Temporal Assertions for the Specification, Verification and Execution of Agent Populations

Wamberto Weber Vasconcelos
Division of Informatics, The University of Edinburgh
80 South Bridge, EH1 1HN Scotland, UK
wvasconcelos@acm.org

1 Introduction

Our aim is to define a simple framework within which software practitioners can set up experiments using *populations of agents*. The envisaged scenario of our framework is one in which a designer of Multi-Agent Systems (MASs, for short) wishes to experiment with a view to understand the dynamics of populations of agents.

Our framework comprises a specification language for temporal properties of agents and a means for verifying and executing specifications. The language is amenable for verification of desirable properties of individual agents and their populations by means of model-checking techniques. The specification is executable (directly or via alternative formats) since we want to observe the actual behaviour of the specified systems.

We propose a seamless representation for temporal and non-temporal aspects of MASs. In this approach, a MAS is specified (and later verified and optionally run) via a set of specifications of its component agents. Each agent is specified as a pair $\Sigma = \langle \Gamma, \Pi \rangle$, where Γ is a set of temporal assertions, explained below, and Π is an associated program. We have exploited this approach in the context of computational logic, and our programs Π are logic programs though any programming paradigm or language can be used instead. Γ specifies the temporal properties Π should possess throughout its execution. However, we shall employ Γ to *guide* the execution of Π, hence guaranteeing that the specified properties are observed throughout Π's execution.

As an initial example, let us consider the following specification which formalises the behaviours we expect from an agent to manage funds for loans:

$$\Sigma = \left\langle \boxed{\begin{array}{rcl} \boxed{start} & \Rightarrow & \bigcirc\ \boxed{funds(1000)} \\ \boxed{funds(F) \wedge F = 0} & \Rightarrow & \bigcirc\ \boxed{dead} \\ \boxed{\begin{array}{r} funds(F) \wedge \mathtt{request}(Client, Loan) \wedge \\ \mathtt{loanOK}(Client, Loan, F) \wedge NewF\ \mathtt{is}\ (F + F * 0.01) - Loan \end{array}} & \Rightarrow & \bigcirc\ \boxed{funds(NewF)} \end{array}}, \boxed{\begin{array}{l} \mathtt{request(Client,Loan)} \leftarrow \\ \quad \vdots \\ \mathtt{loanOK(Client,Loan,F)} \leftarrow \\ \quad \vdots \end{array}} \right\rangle$$

The left-hand side box is Γ, the agent's set of temporal assertions, and the right-hand side is Π, its associated logic program. The "$\bigcirc$" symbol stands for the "next time" temporal operator: $\bigcirc p$ means that p is true in the next state of the computation. The first formula in Γ establishes the initial amount of funds for the agent. The second formula represents the conditions for the termination of the agent: when its funds are depleted to zero. The third formula captures the behaviour of the agent concerning its funds and how they should be updated. Predicate `request/2` defined in Π holds if the agent receives a request for an amount *Loan* from *Client* – if there are no such requests, suitable "dummy" values are returned. Predicate `loanOK/3` holds if the agent agrees, based on its knowledge of the *Client*, its present funds F and the amount *Loan* asked for, to lend the desired amount. The third formula updates the funds by adding 1% to them and subtracting the amount *Loan* from them.

The example above illustrates how one can keep the temporal and non-temporal aspects separated. The (non-temporal) reasoning of an agent may become very sophisticated thus rendering its automatic analysis difficult if not impossible. Since the behaviours of predicates `request/2` and `loanOK/3` do not change over time, their definition can be neatly hidden away in Π and employed only when needed. The temporal aspects of the agent are isolated in Γ and can be studied independently from Π.

The temporal properties in Γ are guaranteed to be observed because, as we shall see below, they ultimately control the execution of the agent. The formulae are called *temporal assertions* because they will be presumed true and a model for the agent's execution, *i.e.* its history, will be constructed step-by-step, if it is possible. An important issue concerns the syntactic restrictions imposed on the possible temporal constructs: these rule out undesirable computational problems involved when executing specifications.

One of our premises concerns the design of the individual agents. We shall assume that design issues can be simplified to a small set of distinct fundamental aspects and their combinations. These aspects are our *features*, described below, and can be exploited within pre-determined boundaries. Such features comprise the essence of agents: whatever the purpose of the actual system is, whatever the particular details of each information agent, the system can still be usefully modeled, verified and executed.

Populations of agents are specified in bottom-up style. Initially, an *agent species* is defined through a set of features. The definition of an agent species has slots/parameters to be input by the designer, giving rise to *breeds* of agents, that is, variations of a species. A breed is a complete agent specification, without any undefined slots. We can build sets of individuals from one particular breed defining a population (of a breed). Each individual agent of a population is an independent and autonomous entity.

We have employed a simplified variant of executable linear temporal logics (LTL) *à la* MetateM [2] as our formalism. The division of a specification into features, species, breeds and populations and their relationship is for the purpose of organisation and to provide a methodology for the preparation of specifications. Our specifications can simply be seen as sets of executable LTL formulae.

In Section 2 we introduce the syntax and semantics of our temporal logic. In Section 3 we formally define how a specification in our temporal logic can be executed. The rationale behind the design decision of our temporal logic is laid out in Section 4. We explain our approach to specifying features of individual species of agents in Section 5 and in Section 6 we show how such features can be methodically designed via attitudes and behaviours. Another important issue concerns the verification of specifications, exploited in Section 7. In Section 8 we describe our approach to defining populations of agents. Finally, in Section 9 we conclude and give directions for future work.

2 A Temporal Logic of Assertions (TeLA)

Our Temporal Logic of Assertions, TeLA for short, can be seen as a scaled-down version of MetateM [2]. Our goal is to reach an adequate balance of expressiveness/pragmatics (offering useful and practical constructs) and computational efficiency (when executing our specifications). Our temporal assertions are of the form:

$$\underbrace{\boxed{\psi_1 \cdots \psi_n}}_{present} \Rightarrow \bigcirc \underbrace{\boxed{\phi_1 \cdots \phi_m}}_{future}$$

where $\psi_1 \cdots \psi_n$ are non-temporal subformulae, that is, there are no temporal operators in them, and $\phi_1 \cdots \phi_m$ are subformulae possibly with temporal operators that refer to the future. We give more details of the syntax and intended semantics of our temporal assertions in the sequel.

2.1 Syntax

Initially, we define a language $\mathcal{T}$ of *terms*. Constants, denoted by $a, b, c, d \in \mathcal{T}$, and variables, denoted by $w, x, y, z \in \mathcal{T}$, are terms. The application of a function f^n (where n is the number of arguments or arity of the function) to any terms $t_1, \ldots, t_n$, denoted by $f^n(t_1, \ldots, t_n)$, is also a term. Other alternative functional symbols are g^n, h^n. Terms consisting of functions and subterms are also called *composed* terms. If it is clear from the context, the superscript of functions can be left out. When necessary, constants, variables and functional symbols can appear subscripted. Standard mathematical constants, like integers, reals, etc., and functions like addition, division, etc. may be used with their usual definition as their intended interpretation.

