Chapter 11
A Temporal Trace Language for Formal Modelling and Analysis of Agent Systems

A. Sharpanskykh and J. Treur

Abstract This chapter presents the hybrid Temporal Trace Language (TTL) for formal specification and analysis of dynamic properties of multi-agent systems. This language supports specification of both qualitative and quantitative aspects, and subsumes languages based on differential equations and temporal logics. TTL has a high expressivity and normal forms that enable automated analysis. Software environments for performing verification of TTL specifications have been developed. TTL proved its value in a number of domains.

A. Sharpanskykh
Vrije Universiteit Amsterdam, Department of Artificial Intelligence, The Netherlands e-mail: sharp@cs.vu.nl

J. Treur
Vrije Universiteit Amsterdam, Department of Artificial Intelligence, The Netherlands e-mail: treur@cs.vu.nl

M. Dastani et al. (eds.), *Specification and Verification of Multi-agent Systems*,
DOI 10.1007/978-1-4419-6984-2_11, © Springer Science+Business Media, LLC 2010

11.1 Introduction

Traditionally, the multi-agent paradigm has been used to improve efficiency of software computation. Languages used to specify such multi-agent systems often had limited expressive power (e.g., executable, close to (logic) programming languages), which nevertheless was sufficient to describe complex distributed algorithms. Recently many agent-based methods, techniques and methodologies have been developed to model and analyse phenomena in the real world (e.g., social, biological, and psychological structures and processes). By formally grounded multi-agent system modelling one can gain better understanding of complex real world processes, test existing theories from natural and human sciences, identify different types of problems in real systems.

Modelling dynamics of systems from the real world is not a trivial task. Currently, continuous modelling techniques based on differential and difference equations are often used in natural science to address this challenge, with limited success. In particular, for creating realistic continuous models for natural processes a great number of equations with a multitude of parameters are required. Such models are difficult to analyze, both mathematically and computationally. Further, continuous modelling approaches, such as the Dynamical Systems Theory [344], provide little help for specifying global requirements on a system being modelled and for defining high level system properties that often have a qualitative character (e.g., reasoning, coordination). Also, sometimes system components (e.g., switches, thresholds) have behaviour that is best modelled by discrete transitions. Thus, the continuous modelling techniques have limitations, which can compromise the feasibility of system modelling in different domains.

Logic-based methods have proved useful for formal qualitative modelling of processes at a high level of abstraction. For example, variants of modal temporal logic [27, 198] gained popularity in agent technology, and for modelling social phenomena. However, logic-based methods typically lack quantitative expressivity essential for modelling precise timing relations as needed in, e.g., biological and chemical processes.

Furthermore, many real world systems (e.g., a television set, a human organisation, a human brain) are hybrid in nature, i.e., are characterized by both qualitative and quantitative aspects. To represent and reason about structures and dynamics of such systems, the possibility of expressing both qualitative and quantitative aspects is required. Moreover, to tackle the issue of complexity and scalability the possibility of modelling of a system at different aggregation levels is in demand. In this case modelling languages should be able to express logical relationships between parts of a system.

To address the discussed modelling demands, the Temporal Trace Language (TTL) is proposed, which subsumes languages based on differential equations and temporal logics, and supports the specification of the system behaviour at different levels of abstraction.

Generally, the expressivity of modelling languages is limited by the possibility to perform effective and efficient analysis of models. Analysis techniques for complex systems include simulation based on system models, and verification of dynamic properties on model specifications and traces generated by simulation or obtained empirically.

For simulation it is essential to have limitations to the language. To this end, an executable language that allows specifying only direct temporal relations can be defined as a sublanguage of TTL; cf. [81]. This language allows representing the dynamics of a system by a (possible large) number of simple temporal (or causal) relations, involving both qualitative and quantitative aspects. Furthermore, using a dedicated tool, TTL formulae that describe the complex dynamics of a system specified in a certain format may be automatically translated into the executable form. Based on the operational semantics and the proof theory of the executable language, a dedicated tool has been developed that allows performing simulations of executable specifications.

To verify properties against specifications of models two types of analysis techniques are widely used: logical proof procedures and model checking [100]. By means of model checking entailment relations are justified by checking properties on the set of all theoretically possible traces generated by execution of a system model. To make such verification feasible, expressivity of both the language used for the model specification and the language used for expressing properties has to be sacrificed to a large extent. Therefore, model specification languages provided by most model checkers allow expressing only simple temporal relations in the form of transition rules with limited expressiveness (e.g., no quantifiers). For specifying a complex temporal relation a large quantity (including auxiliary) of interrelated transition rules is needed. In this chapter normal forms and a transformation procedure are introduced, which enable automatic translation of an expressive TTL specification into the executable format required for automated verification (e.g., by model checking). Furthermore, abstraction of executable specifications, as a way of generating properties of higher aggregation levels, is considered in this chapter. In particular, an approach that allows automatic generation of a behavioural specification of an agent from a cognitive process model is described.

In some situations it is required to check properties only on a limited set of traces obtained empirically or by simulation (in contrast to model checking which requires exhaustive inspection of all possible traces). Such type of analysis, which is computationally much cheaper than model checking, is described in this chapter.

The chapter is organised as follows. Section 11.2 describes the syntax of the TTL language. The semantics of the TTL language is described in Section 11.3. Multi-level modelling of multi-agent systems in TTL and a running example used throughout the chapter are described in Section 11.4. In Section 11.5 relations of TTL to other well-known formalisms are discussed. In Section 11.6 normal forms and transformation procedures for automatic translation of a TTL specification into the executable format are introduced. Furthermore, abstraction of executable specifications is considered in Section 11.6. Verification of specifications of multi-agent

systems in TTL is considered in Section 11.7. Finally, Section 11.8 concludes the chapter.

