Science and and Technology Publications.

Sakellariou, I., Dranidis, D., Ntika, M., and Kefalas, P. (2015). From formal modelling to agent simulation execution and testing. In *Proceedings of the International Conference on Agents and Artificial Intelligence - Volume 1*, ICAART 2015, page 87–98, Setubal, PRT. SCITEPRESS - Science and Technology Publications, Lda.

Sakellariou, I., Kefalas, P., and Stamatopoulou, I. (2008). Enhancing netlogo to simulate bdi communicating agents. In *Artificial Intelligence: Theories, Models and Applications*, pages 263–275, Berlin, Heidelberg. Springer Berlin Heidelberg.

Singh, D., Padgham, L., and Logan, B. (2016). Integrating BDI Agents with Agent-Based Simulation Platforms. *Autonomous Agents and Multi-Agent Systems*, 30(6):1050–1071.

Sánchez, P., Álvarez, B., Martínez, R., and Iborra, A. (2017). Embedding statecharts into teleo-reactive programs to model interactions between agents. *Journal of Systems and Software*, 131:78–97.

Staples, J., Robinson, P. J., Paterson, R. A., Hagen, R. A., Craddock, A. J., and Wallis, P. C. (1989). Quprolog: An extended prolog for meta level programming. In *Meta-Programming in Logic Programming*, page 435–452. MIT Press, Cambridge, MA, USA.

Taillandier, P., Gaudou, B., Grignard, A., Huynh, Q.-N., Marilleau, N., Caillou, P., Philippon, D., and Drogoul, A. (2019). Building, composing and experimenting complex spatial models with the gama platform. *GeoInformatica*, 23(2):299–322.

Tisue, S. and Wilensky, U. (2004). Netlogo: Design and implementation of a multi-agent modeling environment. In *Proceedings of the Agent 2004 Conference on Social Dynamics: Interaction, Reflexivity and Emergence*, volume 2004. Argonne National Laboratory and University of Chicago.