Next we define the language $\mathcal{L}_+$ of *positive* (or non-negative) *literals* and the language $\mathcal{L}_-$ of *negative literals*. The symbols $\top, \bot \in \mathcal{L}_+$, representing truth and falsity, are positive literals. If $t_1, \ldots, t_n \in \mathcal{T}$ are terms then the application of predicate symbol p^n (where n is the arity of p) to them, that is, $p^n(t_1, \ldots, t_n) \in \mathcal{L}_+$, is a positive literal. Alternative predicate symbols are q^n, r^n and s^n; when necessary, predicate symbols can appear subscripted and, if clear from context, their superscript may be left out. Standard mathematical relations such as $>$, $<$, $\geq$, etc., may be used as infix predicate symbols.

The language $\mathcal{L}_-$ of negative literals is defined from $\mathcal{L}_+$: if $l \in \mathcal{L}_+$ then $\neg l \in \mathcal{L}_-$. That is, the negative literals are all the positive literals preceded by exactly one occurrence of the $\neg$ unary operator, no nestings being allowed. The union of positive and negative literals defines the language $\mathcal{L} = \mathcal{L}_+ \cup \mathcal{L}_-$ of all literals.

We define our language $\mathcal{R}$ of *basic temporal formulae* using the language $\mathcal{L}_+$ of positive literals. If $l \in \mathcal{L}_+$ then $l \in \mathcal{R}$, that is, any positive literal is a basic temporal formula. We also define sequences of conjuncts and disjuncts as elements of $\mathcal{R}$: if $\varphi_1, \varphi_2 \in \mathcal{R}$ then $\varphi_1 \wedge \varphi_2 \in \mathcal{R}$ and $\varphi_1 \vee \varphi_2 \in \mathcal{R}$. We now introduce the modal operators to those formulae in $\mathcal{R}$: if $\varphi \in \mathcal{R}$ then $\bigcirc\varphi \in \mathcal{R}$, $\Diamond\varphi \in \mathcal{R}$ and $\Box\varphi \in \mathcal{R}$. It is important to notice that we do not allow negative literals in $\mathcal{R}$.

Finally, we define our language $\mathcal{X}$ of *temporal logic assertions*. A temporal logic assertion is any construct of the form $\phi \Rightarrow \bigcirc(\varphi)$, where ϕ is a conjunction/disjunction of (positive or negative) literals, and $\varphi \in \mathcal{R}$ is a basic temporal formula. We depict in Figure 1 a context-free grammar for our constructs. To improve visualisation, we shall enclose

$$
\begin{array}{rcl}
\textit{Constant} & \rightarrow & a \mid b \mid c \mid d \\
\textit{Variable} & \rightarrow & w \mid x \mid y \mid z \\
\textit{Function} & \rightarrow & f^n \mid g^n \mid h^n \\
\textit{Term} & \rightarrow & \textit{Constant} \mid \textit{Variable} \mid \textit{Function}(\textit{Term}, \ldots, \textit{Term}) \\
\textit{Predicate} & \rightarrow & p^n \mid q^n \mid r^n \mid s^n \\
\textit{PositiveLiteral} & \rightarrow & \top \mid \bot \mid \textit{Predicate}(\textit{Term}, \ldots, \textit{Term}) \\
\textit{NegativeLiteral} & \rightarrow & \neg \textit{PositiveLiteral} \\
\textit{Literal} & \rightarrow & \textit{PositiveLiteral} \mid \textit{NegativeLiteral} \\
\textit{BasicTemporalFormula} & \rightarrow & \textit{PositiveLiteral} \mid \\
 & & \textit{BasicTemporalFormula} \wedge \textit{BasicTemporalFormula} \mid \\
 & & \textit{BasicTemporalFormula} \vee \textit{BasicTemporalFormula} \mid \\
 & & \bigcirc \textit{BasicTemporalFormula} \mid \Diamond \textit{BasicTemporalFormula} \mid \Box \textit{BasicTemporalFormula} \\
\textit{NonTemporalFormula} & \rightarrow & \textit{Literal} \mid \\
 & & \textit{NonTemporalFormula} \wedge \textit{NonTemporalFormula} \mid \\
 & & \textit{NonTemporalFormula} \vee \textit{NonTemporalFormula} \mid \\
\textit{TemporalAssertion} & \rightarrow & \textit{NonTemporalFormula} \Rightarrow \bigcirc \textit{BasicTemporalFormula}
\end{array}
$$

Figure 1: Context-Free Grammar for TeLA

the subformulae in the left- and right-hand side of the $\Rightarrow$ operator in boxes.

2.2 Semantics

A general framework to define the semantics of temporal logics is a *Kripke structure* [8, 9, 10] which is a *collection* of first-order interpretations. We shall employ this approach to define the semantics of our language $\mathcal{X}$ of temporal logic assertions.

We initially define a classic interpretation for first-order logic as $\mathcal{I} = \langle D, \mathbf{F}, \mathbf{P} \rangle$, where D is the *domain* or *universe of discourse*, $\mathbf{F}$ is a mapping between the terms t of a well-formed formula and $d \in D, \mathbf{F} : t \longmapsto d \in D$, and $\mathbf{P}$, a mapping between the atomic formulae $p^n(\mathbf{F}(t_1), \ldots, \mathbf{F}(t_n))$ (with its terms evaluated by $\mathbf{F}$) and $\{\mathbf{true}, \mathbf{false}\}, \mathbf{P} : p^n(\mathbf{F}(t_1), \ldots, \mathbf{F}(t_n)) \longmapsto \{\mathbf{true}, \mathbf{false}\}$. Given a well-formed formula in first-order logic, its distinct interpretations will form a collection $\mathcal{I}_1, \ldots,$ an infinite set if the domain is infinite. We are only interested on those collections of interpretations in which components D and and $\mathbf{F}$ remain unchanged and the only varying part is $\mathbf{P}$.

A Kripke structure for modal (temporal) logics is a collection of classic interpretations as above plus some relation among them. In a Kripke structure each interpretation is more commonly called a *possible world* or *possible state* and is also denoted simply as S. Let $\mathbf{S} = \{S_0, \ldots, S_n\}$ be the set of worlds or states of a Kripke structure. We relate our states via a *reachability relationship* $\rho : \mathbf{S} \times \mathbf{S}$ such that $(S, T) \in \rho, S, T \in \mathbf{S}$, if, and only if, T is ρ-accessible from S. We also employ $\rho(S, T)$ to denote that $(S, T) \in \rho$. A Kripke structure $\mathbf{K}$ is defined as the pair $\langle \mathbf{S}, \rho \rangle$.

Our relation ρ should reflect the (generally accepted) intuitive notions about time, that is, ρ is:

1. *reflexive*, so that every state is ρ-reachable from itself, that is, $\rho(S, S), S \in \mathbf{S}$.
2. *transitive*, so that for any three states S, T, U, if T is ρ-reachable from S and U is ρ-reachable from T,then U is ρ-reachable from S, that is, $\rho(S, T) \wedge \rho(T, U) \rightarrow \rho(S, U), S, T, U \in \mathbf{S}$.