11.2 Syntax of TTL

The language TTL is a variant of an order-sorted predicate logic [299]. Whereas standard multi-sorted predicate logic is meant to represent static properties, TTL is an extension of such language with explicit facilities to represent dynamic properties of systems. To specify state properties for system components, ontologies are used which are specified by a number of sorts, sorted constants, variables, functions and predicates (i.e., a signature). State properties are specified based on such ontology using a standard multi-sorted first-order predicate language. For every system component A (e.g., agent, group of agents, environment) a number of ontologies can be distinguished used to specify state properties of different types. That is, the ontologies *IntOnt*(A), *InOnt*(A), *OutOnt*(A), and *ExtOnt*(A) are used to express respectively internal, input, output and external state properties of the component A. For example, a state property expressed as a predicate *pain* may belong to *IntOnt*(A), whereas the atom *has_temperature*(*environment*, 7) may belong to *ExtOnt*(A). Often in agent-based modelling input ontologies contain elements for describing perceptions of an agent from the external world (e.g, *observed*(a) means that a component has an observation of state property a), whereas output ontologies describe actions and communications of agents (e.g., *performing_action*(b) represents action b performed by a component in its environment).

To express dynamic properties, TTL includes special sorts: $TIME$ (a set of linearly ordered time points), $STATE$ (a set of all state names of a system), $TRACE$ (a set of all trace names; a trace or a trajectory can be thought of as a timeline with a state for each time point), $STATPROP$ (a set of all state property names), and $VALUE$ (an ordered set of numbers). Furthermore, for every sort S from the state language the following TTL sorts exist: the sort S^{VARS}, which contains all variable names of sort S, the sort S^{GTERMS}, which contains names of all ground terms, constructed using sort S; sorts S^{GTERMS} and S^{VARS} are subsorts of sort S^{TERMS}.

In TTL, formulae of the state language are used as objects. To provide names of object language formulae φ in TTL, the operator (*) is used (written as φ^*), which maps variable sets, term sets and formula sets of the state language to the elements of sorts S^{GTERMS}, S^{TERMS}, S^{VARS} and $STATPROP$ in the following way:

1. Each constant symbol c from the state sort S is mapped to the constant name c' of sort S^{GTERMS}.
2. Each variable $x : S$ from the state language is mapped to the constant name $x' \in S^{VARS}$.
3. Each function symbol $f : S_1 \times S_2 \times ... \times S_n \rightarrow S_{n+1}$ from the state language is mapped to the function name $f' : S_1^{TERMS} \times S_2^{TERMS} \times ... \times S_n^{TERMS} \rightarrow S_{n+1}^{TERMS}$.

4. Each predicate symbol $P : S_1 \times S_2 \times ... \times S_n$ is mapped to the function name $P' : S_1^{TERMS} \times S_2^{TERMS} \times ... \times S_n^{TERMS} \rightarrow STATPROP$.
5. The mappings for state formulae are defined as follows:

   a. $(\neg\varphi)^* = not(\varphi^*)$
   b. $(\varphi \& \psi)^* = \varphi^* \wedge \psi^*, (\varphi | \psi)^* = \psi^* \vee \psi^*$
   c. $(\varphi \Rightarrow \psi)^* = \varphi^* \rightarrow \psi^*, (\varphi \Leftrightarrow \psi)^* = \varphi^* \leftrightarrow \psi^*$
   d. $(\forall x\, \varphi(x))^* = \forall x'\, \varphi^*(x')$, where x is variable over sort S and x' is any constant of S^{VARS}; the same for $\exists$.

It is assumed that the state language and the TTL define disjoint sets of expressions. Therefore, further in TTL formulae we shall use the same notations for the elements of the object language and for their names in the TTL without introducing any ambiguity. Moreover we shall use t with subscripts and superscripts for variables of the sort $TIME$; and γ with subscripts and superscripts for variables of the sort $TRACE$.

A state is described by a function symbol $state : TRACE \times TIME \rightarrow STATE$. A trace is a temporally ordered sequence of states. A time frame is assumed to be fixed, linearly ordered, for example, the natural or real numbers. Such an interpretation of a trace contrasts to Mazurkiewicz traces [306] that are frequently used for analysing behaviour of Petri nets. Mazurkiewicz traces represent restricted partial orders over algebraic structures with a trace equivalence relation. Furthermore, as opposed to some interpretations of traces in the area of software engineering, a formal logical language is used here to specify properties of traces.

The set of function symbols of TTL includes $\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$: $STATPROP \times STATPROP \rightarrow STATPROP$; $not : STATPROP \rightarrow STATPROP$, and $\forall, \exists : S^{VARS} \times STATPROP \rightarrow STATPROP$, of which the counterparts in the state language are boolean propositional connectives and quantifiers. Further we shall use $\vee, \wedge, \rightarrow, \leftrightarrow$ in infix notation and $\forall, \exists$ in prefix notation for better readability. For example, using such function symbols the state property about external world expressing that there is no rain and no clouds can be specified as: $not(rain) \wedge not(clouds)$.

To formalise relations between sorts $VALUE$ and $TIME$, functional symbols $-, +, /, \bullet : TIME \times VALUE \rightarrow TIME$ are introduced. Furthermore, for arithmetical operations on the sort $VALUE$ the corresponding arithmetical functions are included.

States are related to state properties via the satisfaction relation denoted by the prefix predicate *holds* (or by the infix predicate $\models$): $holds(state(\gamma, t), p)$ (or $state(\gamma, t) \models p$), which denotes that state property p holds in trace γ at time point t.

Both $state(\gamma, t)$ and p are terms of the TTL language. In general, TTL terms are constructed by induction in a standard way from variables, constants and function symbols typed with all before-mentioned TTL sorts. Transition relations between states are described by dynamic properties, which are expressed by TTL-formulae. The set of *atomic TTL-formulae* is defined as:

1. If v_1 is a term of sort $STATE$, and u_1 is a term of the sort $STATPROP$, then $holds(v_1, u_1)$ is an atomic TTL formula.
2. If τ_1, τ_2 are terms of any TTL sort, then $\tau_1 = \tau_2$ is a TTL-atom.
3. If t_1, t_2 are terms of sort $TIME$, then $t_1 < t_2$ is a TTL-atom.
4. If v_1, v_2 are terms of sort $VALUE$, then $v_1 < v_2$ is a TTL-atom.

The set of well-formed TTL-formulae is defined inductively in a standard way using Boolean connectives and quantifiers over variables of TTL sorts. An example of the TTL formula, which describes observational belief creation of an agent, is given below:

> In any trace, if at any point in time $t1$ the agent A observes that it is raining, then there exists a point in time $t2$ after $t1$ such that at $t2$ in the trace the agent A believes that it is raining.

$$\forall\gamma\ \forall t1\ [holds(state(\gamma, t1), observation_result(itsraining)) \Rightarrow$$
$$\exists t2 > t1\ \ holds(state(\gamma, t2), belief(itsraining))]$$

The possibility to specify arithmetical operations in TTL allows modelling of continuous systems, which behaviour is usually described by differential equations. Such systems can be expressed in TTL either using discrete or dense time frames. For the discrete case, methods of numerical analysis that approximate a continuous model by a discrete one are often used, e.g., Euler's and Runge-Kutta methods [334]. For example, by applying Euler's method for solving a differential equation $dy/dt = f(y)$ with the initial condition $y(t_0) = y_0$, a difference equation $y_{i+1} = y_i + h * f(y_i)$ (with i the step number and $h > 0$ the step size) is obtained. This equation can be modelled in TTL in the following way:

$$\forall\gamma\ \forall t\ \forall v : VALUE\ holds(state(\gamma, t),\ has_value(y, v)) \Rightarrow$$
$$holds(state(\gamma, t+1),\ has_value(y, v + h \bullet f(v)))$$

The traces γ satisfying the above dynamic property are the solutions of the difference equation.

Furthermore, a dense time frame can be used to express differential equations with derivatives specified using the epsilon-delta definition of a limit, which is expressible in TTL. To this end, the following relation is introduced, expressing that $x = dy/dt$:

$\mathbf{is_diff_of}(\gamma, \mathbf{x}, \mathbf{y})$:

$$\forall t, w\ \forall\epsilon > 0\ \exists\delta > 0\ \forall t', v, v'$$
$$0 < dist(t', t) < \delta\ \&\ holds(state(\gamma, t), has_value(x, w))$$

$\&holds(state(\gamma, t), has_value(y, v))$

$\&holds(state(\gamma, t'), has_value(y, v'))$

$\Rightarrow\ dist((v' - v)/(t' - t), w) < \epsilon$

where $dist(u, v)$ is defined as the absolute value of the difference.

Furthermore, a study has been performed in which a number of properties of continuous systems and theorems of calculus were formalized in TTL and used in reasoning [83].

11.3 Semantics of TTL

An *interpretation* of a TTL formula is based on the standard interpretation of an order sorted predicate logic formula and is defined by a mapping I that associates each:

1. sort symbol S to a certain set (subdomain) D_S, such that if $S \subseteq S'$ then $D_S \subseteq D'_S$
2. constant c of sort S to some element of D_S
3. function symbol f of type $< X_1, ..., X_i > \rightarrow X_{i+1}$ to a mapping: $I(X_1) \times ... \times I(X_i) \rightarrow I(X_{i+1})$
4. predicate symbol P of type $< X_1, ..., X_i >$ to a relation on $I(X_1) \times ... \times I(X_i)$

A *model* M for the TTL is a pair $M =< I, V >$, where I is an interpretation function, and V is a variable assignment, mapping each variable x of any sort S to an element of D_S. We write $V[x/v]$ for the assignment that maps variables y other than x to $V(y)$ and maps x to v. Analogously, we write $M[x/v] =< I, V[x/v] >$.

If $M =< I, V >$ is a model of the TTL, then *the interpretation of a TTL term* τ, denoted by τ^M, is inductively defined by:

1. $(x)^M = V(x)$, where x is a variable over one of the TTL sorts.
2. $(c)^M = I(c)$, where c is a constant of one of the TTL sorts.
3. $f(\tau_1, ..., \tau_k)^M = I(f)(\tau_1^M, ..., \tau_k^M)$, where f is a TTL function of type $S_1 \times ... \times S_n \rightarrow S$ and $\tau_1, ..., \tau_n$ are terms of TTL sorts $S_1, ..., S_n$.

The truth definition of TTL for the model $M =< I, V >$ is inductively defined by:

1. $\models_M P_i(\tau_1, ..., \tau_k)$ iff $I(Pi)(\tau_1^M, ..., \tau_k^M) = true$
2. $\models_M \neg\varphi$ iff $\not\models_M \varphi$
3. $\models_M \varphi \wedge \psi$ iff $\models_M \varphi$ and iff $\models_M \psi$
4. $\models_M \forall x(\varphi(x))$ iff $\models_{M[x/v]} \varphi(x)$ for all $v \in D_S$, where x is a variable of sort S.