Let us denote the reflexive and transitive closure of our set ρ as ρ^*: $\rho^*(S, T)$ holds iff $(S, T) \in \rho^*$. Clearly, $\rho(S, T) \rightarrow \rho^*(S, T)$ but not the converse. We also assume that ρ is

3. *discrete*: it is always possible to find two distinct states $S \neq T, \rho(S, T)$, so that there are no intermediate states U, that is $\exists S \exists T \forall U [\rho(S, T) \wedge \neg(\rho^*(S, U) \wedge \rho^*(U, T))]$
4. *infinite*: for any state S, there is another distinct state T ρ-accessible, that is, $\forall S \exists T (S \neq T \wedge \rho^*(S, T))$

and

5. has a *smallest element* S_0 from which every other state is ρ-accessible, that is, $\exists S_0 \forall S_i \rho^*(S_0, S_i)$

We employ $\mathbf{K}(S, \varphi)$ as an evaluation function mapping a temporal logic assertion $\varphi \in \mathcal{X}$ onto the truth-values $\{\mathbf{true}, \mathbf{false}\}$ using interpretation (or world or state) S. $\mathbf{K}(S, \varphi)$ is defined as below:

1. $\mathbf{K}(S, \top) = \mathbf{true}, \mathbf{K}(S, \bot) = \mathbf{false}$, for any $S \in \mathbf{S}$.
2. $\mathbf{K}(S, p^n(t_1, \ldots, t_n)) = \mathbf{true}$ iff $\mathbf{P}(p^n(\mathbf{F}(t_1), \ldots, \mathbf{F}(t_n))) = \mathbf{true}$, $S = \mathcal{I} = \langle D, \mathbf{F}, \mathbf{P} \rangle$.
3. $\mathbf{K}(S, \neg\varphi) = \mathbf{true}$ iff $\mathbf{K}(S, \varphi) = \mathbf{false}$.
4. $\mathbf{K}(S, \varphi_1 \wedge \varphi_2) = \mathbf{true}$ iff $\mathbf{K}(S, \varphi_1) = \mathbf{K}(S, \varphi_2) = \mathbf{true}$.
5. $\mathbf{K}(S, \varphi_1 \vee \varphi_2) = \mathbf{true}$ iff $\mathbf{K}(S, \varphi_1) = \mathbf{true}$ or $\mathbf{K}(S, \varphi_2) = \mathbf{true}$.
6. $\mathbf{K}(S, \varphi_1 \Rightarrow \varphi_2) = \mathbf{true}$ iff $\mathbf{K}(S, \varphi_1) = \mathbf{K}(S, \varphi_2) = \mathbf{true}$ or $\mathbf{K}(S, \varphi_1) = \mathbf{false}$.
7. $\mathbf{K}(S, \bigcirc\varphi) = \mathbf{true}$ iff $\mathbf{K}(T, \varphi) = \mathbf{true}$ for some $T \in \mathbf{S}, \rho(S, T)$.
8. $\mathbf{K}(S, \Diamond\varphi) = \mathbf{true}$ iff $\mathbf{K}(T, \varphi) = \mathbf{true}$ for some $T \in \mathbf{S}, \rho^*(S, T)$.
9. $\mathbf{K}(S, \Box\varphi) = \mathbf{true}$ iff $\mathbf{K}(T, \varphi) = \mathbf{true}$ for all $T \in \mathbf{S}, \rho^*(S, T)$.

We will consider all variables implicitly universally quantified, that is $\mathbf{F}(x) = d$ for all $d \in D$.

3 Executing TeLA Specifications

An agent (or, more correctly as we shall see below, an *agent species*) is specified by the pair $\Sigma = \langle \Gamma, \Pi \rangle$, where $\Gamma \subset \mathcal{X}$ is a set of temporal assertions and Π is an ordinary logic program. Σ may be understood as a formal means to execute program Π guaranteeing that the properties Γ are preserved. Alternatively, Σ can be interpreted as the specification of a restricted sort of execution (that in which Γ hold) of Π.

Given a specification $\Sigma = \langle \Gamma, \Pi \rangle$ we can execute Σ by constructively building a Kripke structure $\mathbf{K} = \langle \mathbf{S}, \rho \rangle$ such that $\mathbf{K}(S, \Gamma) = \mathbf{true}, S \in \mathbf{S}$. We start with a given initial state S_0 in which nothing is assumed true, that is, $\mathbf{P}(p^n(\mathbf{F}(t_1), \ldots, \mathbf{F}(t_n))) = \mathbf{false}$, for any $p^n(\mathbf{F}(t_1), \ldots, \mathbf{F}(t_n))$. We shall adopt the view of states/interpretations being built as *sets* of positive literals: $\mathbf{P}(p^n(\mathbf{F}(t_1), \ldots, \mathbf{F}(t_n)) = \mathbf{true}$ iff $p^n(\mathbf{F}(t_1), \ldots, \mathbf{F}(t_n)) \in S$. This convention makes the description of states more economic – those literals that are false simply need not be represented. If a literal is not to be found in a state, it is considered false, in the same fashion of the closed-world assumption of logic programming [1, 7].

Our execution of Σ proceeds as follows: given a state S_i, for every $\gamma \in \Gamma$ of the form $\phi \Rightarrow \bigcirc\varphi$ such that ϕ, a first-order (non-temporal) conjunction/disjunction, can be proved using S_i and Π (see below) then use all the non-temporal literals in φ to build S_{i+1}. If there are any temporal subformulae in φ then they are to be carried over to future states $S_{i+1}, S_{i+2}, \ldots$ and attempts will be made to satisfy them at every possible opportunity.

Initially we extend $\mathbf{K}$ to take into account both a state and an associated logic program:

Definition 3.1. Given a state S and a logic program Π we define $\mathbf{K}(S,\Pi,\gamma) = \mathbf{true}$ where γ is a temporal assertion or a subformula of a temporal assertion iff

1. $\mathbf{K}(S,\gamma) = \mathbf{true}$ or
2. $\gamma \in \mathcal{L}$ and $\Pi \vdash \gamma$.

This definition employs the standard Kripke semantics to map a formula onto its truth values using the interpretation provided by a state, as stated in the first case. However, the second case considers the possibility of γ, a literal, holding in the associated logic program Π. The syntactic constraints on γ in the second case are required since we do not want any special-purpose inference mechanism to cope with the proofs using the logic program. Rather, we shall assume the basic backward-chaining SLDNF resolution procedure [1, 7] whenever we want to run portions of Π.

We now formally define the execution mechanism for an agent specification.

Definition 3.2. Relation $\rho^{\Delta}_{\Sigma}(S_i, S_{i+1}), \Sigma = \langle \Gamma, \Pi \rangle$, holds if for all $\gamma \in \Gamma$ of the form $\phi \Rightarrow \bigcirc\varphi$ such that $\mathbf{K}(S_i,\Pi,\phi) = \mathbf{true}$ then $\mathbf{K}^{\Delta}(S_{i+1},\varphi) = \mathbf{true}$.

That is, in order to obtain state S_{i+1} given S_i and $\Sigma = \langle \Gamma, \Pi \rangle$, we have to examine all those temporal logic assertions $\gamma \in \Gamma$ whose left-hand side subformulae ϕ hold (either in S or Π) and use their right-hand side to build S_{i+1} via $\mathbf{K}^{\Delta}$, defined below.