The semantics of connectives and quantifiers is defined in the standard way. A number of important properties of TTL are formulated in form of axioms:

1. Equality of traces:
   $\forall \gamma_1, \gamma_2\ [\forall t[state(\gamma_1, t) = state(\gamma_2, t)] \Rightarrow \gamma_1 = \gamma_2]$
2. Equality of states:
   $\forall s_1, s_2 [\forall a : STATPROP[truth_value(s_1, a) = truth_value(s_2, a)] \Rightarrow s_1 = s_2]$
3. Truth value in a state:
   $holds(s, p) \Leftrightarrow truth_value(s, p) = true$
4. State consistency axiom:
   $\forall \gamma, t, p\ (holds(state(\gamma, t), p) \Rightarrow \neg holds(state(\gamma, t), not(p)))$
5. State property semantics:

   a. $holds(s, (p1 \wedge p2)) \Leftrightarrow holds(s, p_1)\ \&\ holds(s, p_2)$
   b. $holds(s, (p1 \vee p2)) \Leftrightarrow holds(s, p_1) \mid holds(s, p_2)$
   c. $holds(s, not(p_1)) \Leftrightarrow \neg holds(s, p_1)$

6. For any constant variable name x from the sort S^{VARS}:
   $holds(s, (\exists(x, F))) \Leftrightarrow \exists x' : S^{GTERMS}\ holds(s, G)$, and $holds(s, (\forall(x, F))) \Leftrightarrow \forall x' : S^{GTERMS}\ holds(s, G)$ with G, F terms of sort $STATPROP$, where G is obtained from F by substituting all occurrences of x by x'.
7. Partial order axioms for the sort $TIME$:

   a. $\forall t\ t \leq t$ (Reflexivity)
   b. $\forall t_1, t_2\ [t_1 \leq t_2 \wedge t_2 \leq t_1] \Rightarrow t_1 = t_2$ (Anti-Symmetry)
   c. $\forall t_1, t_2, t_3\ [t_1 \leq t_2 \wedge t_2 \leq t_3] \Rightarrow t_1 \leq t_3$ (Transitivity)

8. Axioms for the sort $VALUE$: the same as for the sort $TIME$ and standard arithmetic axioms.
9. Axioms, which relate the sorts $TIME$ and $VALUE$:

   a. $(t + v_1) + v_2 = t + (v_1 + v_2)$
   b. $(t \bullet v_1) \bullet v_2 = t \bullet (v_1 \bullet v_2)$

10. (Optional) Finite variability property (for any trace γ).
    This property ensures that a trace is divided into intervals such that the overall system state is stable within each interval, i.e., each state property changes its truth value at most a finite number of times:
    $\forall t_0, t_1\ t_0 < t_1 \Rightarrow \exists \delta > 0[\forall t[t_0 \leq t\ \&\ t \leq t_1] \Rightarrow \exists t_2[t_2 \leq t\ \&\ t < t_2 + \delta\ \&\ \forall t_3[t_2 \leq t_3\ \&\ t_3 \leq t_2 + \delta]] \Rightarrow state(\gamma, t_3) = state(\gamma, t)]$

11.4 Multi-level Modelling of Multi-Agent Systems in TTL

With increase of the number of elements within a multi-agent system, the complexity of the dynamics of the system grows considerably. To analyze the behaviour of a complex multi-agent system (e.g., for critical domains such as air traffic control and health care), appropriate approaches for handling the dynamics of the multi-agent system are important. Two of such approaches for TTL specifications of multi-agent systems are considered in this section: aggregation by agent clustering is considered in Section 11.4.1 and organisation structures are discussed in Section 11.4.2.

11.4.1 Aggregation by agent clustering

One of the approaches to manage complex dynamics is by distinguishing *different aggregation levels*, based on clustering of a multi-agent system into parts or components with further specification of their dynamics and relations between them; e.g., [264]. At the lowest aggregation level a component is an agent or an environmental object (e.g., a database), with which agents interact. Further, at higher aggregation levels a component has the form of either a group of agents or a multi-agent system as a whole. In the simplest case two levels can be distinguished: the lower level at which agents interact and the higher level, where the whole multi-agent system is considered as one component. In the general case the number of aggregation levels is not restricted. Components interact with each other and the environment via input and output interfaces described in terms of interaction (i.e., input and output) ontologies. A component receives information at its input interface in the form of observation results and communication from other components. A component generates at its output communication, observation requests and actions performed in the environment. Some elements from the agent's interaction ontology are provided in Table 11.1.

Table 11.1 Interaction ontology

Ontology element	Description
$observation_request_from_for(C : COMPONENT, I : INFO_ELEMENT)$	I is to be observed in the world for C (active observation)
$observation_result_to_for(C : COMPONENT, I : INFO_ELEMENT)$	Observation result I is provided to C (for active observation)
$observed(I : INFO_ELEMENT)$	I is observed at the component's input (passive observation)
$communicated_from_to(C1 : COMPONENT, C2 : COMPONENT, s_act : SPEECH_ACT, I : INFO_ELEMENT)$	Specifies speech act s_act (e.g., inform, request, ask) from $C1$ to $C2$ with the content I
$to_be_performed(A : ACTION)$	Action A is to be performed

For the explicit indication of an aspect of a state for a component, to which a state property is related, sorts $ASPECT_COMPONENT$ (a set of the component aspects of a system; i.e., input, output, internal); $COMPONENT$ (a set of all component names of a system); $COMPONENT_STATE_ASPECT$ (a set of all names of aspects of all component states) and a function symbol

$$comp_aspect : ASPECT_COMPONENT \times COMPONENT \rightarrow COMPONENT_STATE_ASPECT$$

are used. In multi-agent system specifications, in which the indication of the component's aspects is needed, the definition of the function symbol state introduced earlier is extended as $state : TRACE \times TIME \times COMPONENT_STATE_ASPECT \rightarrow STATE$. For example,

$$holds(state(trace1, t1, input(A)), observation_result(sunny_weather))$$

Here $input(A)$ belongs to sort $COMPONENT_STATE_ASPECT$.
At every aggregation level the behaviour of a component is described by a set of dynamic properties. The dynamic properties of components of a higher aggregation level may have the form of a few temporal expressions of high complexity. At a lower aggregation level a system is described in terms of more basic steps. This usually takes the form of a specification consisting of a large number of temporal expressions in a simpler format. Furthermore, the dynamic properties of a component of a higher aggregation level can be logically related by an interlevel relation to dynamic properties of components of an adjacent lower aggregation level. This interlevel relation takes the form that a number of properties of the lower level logically entail the properties of the higher level component.

In the following a running example used throughout the chapter is introduced to illustrate aggregation by agent clustering in a multi-agent system for co-operative information gathering. For simplicity, this system is considered at two aggregation levels (see Figure 11.1). At the higher level the multi-agent system as a whole is considered. At the lower level four components and their interactions are specified: two information gathering agents A and B, agent C, and environment component E representing the external world. Each of the agents is able to acquire partial information from an external source (component E) by initiated observations. Each agent can be reactive or proactive with respect to the information acquisition process. An agent is proactive if it is able to start information acquisition independently of requests of any other agents, and an agent is reactive if it requires a request from some other agent to perform information acquisition.

Observations of any agent taken separately are insufficient to draw conclusions of a desired type; however, the combined information of both agents is sufficient. Therefore, the agents need to co-operate to be able to draw conclusions. Each agent can be proactive with respect to the conclusion generation, i.e., after receiving both observation results an agent is capable to generate and communicate a conclusion

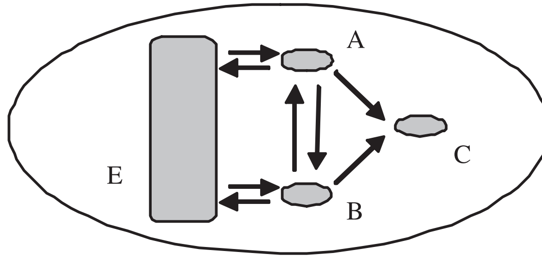

Fig. 11.1 The co-operative information gathering multi-agent system. *A* and *B* represent information gathering agents; *C* is an agent that obtains the conclusion information; *E* is an environmental component.

to agent C. Moreover, an agent can be request pro-active to ask information from another agent, and an agent can be pro-active or reactive in provision of (already acquired) information to the other agent.

For the lower-level components of the multi-agent system, a number of dynamic properties were identified and formalized as it is shown below. In the formalization the variables $A1$ and $A2$ are defined over the sort $AGENT^{TERMS}$, the constant E belongs to the sort $ENVIRONMENTAL_COMPONENT^{GTERMS}$, the variable IC is defined over the sort $INFORMATION_CHUNK^{TERMS}$, the constants $IC1$, $IC2$ and $IC3$ belong to the sort $INFORMATION_CHUNK^{GTERMS}$ and the constant C belongs to the sort $AGENT^{TERMS}$.

DP1(A1, A2) (Effectiveness of information request transfer between agents)

If agent $A1$ communicates a request for an information chunk to agent $A2$ at any time point $t1$, then this request will be received by agent $A2$ at time point $t1+c$.

$\forall IC \forall t1$
$[\ holds(state(\gamma, t1, output(A1)), communicated_from_to(A1, A2, request, IC)))$
$\Rightarrow holds(state(\gamma, t1+c, input(A2)),$
$communicated_from_to(A1, A2, request, IC))]$

DP2(A1, A2) (Effectiveness of information transfer between agents)

If agent $A1$ communicates information chunk to agent $A2$ at any time point $t1$, then this information will be received by agent $A2$ at the time point $t1+c$.

$\forall IC \forall t1$
$[\ holds(state(\gamma, t1, output(A1)), communicated_from_to(A1, A2, inform, IC)))$
$\Rightarrow holds(state(\gamma, t1+c, input(A2)),$
$communicated_from_to(A1, A2, inform, IC)))]$

DP3(A1, E) (Effectiveness of information transfer between an agent and environment)

If agent $A1$ communicates an observation request to the environment at any time point $t1$, then this request will be received by the environment at the time point $t1+c$.

$$\forall IC \forall t1\ [holds(state(\gamma, t1, output(A1)), observation_request_from_for(A1, IC)) \\ \Rightarrow holds(state(\gamma, t1+c, input(E)), observation_request_from_for(A1, IC))]$$

DP4(A1, E) (Information provision effectiveness)

If the environment receives an observation request from agent $A1$ at any time point $t1$, then the environment will generate a result for this request at the time point $t1+c$.

$$\forall IC \forall t1 [holds(state(\gamma, t1, input(E)), observation_request_from_for(A1, IC)) \\ \Rightarrow holds(state(\gamma, t1+c, output(E)), observation_result_to_for(A1, IC))]$$

DP5(E, A1) (Effectiveness of information transfer between environment and an agent)

If the environment generates a result for an agent's information request at any time point $t1$, then this result will be received by the agent at the time point $t1+c$.

$$\forall IC \forall t1 [holds(state(\gamma, t1, output(E)), observation_result_to_for(A1, IC)) \\ \Rightarrow holds(state(\gamma, t1+c, input(A1)), observation_result_to_for(A1, IC))]$$