Δ is a set of *temporal constraints*: these are basic temporal formulae to be satisfied at future states $S_{i+2}, S_{i+3}, \ldots$. Δ is necessary because there might be subformulae in φ that refer to other future states in the execution, and thus cannot be satisfied simply in S_{i+1}. For instance, $\bigcirc\bigcirc\varphi$ holds in S_i if $\bigcirc\varphi$ holds in S_{i+1} and φ holds in S_{i+2} – in order to carry any temporal constraints "down the line" we have employed the set Δ.

We now recursively define the auxiliary relation $\mathbf{K}^{\Delta}$ as a series of possible cases, depending on the format of φ:

Definition 3.3. $\mathbf{K}^{\Delta}(S,\varphi)$ for a given basic temporal formula $\varphi \in \mathcal{R}$ is defined as:

i) $\mathbf{K}^{\emptyset}(\{p(t_1,\ldots,t_n)\}, p(t_1,\ldots,t_n)) = \mathbf{true}$, that is, if φ is a positive literal, $s = \{\varphi\}$ is the set that will make $\mathbf{K}(S,\varphi) = \mathbf{true}$. In this case, Δ the set of temporal constraints for future states is the empty set.

ii) $\mathbf{K}^{\Delta_1 \cup \Delta_2}(S_1 \cup S_2, \varphi_1 \wedge \varphi_2) = \mathbf{true}$ if $\mathbf{K}^{\Delta_1}(S_1,\varphi_1) = \mathbf{K}^{\Delta_2}(S_2,\varphi_2) = \mathbf{true}$, that is, if φ is a conjunction of the form $\varphi_1 \wedge \varphi_2$, then its associated state $S = S_1 \cup S_2$ is the union of the associated states of each conjunct. Similarly, $\Delta = \Delta_1 \cup \Delta_2$ the set of temporal constraints on future states is the union of the sets of temporal constraints of each conjunct.

iii) $\mathbf{K}^{\Delta}(S, \varphi_1 \vee \varphi_2) = \mathbf{true}$ if $\mathbf{K}^{\Delta}(S,\varphi_1) = \mathbf{true}$ or $\mathbf{K}^{\Delta}(S,\varphi_2) = \mathbf{true}$ or $\mathbf{K}^{\Delta}(S,\varphi_1 \wedge \varphi_2) = \mathbf{true}$, that is, if φ is a disjunction, then its associated state S and set of temporal constraints Δ are obtained by analysing each disjunct individually or by analysing their conjunction.

iv) $\mathbf{K}^{\{\varphi\}}(\emptyset, \bigcirc\varphi) = \mathbf{true}$, that is, we associate the empty state with $\bigcirc\varphi$ and $\Delta = \{\varphi\}$ is the set of temporal constraints for the next state(s).

v) $\mathbf{K}^{\Delta}(S, \Diamond\varphi) = \mathbf{true}$ if $\mathbf{K}^{\Delta}(S,\varphi) = \mathbf{true}$ or $s = \emptyset, \Delta = \{\Diamond\varphi\}$, that is, the state associated with $\Diamond\varphi$ is either one in which φ holds (with the appropriate Δ) or an empty state with $\Diamond\varphi$ as a temporal constraint for the next state(s).

vi) $\mathbf{K}^{\{\Box\varphi\}}(S, \Box\varphi) = \mathbf{true}$ if $\mathbf{K}^{\Delta}(S,\varphi) = \mathbf{true}$, that is, the state S associated with $\Box\varphi$ is one in which $\mathbf{K}^{\Delta}(S,\varphi)$ holds and the temporal constraint is $\{\Box\varphi\}$.

The definition for $\mathbf{K}^{\Delta}$ obtains one state S in which a basic temporal formula φ holds true and the set Δ of temporal constraints which ought to hold in future state(s). It obtains S by decomposing φ into its basic subformulae and recursively analysing them. We can naturally extend $\mathbf{K}^{\Delta}$ for sets of basic temporal formulae: $\mathbf{K}^{\Delta}(S, \{\varphi_1, \ldots, \varphi_n\}) = \mathbf{true}$ if $\mathbf{K}^{\Delta}(S, \varphi_1 \wedge \cdots \wedge \varphi_n\}) = \mathbf{true}$.

The definitions for ρ^{Δ}_{Σ} and $\mathbf{K}^{\Delta}$ only contemplate one single transition, *i.e.* one step in our execution. We now define a complete execution:

Definition 3.4. An execution of a temporal specification $\Sigma = \langle \Gamma, \Pi \rangle$ is an infinite sequence of states $\sigma_{\Sigma} = S_0 S_1 S_2 \cdots$ such that $S_0 = \Delta_0 = \emptyset$ and for all $i > 0$,

a) $\rho^{\Delta'_i}_{\Sigma}(S_{i-1}, S'_i)$ and

b) $\mathbf{K}^{\Delta''_i}(S''_i, \Delta_{i-1})$ and

c) $S_i = S'_i \cup S''_i$ and $\Delta_i = \Delta'_i \cup \Delta''_i$.

That is, given the current state S_{i-1} and its set of temporal constraints Δ_{i-1} we can obtain a next state S_i and its set of temporal constraints Δ_i. The first condition builds a temporary state S'_i using only Σ and S_{i-1}; a set Δ'_i with any temporal constraints found when solving $\rho^{\Delta'_i}_{\Sigma}(S_{i-1}, S'_i)$ is also assembled at this point. The second formula employs the temporal constraints Δ_{i-1} of the previous state S_{i-1} to build S''_i; any temporal constraints arising from this analysis are included in Δ''_i. Finally, the resulting state S_i and set Δ of temporal constraints are the unions, respectively, $S'_i \cup S''_i$ and $\Delta'_i \cup \Delta''_i$.

3.1 Example

Let us now consider a simple example to illustrate how the definitions above can be put to work. Let there be the following simple agent specification:

$$\Sigma = \left\langle \begin{array}{|r|}\hline 1.\ \boxed{\top} \Rightarrow \bigcirc \boxed{p} \\ 2.\ \boxed{p} \Rightarrow \bigcirc \boxed{\Box q} \\ 3.\ \boxed{p \wedge q \wedge r} \Rightarrow \bigcirc \boxed{s} \\ \hline \end{array}, \begin{array}{|l|}\hline r \leftarrow \ldots \\ \\ \\ \hline \end{array} \right\rangle$$

We have numbered the temporal assertions in Γ for easy reference. Given the above specification we then have:

1. Initially, $S_0 = \Delta_0 = \emptyset$ are given.
2. $S_1 = \{p\}$, that is, $\rho_{\Sigma}^{\Delta_0}(S_0, \{p\})$ since temporal assertion 1 is the only one whose left-hand side is true in the empty state S_0. As the right-hand side has no temporal operators besides the compulsory $\bigcirc$, then $\Delta_1 = \emptyset$.
3. $S_2 = \{p, q\}$, that is, $\rho_{\Sigma}^{\Delta_1}(S_1, \{p, q\})$, since the left-hand side of assertions 1 and 2 hold in S_1 which causes their right-hand side to "happen" in S_2. Assertion 2 is responsible for the addition of a temporal constraint: $\Delta_2 = \{\Box q\}$, because $\mathbf{K}^{\{\Box q\}}(S_2, \Box q) = \mathbf{K}^{\emptyset}(S_2, q) = \mathit{true}$.
4. $S_3 = S_4 = \ldots = \{p, q, s\}$, that is, $\rho_{\Sigma}^{\Delta_i}(S_i, \{p, q, s\}), i \geq 2$, a fixpoint is reached in which no more propositions are inferred. Assertions 1 and 2 are proved since its left-hand side propositions $\{p, q\}$ are in S_i, hence we include p and q in S_{i+1}. Proposition q would also be included due to the temporal constraint in $\Delta_i, \mathbf{K}^{\{\Box q\}}(\{q\}, \{\Box q\})$. Assertion 3 is triggered as well, since p and q are in S_i and r is proved in Π.