DP6(A1, A2) (Information acquisition reactiveness)

If agent $A2$ receives a request for an information chunk from agent $A1$ at any time point $t1$, then agent $A2$ will generate a request for this information to the environment at the time point $t1+c$.

$$\forall IC \forall t1 [holds(state(\gamma, t1, input(A2)), communicated_from_to(A1, A2, request, IC)) \\ \Rightarrow holds(state(\gamma, t1+c, output(A2)), observation_result_to_for(A2, IC))]$$

DP7(A1, A2) (Information provision reactiveness)

If exists a time point $t1$ when agent $A2$ received a request for a chunk of information from agent $A1$, then for all time points $t2$ when the requested information is provided to agent $A2$, this information will be further provided by agent $A2$ to agent $A1$ at the time point $t2+c$.

$$\forall IC\ [\exists t1\ [\ t1 < t\ \&\ holds(state(\gamma, t1, input(A2)), \\ communicated_from_to(A1, A2, request, IC))]] \\ \Rightarrow\ \forall t2[\\ t < t2\ \&\ holds(state(\gamma, t2, input(A2)), observation_result_to_for(A2, IC)) \Rightarrow \\ holds(state(\gamma, t2+c, output(A2)), \\ communicated_from_to(A2, A1, inform, IC))]]$$

DP8(A1, A2) (Conclusion proactiveness)

For any time points $t1$ and $t2$, if agent $A1$ receives a result for its observation request from the environment (information chunk $IC1$) at $t1$ and it receives information required for the conclusion generation from agent $A2$ (information chunk $IC2$) at $t2$, then agent $A1$ will generate a conclusion based on the received information (information chunk $IC3$) to agent C at a time point $t4$ later than $t1$ and $t2$.

$$\forall t1, t2\ t1 < t\ \&\ t2 < t \&$$
$$holds(state(\gamma, t1, input(A1)), observation_result_to_for(A1, IC1))\&$$
$$holds(state(\gamma, t2, input(A1)), communicated_from_to(A2, A1, inform, IC2))$$
$$\Rightarrow \exists\ t4 > t\&$$
$$[holds(state(\gamma, t4, output(A1)), communicated_from_to(A1, C, inform, IC3))]$$

DP9(A1, E) (Information acquisition proactiveness)

At some time point an observation request for information chunk IC1 is generated by agent $A1$ to the environment.

$$holds(state(\gamma, c, output(A1)), observation_request_from_for(A1, IC1))$$

DP10(A1, A2) (Information request proactiveness)

At some time point a request for information chunk IC2 is communicated by agent $A1$ to agent $A2$.

$$holds(state(\gamma, c, output(A1)), communicated_from_to(A1, A2, request, IC2))$$

11.4.2 Organisation structures

Organisations have proven to be a useful paradigm for analyzing and designing multi-agent systems [146, 172]. Representation of a multi-agent system as an organisation consisting of roles and groups can tackle major drawbacks concerned with traditional multi-agent models; e.g., high complexity and poor predictability of dynamics in a system [172]. We adopt a generic representation of organisations, abstracted from instances of real agents. As has been shown in [240], organisational structure can be used to limit the scope of interactions between agents, reduce or explicitly increase redundancy of a system, or formalize high-level system goals, of which a single agent may be not aware. Moreover, organisational research has recognized the advantages of agent-based models; e.g., for analysis of structure and dynamics of real organisations.

An *organisation structure* is described by relationships between roles at the same and at adjoining aggregation levels and between parts of the conceptualized environment and roles. The specification of an organisation structure uses the following elements:

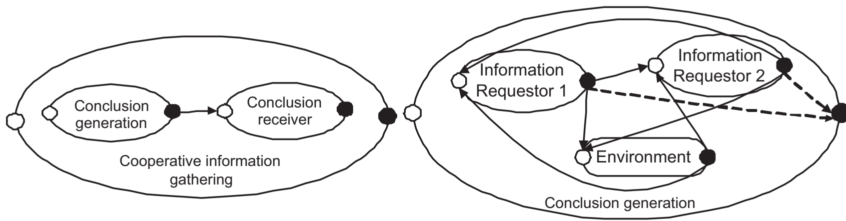

Fig. 11.2 An organisation structure for the co-operative information gathering multi-agent system represented at the aggregation level 2 (left) and at the aggregation level 3 (right).

1. A *role* represents a subset of functionalities, performed by an organisation, abstracted from specific agents who fulfil them.
   Each role can be composed by several other roles, until the necessary detailed level of aggregation is achieved, where a role that is composed of (interacting) subroles, is called a composite role. Each role has an input and an output interface, which facilitate in the interaction (communication) with other roles. The interfaces are described in terms of interaction (input and output) ontologies. At the highest aggregation level, the whole organisation can be represented as one role. Such representation is useful both for specifying general organisational properties and further utilizing an organisation as a component for more complex organisations. Graphically, a role is represented as an ellipse with white dots (the input interfaces) and black dots (the output interfaces). Roles and relations between them are specified using sorts and predicates from the structure ontology (see Table 11.2). For the example of co-operative information gathering system considered in Section 11.4.1, an organisation structure may be defined as shown in Figure 11.2. The structure is represented at three aggregation levels: at the first level the organization as a whole is considered, at the second level the Co-operative information gathering role with its subroles is considered; at the third aggregation level the Conclusion generation role with its subroles is represented.
2. An *interaction link* represents an information channel between two roles at the same aggregation level. Graphically, it is depicted as a solid arrow, which denotes the direction of possible information transfer.
3. *The conceptualized environment* represents a special component of an organisation model. Similarly to roles, the environment has input and output interfaces, which facilitate in the interaction with roles of an organisation. The interfaces are conceptualized by the environment interaction (input and output) ontologies.
4. An *interlevel link* connects a composite role with one of its subroles. It represents information transfer between two adjacent aggregation levels. It may describe an ontology mapping for representing mechanisms of information abstraction. Graphically, it is depicted as a dashed arrow, which shows the direction of the interlevel transition.

Table 11.2 Ontology for formalizing organizational structure

Predicate	Description
$is_role : ROLE$	Specifies a role in an organization
$has_subrole : ROLE \times ROLE$	For a subrole of a composite role
$source_of_interaction : ROLE \times INTERACTION_LINK$	Specifies a source role of an interaction
$destination_of_interaction : ROLE \times INTERACTION_LINK$	Specifies a destination role of interaction
$interlevel_connection_from : ROLE \times INTERLEVEL_LINK$	Identifies a source role of an interlevel link
$interlevel_connection_to : ROLE \times INTERLEVEL_LINK$	Identifies a destination role of an interlevel link
$part_of_env_in_interaction : ENVIRONMENT \times ENVIRONMENT_INTERACTION_LINK$	Identifies the conceptualized part of the environment involved in interaction with a role
$has_input_ontology : ROLE \times ONTOLOGY$	Specifies an input ontology for a role
$has_output_ontology : ROLE \times ONTOLOGY$	Specifies an output ontology for a role
$has_input_ontology : ENVIRONMENT \times ONTOLOGY$	Specifies an input ontology for the environment
$has_output_ontology : ENVIRONMENT \times ONTOLOGY$	Specifies an output ontology for the environment
$has_interaction_ontology : ROLE \times ONTOLOGY$	Specifies an interaction ontology for a role

At each aggregation level, it can be specified how the organization's behaviour is assumed to be. The dynamics of each structural element are defined by the specification of a set of dynamic properties. We define five types of dynamic properties:

1. A *role property* (RP) describes the relationship between input and output states of a role, over time. For example, a role property of Information requester 2 is:

   Information acquisition reactiveness

   $\forall IC \forall t1\, [holds(state(\gamma, t1, input(InformationRequester2)),$
   $communicated_from_to(InformationRequester1, InformationRequester2,$
   $request, IC))$
   $\Rightarrow holds(state(\gamma, t1 + c, output(InformationRequester2)),$
   $observation_result_to_for(InformationRequester2, IC))]$

2. A *transfer property* (TP) describes the relationship of the output state of the source role of an interaction link to the input state of the destination role. For example, a transfer property for the roles Information requester 1 and Information requester 2 is:

   Effectiveness of information transfer between roles

   $\forall IC \forall t1\; [holds(state(\gamma, t1, output(InformationRequester1)),$
   $communicated_from_to(InformationRequester1, InformationRequester2,$
   $inform, IC)))$

$\Rightarrow holds(state(\gamma, t1+c, input(InformationRequester2)),$
$communicated_from_to(InformationRequester1, InformationRequester2,$
$inform, IC)))]$

3. An *interlevel link property* (ILP) describes the relationship between the input or output state of a composite role and the input or output state of its subrole. Note that an interlevel link is considered to be instantaneous: it does not represent a temporal process, but gives a different view (using a different ontology) on the same information state. An interlevel transition is specified by an ontology mapping, which can include information abstraction.
4. An *environment property* (EP) describes a temporal relationship between states or properties of objects of interest in the environment.
5. An *environment interaction property* (EIP) describes a relation either between the output state of the environment and the input state of a role (or an agent) or between the output state of a role (or an agent) and the input state of the environment. For example,

Effectiveness of information transfer between a role and environment
$\forall IC \forall t1\ [holds(state(\gamma, t1, output(InformationRequester1)),$
$observation_request_from_for(InformationRequester1, IC))$
$\Rightarrow holds(state(\gamma, t1+c, input(E)),$
$observation_request_from_for(InformationRequester1, IC))]$

The specifications of organisation structural relations and dynamics are imposed onto the agents, who will eventually enact the organisational roles. For more details on organisation-oriented modelling of multi-agent systems we refer to [263].

11.5 Relation to Other Languages

In this section TTL is compared to a number of existing languages for modelling dynamics of a system.

Executable languages can be defined as sublanguages of TTL. An example of such a language, which was designed for simulation of dynamics in terms of both qualitative and quantitative concepts, is the LEADSTO language, cf. [81]. The LEADSTO language models direct temporal or causal dependencies between two state properties in states at different points in time as follows. Let α and β be state properties of the form 'conjunction of atoms or negations of atoms', and e, f, g, h non-negative real numbers (constants of sort $VALUE$). In LEADSTO the notation $\alpha \longrightarrow_{e,f,g,h} \beta$, means:

If state property α holds for a certain time interval with duration g, then after some delay (between e and f) state property β will hold for a certain time interval of length h.

A specification in LEADSTO format has as advantages that it is executable and that it can often easily be depicted graphically, in a causal graph or system dynamics style. In terms of TTL, the fact that the above statement holds for a trace γ is expressed as follows:

$$\forall t1[\forall t[t1 - g \leq t \;\&\; t < t1 \Rightarrow holds(state(\gamma, t), \alpha)] \Rightarrow$$
$$\exists d : VALUE[e \leq d \;\&\; d \leq f \;\&\; \forall t'[t1 + d \leq t' \;\&\; t' < t1 + d + h \Rightarrow$$
$$holds(state(\gamma, t'), \beta)]$$

Furthermore, TTL has some similarities with the situation calculus [365] and the event calculus [272]. However, a number of important syntactic and semantic distinctions exist between TTL and both calculi. In particular, the central notion of the situation calculus - a situation - has different semantics than the notion of a state in TTL. That is, by a situation is understood a history or a finite sequence of actions, whereas a state in TTL is associated with the assignment of truth values to all state properties (a 'snapshot' of the world). Moreover, in contrast to situation calculus, where transitions between situations are described by execution of actions, in TTL action executions are used as properties of states.