Although the execution of the above agent specification is infinite, we may stop it by defining terminating conditions. For instance, we may want the execution to stop after the three propositions $\{p, q, s\}$ have been proved.

4 Design Rationale for TeLA

We have stripped MetateM down to its set of temporal operators which refer to the future. This was due to the fact that the temporal model checkers (*e.g.*, SPIN [5], SMV and STeP [3]) we want to use for verifying our MASs only allow future operators. Since we do not have temporal past operators, we had to "shift" the point of reference of our formulae: rather than having a $\mathit{PastConditions} \Rightarrow \mathit{ActionsInPresent\&Future}$ pattern in our formulae as in MetateM, we have instead $\mathit{PresentConditions} \Rightarrow \mathit{ActionsInPresent\&Future}$.

Our semantics is slightly different from those of usual temporal logics. The syntactic restrictions of temporal assertions ensure that the right-hand side of the temporal assertions comprises only conjuncts or disjuncts of positive literals, possibly with temporal operators. These literals are not proved. Rather, they are considered true in the next state (if they do not have temporal operators) or will become temporal constraints for the future state(s). The left-hand side of a temporal assertion has to be proved true in order to have the right-hand side literals considered true. The other possibilities for the proof of a temporal assertion, *viz. i)* left side false and right side true; *ii)* both sides false shall not be considered when executing a TeLA specification.

We adopt a left-to-right proof mechanism *à la* Prolog [1, 7] to examine the sequence of literals on the left-hand side of our temporal assertions. When we are using our TeLA formulae to guide the execution of the system we are solving an instance of the satisfiability problem, that is, that of finding a model for a logical theory. A sequence of non-temporal left-hand side literals will only have its ith literal proved if its $(i-1)$th literal is proved first. Although constructs of the form $A \Rightarrow (B \wedge \bigcirc C)$, A, B non-temporal literals, are ruled out by our syntactic restrictions, formula $A \wedge B \Rightarrow \bigcirc C$ can be employed instead – the left-to-right proof mechanism plus the restricted semantics of the $\Rightarrow$ operator renders both constructs equivalent with respect to the resulting model.

Disjunctions on the right-hand side of temporal assertions have a non-deterministic effect when we are building models. If we are to build a state in which, for instance, $p \vee q$ is to be true, we have three possibilities, $\{p\}, \{q\}$ and $\{q, p\}$. Similarly, if $\Diamond p$ is to be true in the next state, we have three possibilities, p being true in the next state, p being true in the next state and $\Diamond p$ being a temporal constraint for the future states, and p being false in the next state and $\Diamond p$ being a temporal constraint for future states.

5 Features of Agents

Rather than having our users (re-)defining the individual agents in great detail and specificity, we provide a generic list of *features* each agent should have and means of "tuning" them. These features have been chosen on the grounds that they play an important role in the dynamics of the target class of MAS, having a direct impact/effect in the overall collective behaviour of the populations.

Each feature has a selection of options the user chooses from in order to customise it for a particular experiment. All features must be specified by the user otherwise a default setting is employed. Associated with each feature we have a (possibly empty) set of temporal assertions, however the users of our framework need not be aware of these since they will not be manipulated directly – ideally, a suitable interface should present features and allow their customisation. Some useful features of a generic species of resource-aware agents are described below.

- **Resources** – This feature captures the case when an agent has an initial amount of resources. Depending on the actual system being modelled, resources can be money, air tickets, energy, etc.:

  $$\boxed{\mathit{start}} \Rightarrow \bigcirc \boxed{\mathit{resources}(R)}$$

  as the axiom representing that an agent has, at the start of the computation, R units of resources as a feature.

- **Metabolism** – this feature describes the rate at which resources change during the agent's life. The metabolism feature can be *positive* (that is, resources increase with time) or *negative* (that is, resources are depleted with time). A positive/negative metabolism can be *low*, *medium* or *high*. Such a feature could be represented, for instance, by the formula

$$\boxed{resources(R) \wedge acc\_dep(P) \wedge rate(T) \wedge NewR \; \texttt{is} \; R + (P * R * T/100)} \Rightarrow \bigcirc \boxed{resources(NewR)}$$

  That is, if R is the current value of the resources and if $P \in \{-1, +1\}$ has been specified as the accrete ($+1$) or deplete (-1) form of metabolism and $T \in \{1, 5, 10\}$ is the percentage/rate of accretion/depletion (1 = low; 5 = medium; 10 = high), then in the next state of the computation we shall update *resources* as *NewR*, where *NewR* is the result of the expression $R + (P * R * T/100)$.

- **Death Conditions** – In some cases the agent should stop working (*i.e.* it should "die") when certain circumstances arise:

$$\boxed{\begin{array}{l} resources(R) \wedge lower(LowerLimit) \wedge upper(UpperLimit) \wedge \\ (R < LowerLimit \vee R > UpperLimit) \end{array}} \Rightarrow \bigcirc \boxed{dead}$$

  That is, if the current value R of resources is lower than *LowerLimit* or higher than *UpperLimit*, then the agent is considered dead. If either the upper or lower limit does not apply, then we can specify them as, respectively $upper(+\infty)$ and $lower(-\infty)$.

  When more complex conditions need to be specified we can resort to a predicate defined in the associated logic program Π, thus "hiding" the actual computations/situations. For instance, an agent may die when a particular message is received, when a final goal is reached or when any combination of conditions are reached. The following formula captures this feature:

$$\boxed{\texttt{deathConditions}} \Rightarrow \bigcirc \boxed{dead}$$

  Predicate `deathConditions` should be defined by the agent designer in Π its default definition being $\perp$, *i.e.*, false.

- **Need/Drive** – Each agent must have a *purpose* for its existence in the system. We can capture this aspect with the need/drive feature: this refers to what the agent wishes for its resources. Some possible options for this feature are:

  – *maximise*, *i.e.* the agent wants to keep its resources above a certain threshold:

$$\boxed{resources(R) \wedge drive(maximise)} \Rightarrow \bigcirc \boxed{getResources}$$

  – *stabilise*, *i.e.* the agent wants to keep its resources within certain boundaries:

$$\boxed{\begin{array}{r} resources(R) \wedge drive(stabilise) \wedge \\ threshold([Upper, Lower]) \wedge (R < Lower \vee R > Upper) \end{array}} \Rightarrow \bigcirc \boxed{getResources}$$

  – *minimise*, *i.e.* the agent wants to keep its resources below a certain threshold:

$$\boxed{\begin{array}{r} resources(R) \wedge drive(minimise) \wedge \\ threshold([Upper, Lower]) \wedge R \geq Lower \end{array}} \Rightarrow \bigcirc \boxed{spendResources}$$

  The predicates *getResources* and *spendResources* will be considered true in the next state of the computation and they can be regarded as "flags" which will be ensure that in the next state appropriate temporal assertions are proved true.

- **Strategy** – This feature specifies the adopted behaviour of the agent when it is trying to fulfill its needs (or follow its drive). The strategy of an agent can either be *careful* or *greedy*. A careful agent tries to fulfill its needs with a low frequency:

$$\boxed{getResources \wedge careful} \Rightarrow \bigcirc \boxed{\bigcirc \bigcirc \, requestResource}$$

  That is, if a careful agent is to get resources, then it waits for the third next state of the computation to request for resources. A greedy agent, contrastingly, request resources at every next state, for three consecutive times:

$$\boxed{getResources \wedge greedy} \Rightarrow \bigcirc \boxed{requestResource \wedge \bigcirc requestResource \wedge \bigcirc \bigcirc \, requestResource}$$

  This feature allows for more sophisticated definitions in which the frequency of the agent's attempts to fulfill its needs is dynamically adjusted: within boundaries of the careful/greedy choices, the agent can get more frequent if its death is imminent or if it has a high metabolism.

The specification of an agent should include all the particular choices the user has made. A special formula carries the details of the agent's design to every state ρ-accessible from the initial state S_0. For instance,

$$\boxed{start} \quad \Rightarrow \quad \bigcirc \boxed{\Box \left(\begin{array}{l} acc\_dep(+1) \wedge rate(10) \wedge lower(0) \wedge upper(+\infty) \wedge \\ drive(stabilise) \wedge threshold(+\infty, 1000) \wedge careful \end{array} \right)}$$

represents the design choices for our previous axioms. Since the left-hand side is true in the initial *start* state, the right-hand side literals will be included in all future states as they all become temporal constraints.

Other formulae are necessary in the specification of an agent. These formulae represent "common sense" temporal relations and are to be included in any specification. A special state, the initial state, can be characterised by the following formulae:

$$\begin{array}{rcl} \boxed{\neg startFlag} & \Rightarrow & \bigcirc \boxed{start \wedge startFlag} \\ \boxed{startFlag} & \Rightarrow & \bigcirc \boxed{startFlag} \end{array}$$

The combined effect of the formulae above causes *start* to be true at the second state of the execution and never to be true again in the execution (assuming no other formulae infer, directly or not, *start*). The formula below deals with the death of an agent:

$$\boxed{dead} \Rightarrow \bigcirc \boxed{dead}$$

That is, once *dead*, then the agent will henceforth remain *dead*.

6 Designing Agents using TeLA

TeLA allows for a large class of computations to be specified and executed. TeLA can be used as a programming language *per se*, with a straightforward way to describe usual programming practices, such as establishing initial values, specifying how values should change at each state in the execution and terminating conditions. Temporary values can be passed on to subsequent stages of the execution and employed when needed. A simple template for a generic program in TeLA is, for instance,

$$\Gamma = \boxed{\begin{array}{rcl} \boxed{start} & \Rightarrow & \bigcirc \boxed{data(InitialValues)} \\ \boxed{data(CurrentValues) \wedge \texttt{compute}(CurrentValues, NewValues)} & \Rightarrow & \bigcirc \boxed{data(NewValues)} \\ \boxed{data(CurrentValues) \wedge \texttt{terminationCondition}(CurrentValues)} & \Rightarrow & \bigcirc \boxed{dead} \end{array}}$$

`compute`/2 and `terminationCondition`/1 are arbitrarily complex predicates defined in the associated program Π. An instance of the template above is the following program to sum the elements of a list of numbers:

$$\Gamma = \boxed{\begin{array}{rcl} \boxed{start} & \Rightarrow & \bigcirc \boxed{sum(InitialList, 0)} \\ \boxed{sum([X|Xs], Ac) \wedge NewAc \texttt{ is } Ac + X} & \Rightarrow & \bigcirc \boxed{sum(Xs, NewAc)} \\ \boxed{sum([], Ac)} & \Rightarrow & \bigcirc \boxed{sum(Ac) \wedge dead} \end{array}}$$

Any tail-recursive logic program can be reimplemented using TeLA with relative ease. Non tail-recursive programs may also be reimplemented, however their TeLA versions may be more complicated to obtain and less intuitive to follow. Since we can program a Turing Machine in TeLA, its computational power is, in principle, sufficient to replicate any program.

We suggest at this point a general approach to design agents using TeLA. We make a distinction between two concepts, a (description of a) *behaviour* and an *attitude*. A behaviour is a set of temporal assertions which depict, by means of the literals on their left-hand side, actions to be performed as these literals are proved. An attitude is a literal used to flag a behaviour and that gets changed upon certain events. An attitude can be seen as a *guard* of a temporal assertion, making sure that it is only employed when conditions are appropriate. To avoid unnecessary costly proofs of literals, possibly with unwanted (and undoable) side-effects, attitudes should be the very first literal in the left-hand side of a temporal assertion. The following template illustrates the roles of attitudes and behaviours in a TeLA specification:

$$\Gamma = \boxed{\begin{array}{rcl} \boxed{start} & \Rightarrow & \bigcirc \boxed{attitude(a_1)} \\ \boxed{attitude(a_1) \wedge Behaviour_1} & \Rightarrow & \bigcirc \boxed{attitude(a_2)} \\ \boxed{attitude(a_2) \wedge Behaviour_2} & \Rightarrow & \bigcirc \boxed{attitude(a_3)} \\ \boxed{attitude(a_3) \wedge Behaviour_3} & \Rightarrow & \bigcirc \boxed{attitude(a_1)} \end{array}}$$

where $Behaviour_i$ are conjuncts/disjuncts of non-temporal literals. Each attitude a_i defines a state of a finite state machine, graphically depicted by a graph:

$$\begin{tikzcd} start \arrow[r] & a_1 \arrow[r] & a_2 \arrow[r] & a_3 \arrow[ll, bend right] \end{tikzcd}$$

We have specified, using the attitude/behaviour distinction, two species of agents, an *auctioneer* agent species and a *bidder* agent species. The former advertises items to be read by the latter who then places a bid for the item. The auctioneer agent receives a number of bids and then chooses among them the one that is best according to criteria. First we show the auctioneer agent:

$$\Gamma = \left[\begin{array}{rcl}
\boxed{start} & \Rightarrow & \bigcirc\ \boxed{\begin{array}{l} proceedings(0) \wedge items(ItemList) \wedge \\ advertiseAtt \end{array}} \\
\boxed{items([])} & \Rightarrow & \bigcirc\ \boxed{dead} \\
\boxed{\begin{array}{r} advertiseAtt \wedge items(ItemList) \wedge \\ \texttt{advertise}(ItemList, AdItem) \end{array}} & \Rightarrow & \bigcirc\ \boxed{receiveBidsAtt(AdItem)} \\
\boxed{receiveBidsAtt(AdItem) \wedge \texttt{inBids}(AdItem, Bids)} & \Rightarrow & \bigcirc\ \boxed{chooseBidsAtt(AdItem, Bids)} \\
\boxed{\begin{array}{r} chooseBidsAtt(AdItem, Bids) \wedge \\ \texttt{chooseBids}(AdItem, Bids, ChosenBid, RestBids) \end{array}} & \Rightarrow & \bigcirc\ \boxed{informBiddersAtt(ChosenBid, RestBids)} \\
\boxed{\begin{array}{r} informBiddersAtt(ChosenBid, RestBids) \wedge \\ \texttt{informBidders}(ChosenBid, RestBids) \end{array}} & \Rightarrow & \bigcirc\ \boxed{updateAtt(ChosenBid)} \\
\boxed{\begin{array}{r} updateAtt(ChosenBid) \wedge items(Items) \wedge proceedings(Procs) \wedge \\ \texttt{update}(ChosenBid, Items, Procs, NewItems, NewProcs) \end{array}} & \Rightarrow & \bigcirc\ \boxed{advertiseAtt} \\
\boxed{\neg updateAtt(_) \wedge items(Items) \wedge proceedings(Procs)} & \Rightarrow & \bigcirc\ \boxed{items(Items) \wedge proceedings(Procs)}
\end{array}\right]$$

The bidders have the following Γ:

$$\Gamma = \left[\begin{array}{rcl}
\boxed{start} & \Rightarrow & \bigcirc\ \boxed{\begin{array}{l} funds(InitialFunds) \wedge items([]) \wedge \\ readAdsAtt \end{array}} \\
\boxed{funds(0)} & \Rightarrow & \bigcirc\ \boxed{dead} \\
\boxed{readAdsAtt \wedge \texttt{inAds}(ListAds)} & \Rightarrow & \bigcirc\ \boxed{chooseAd(ListAds)} \\
\boxed{\begin{array}{r} chooseAd(ListAds) \wedge funds(Funds) \wedge \\ \texttt{choose}(ListAds, Funds, ChosenAd) \end{array}} & \Rightarrow & \bigcirc\ \boxed{placeBidAtt(ChosenAd)} \\
\boxed{placeBidAtt(ChosenAd) \wedge \texttt{placeBid}(ChosenAd)} & \Rightarrow & \bigcirc\ \boxed{getConfirmationAtt(ChosenAd)} \\
\boxed{\begin{array}{r} getConfirmationAtt(ChosenAd) \wedge \\ \texttt{getConfirmation}(ChosenAd, Confirmation) \end{array}} & \Rightarrow & \bigcirc\ \boxed{updateAtt(ChosenAd, Confirmation)} \\
\boxed{\begin{array}{r} updateAtt(ChosenAd, Confirmation) \wedge funds(Funds) \wedge items(Items) \wedge \\ \texttt{update}(ChosenAd, Items, Funds, NewItems, NewFunds) \end{array}} & \Rightarrow & \bigcirc\ \boxed{readAdsAtt} \\
\boxed{\neg updateAtt(_) \wedge items(Items) \wedge funds(Funds)} & \Rightarrow & \bigcirc\ \boxed{items(Items) \wedge funds(Funds)}
\end{array}\right]$$

The definitions of the associated logic programs Π are, of course, essential for the actual execution of the MAS. However, the predicates defined in Π, that is, all those literals in `teletype font`, do not possess any temporal properties of interest: they may be considered as literals that are proven true (or not) in an instantaneous fashion. Any important results obtained in the proof of literals in Π must be carried around the states of the specification execution. Clearly, the designer must decide on which aspects of the agent have relevant temporal properties and are to be represented via temporal assertions or can be safely encapsulated as a predicate in Π.

7 Verification of TeLA Specifications

MASs are sets of TeLA specifications $\{\Sigma_1, \ldots, \Sigma_n\}$ which interact with each other. The actual execution of each agent specification Σ_i is independent from all other agents, in a distributed fashion. Designing reliable distributed systems is difficult and a long-standing problem in software engineering has been the supply of reliable tools and methods to check the correctness of such systems [6].

In order to ensure that desirable properties hold, specifications of MASs should be automatically verifiable [4]. The design of TeLA was influenced by this drive: we want our specifications to be verified using existing model checkers like SPIN [5], SMV and theorem provers like STeP [3]. Given an agent specification $\Sigma = \langle \Gamma, \Pi \rangle$ (or a set of interacting agents specifications) we want to check if a set of temporal properties Ξ hold. The restricted syntax of TeLA specifications guarantees that its formulae are acceptable by any linear temporal logic theorem prover or model checker. The syntax of Ξ, the temporal properties to be checked, should conform to the theorem prover or model checker to be used.

The associated logic program Π is an essential part of a specification. It provides a means for encapsulation of non-temporal parts of the specification and plays an important role in the execution and, consequently, verification of agents specifications. However, since we want to use existing techniques and tools to verify our specifications we must somehow eliminate the need for a special proof mechanism when dealing with literals in Π. To solve this, we propose that temporal properties for the literals in Π be supplied by the designer who is verifying the specification. For instance, the literal $\texttt{inAds}(ListAds)$ of our previous specification for a bidder agent, to be proved in Π, could be provided with the temporal property $\texttt{advertise}(ItemList, AdItem) \Rightarrow \Diamond \texttt{inAds}([AdItem])$ stating that items advertised by the auctioneer are eventually noticed by the bidders. The temporal properties of Π's literals are denoted by Γ_Π and should conform to the syntax of the model checker or theorem prover used.

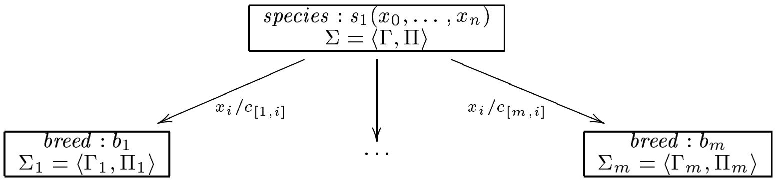
The verification of a MAS specification can be seen as an attempt to prove that Ξ holds, if Γ_i and Γ_{Π_i} hold. More formally, we want to know if the formula $[(\Gamma_1 \wedge \cdots \wedge \Gamma_n) \wedge (\Gamma_{\Pi_1} \wedge \cdots \wedge \Gamma_{\Pi_n})] \Rightarrow \Xi$ holds, where a set of formulae is understood as the conjunction of its elements.

8 Populations of Agents

Our MASs will be modelled via the specification of populations of species of agents. An agent species is an open specification $\Sigma = \langle \Gamma, \Pi \rangle$ in which there are slots/parameters (either in Γ or Π) that should be provided in order to complete it. Γ contains the temporal features of the species, and Π contains its non-temporal/operational details. A species will be referred to by the construct:

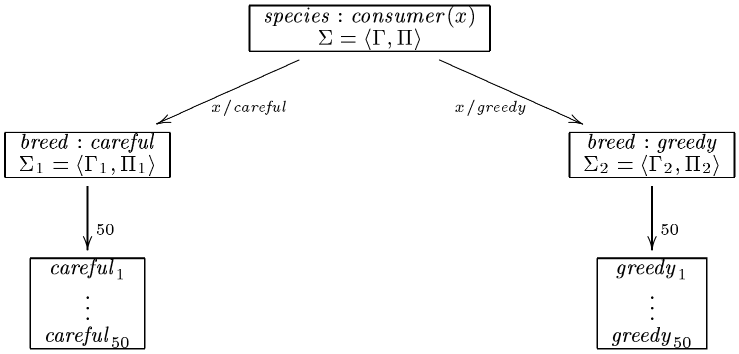
$$\left[\begin{array}{c} species : SpeciesName(x_0, \ldots, x_n) \\ \Sigma = \langle \Gamma, \Pi \rangle \end{array}\right]$$

The x_i comprise a list with all slots/parameters that must be provided. Given a species, a number of breeds can be defined by customising its list of parameters $x_0, \ldots, x_n$:



The values for the slots/parameters may be provided by the designer of the experiment or randomly generated from a list of possible values.

A population is a set of individuals from a breed. The exact number of individuals must be provided by the designer of the experiment. Alternatively, a random number will be chosen by the system. For instance, an initial species *consumer* can be specified and then further specialised into two breeds, *careful* and *greedy*. The populations of each agent are defined via an integer informing the number of individuals:



Although each feature in isolation does not offer much in terms of diversity of behaviours, their combination and interplay allow for interesting phenomena to happen. Furthermore, the definition of different species and breeds of agents and how they relate to each other offer another dimension for enriching the experiments.

During the execution/simulation of the system, the population specification causes 50 instances of careful and 50 instances of greedy *consumer* agents to be started up simultaneously, each of them independently exhibiting the features established in the species definition, customised to the breed.

Our information ecosystems, as referred to in the SLIE grant proposal, can be now conveniently defined in terms of such populations of breeds of agents. Each species of agents has well-established features that can be customised to particular breeds. Once an agent species is available, it can be (re-)used in different experiments changing the specific details of breeds, number of individuals and relationship among populations.

8.1 Relationships among Populations: Topologies for Communication

After the populations have been defined in terms of numbers of agents of a particular breed, we ought to offer a way to explicitly relate distinct populations. A simple way is by means of another feature specifying which other breeds of agents an individual agent is allowed to communicate with.

Given that we aim at systems with a large number of agents, it is not feasible to require that the relationships among populations be specified on an individual by individual basis, that is, informing that the individual agent a_i of population p is related with individual agent b_j of population q. Clearly, this approach does not scale up and may lead to many mistakes.

Populations can be related in a general fashion, that is, we can simply inform that individuals of breed b_i should only receive messages from individuals of breed b_j. The relationship can be made more precise by establishing the maximum number of agents of breed b_i that can be connected at any moment with an agent of breed b_j. If a population p is related to more than one distinct population $q_1, \ldots, q_n$, then a percentage could be supplied informing how likely it is that an individual in p is connected with q_1, etc.

9 Conclusions and Directions for Future Work

We have proposed a framework for the specification, verification and execution of populations of agents. An agent is specified via the pair $\Sigma = \langle \Gamma, \Pi \rangle$ where Γ is a set of temporal assertions and Π is an associated logic program. Intuitively, Γ represents a set of temporal restrictions that should be obeyed as Π is executed.

Γ is a set of formulae of TeLA, our special-purpose temporal logic of assertions which describes time-related aspects of an agent. Γ is used as a set of axioms when we want to verify temporal properties via model-checkers. Γ can also be executed, thus defining the dynamics of the computations in an agent.

In order to define populations of agents, we initially need to specify a *species*, which is an open agent specification, then customise the species as distinct *breeds* and finally determine the number of individuals of a breed. The set of individuals of a particular breed is a population. Our MASs may consist of a number of different populations of agents.

The division of an agent specification Σ into Γ and Π can be explained in a number of ways. First, by keeping all temporal considerations restricted to Γ and having Π free from temporal operators, we can exploit efficient, conventional

SLDNF-resolution within Π. Γ is used to guide the execution of the agent: temporal operators are dealt with in a straightforward fashion, when we are trying to build the sequence of possible worlds in which an execution is possible. Second, the set of formulae Γ provides us with a description of the properties to be satisfied by models (executions) of Π. However, since we are trying to execute the specification of the agent, Γ actually tells us how the execution should proceed. Since Γ formalises features of the agent, if we guarantee that Γ holds throughout an execution, then we can say that the features constrain the execution of Π.

The major contributions of this work lies in the proposal of a special-purpose temporal language that is amenable to verification via existing model-checkers and is also executable. We have also suggested a means to combine our temporal language with conventional logic programming, conveniently enabling the specification of actual programs, an essential feature if we want our agents to possess useful behaviours. It is important to notice that specifications in TeLA are declarative which renders them executable in alternative fashions: distributed executions and simulations are equally possible with the same specification. For verification purposes, our specifications can be simply regarded as temporal logic formulae.

We have implemented a program to build the sequence of states of a specification $\Sigma = \langle \Gamma, \Pi \rangle$. This program actually executes the specification using Γ and resorting to Π to prove some literals. The sequence of states can be seen as a trace/run of the specification. A number of MASs have been specified and executed, including the banking scenario and the simple auction scenario whose agents were shown as examples above. We adopted a distributed platform for our execution with each agent an independent process communicating via message-passing but we could have opted to simulate the execution using a single process.

We want to explore the adequacy and ease-of-use of our approach to specify, verify and execute more sophisticated MASs, such as the tourism scenario. We believe that TeLA may experience slight adjustments to allow for the description of more generic kinds of computations. Independently, we shall develop tools to help people design populations of agents and experiment with them.

References

[1] K. R. Apt. *From Logic Programming to Prolog.* Prentice-Hall, U.K., 1997.

[2] H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. MetateM: an Imperative Approach to Temporal Logic Programming. *Formal Aspects of Computing*, 7(E):111–154, 1995.

[3] N. J. Bjørner, A. Browne, M. A. Colón, E. Finkbeiner, Z. Manna, H. B. Sipma, and T. E. Uribe. Verifying Temporal Properies of Reactive Systems: a STeP Tutorial. *Formal Methods in System Design*, 16(3):227–270, June 2000.

[4] M. Fisher and M. Wooldridge. On the Formal Specification and Verification of Multi-Agent Systems. *International Journal of Cooperative Information Systems*, 6(1):37–65, 1997.

[5] G. J. Holzmann. The SPIN Model Checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[6] G. J. Holzmann, E. Najm, and A. Serhrouchni. SPIN Model Checking: an Introduction. *Int. Journal of Software Tools for Technology Transfer*, 2(4):321–327, March 2000.

[7] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Second, Extended edition, 1993.

[8] Z. Manna. *Theoretical Foundations of Programming Methodologies*, chapter Verification of Sequential Programs: Temporal Axiomatization. NATO Scientific Series. D. Reidel Publ. Co., Holland, 1981.

[9] A. Pnuelli. The Temporal Logic of Programs. In *Proc. 18th FOCS-IEEE*, pages 46–57, 1977.

[10] N. Rescher and A. Urquhart. *Temporal Logic.* Springer-Verlag, Wien, 1971.