Moreover, although a time line has been introduced to the situation calculus [339], still only a single path (a temporal line) in the tree of situations can be explicitly encoded in the formulae. In contrast, TTL provides more expressivity by allowing explicit references to different temporally ordered sequences of states (traces) in dynamic properties. For example, this can be useful for expressing the property of trust monotonicity:

> For any two traces $\gamma1$ and $\gamma2$, if at each time point t agent A's experience with public transportation in $\gamma2$ at t is at least as good as A's experience with public transportation in $\gamma1$ at t, then in trace $\gamma2$ at each point in time t, A's trust is at least as high as A's trust at t in trace $\gamma1$.

$$\forall \gamma1, \gamma2[\forall t, \forall v1 : VALUE[holds(state(\gamma1, t), has_value(experience, v1))\&$$
$$[\forall v2 : VALUE\ holds(state(\gamma2, t), has_value(experience, v2) \rightarrow v1 \leq v2)]] \Rightarrow$$
$$[\forall t, \forall w1 : VALUE[holds(state(\gamma1, t), has_value(trust, w1))\&$$
$$[\forall w2 : VALUE\ holds(state(\gamma2, t), has_value(trust, w2) \rightarrow w1 \leq w2)]]]]$$

In contrast to the event calculus, TTL does not employ the mechanism of events that initiate and terminate fluents. Event occurrences in TTL are considered to be state occurrences the external world. Furthermore, similarly to the situation calculus, also in the event calculus only one time line is considered.

Formulae of the loosely guarded fragment of the first-order predicate logic [16], which is decidable and has good computational properties (deterministic exponential time complexity), are also expressible in TTL:

$$\exists y((\alpha_1 \wedge ... \wedge \alpha_m) \ \wedge \ \psi(x, y)) \text{ or } \forall y((\alpha_1 \wedge ... \wedge \alpha_m) \rightarrow \psi(x, y)),$$

where x and y are tuples of variables, $\alpha_1 ... \alpha_m$ are atoms that relativize a quantifier (the guard of the quantifier), and $\psi(x,y)$ is an inductively defined formula in the guarded fragment, such that each free variable of the formula is in the set of free variables of the guard.

Similarly the fluted fragment [348] and $\exists * \forall *$ [3] can be considered as sublanguages of TTL.

TTL can also be related to temporal languages that are often used for verification (e.g., LTL and CTL [39, 198]). For example, dynamic properties expressed as formulae in LTL can be translated to TTL by replacing the temporal operators of LTL by quantifiers over time. E.g., consider the LTL formula

$$\mathbf{G}(observation_result(itsraining) \rightarrow \mathbf{F}(belief(itsraining)))$$

where the temporal operator **G** means 'for all later time points', and **F** 'for some later time point'. The first operator can be translated into a universal quantifier, whereas the second one can be translated into an existential quantifier.

Using TTL, this formula then can be expressed, for example, as follows:

$$\forall t1[holds(state(\gamma,t1), observation_result(itsraining)) \Rightarrow$$
$$\exists t2 > t1\ holds(state(\gamma,t2), belief(itsraining))]$$

Note that the translation is not bi-directional, i.e., it is not always possible to translate TTL expressions into LTL expressions due to the limited expressive power of LTL. For example, the property of trust monotonicity specified in TTL above cannot be expressed in LTL because of the explicit references to different traces. Similar observations apply for other well-known modal temporal logics such as CTL.

In contrast to the logic of McDermott [309], TTL does not assume structuring of traces in a tree. This enables reasoning about independent sequences of states (histories) in TTL (e.g., by comparing them), which is also not addressed by McDermott.

11.6 Normal Forms and Transformation Procedures

In this Section, normal forms for TTL formulae and the related transformation procedures are described. Normal forms create the basis for the automated analysis of TTL specifications, which is addressed later in this chapter. In Section 11.6.1 the past implies future normal form and a procedure for transformation of any TTL formula into this form are introduced. In Section 11.6.2 the executable normal form and a procedure for transformation of TTL formulae in the past implies future normal form into the executable normal form are described. A procedure for abstraction of executable specifications is described in Section 11.6.3.

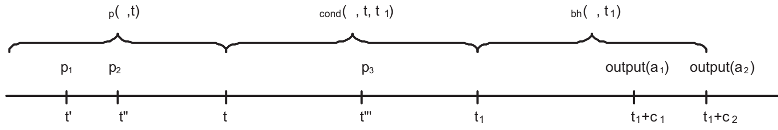

Fig. 11.3 Graphical illustration of the structure of the past implies future normal form

11.6.1 Past Implies Future Normal Form

First, the past implies future normal form is introduced.

Definition 11.1. (Past Implies Future Normal Form) The past implies future normal form for TTL formulae is specified by a logical implication from a temporal input pattern to a temporal output pattern:

$$[\varphi_p(\gamma,t) \Rightarrow \varphi_f(\gamma,t)],$$

where $\varphi_p(\gamma,t)$ is a past statement (i.e., for all time variables s in $\varphi_p(\gamma,t) : s \leq t$ or $s < t$) and $\varphi_f(\gamma,t)$ is a future statement (i.e., for all time variables s in $\varphi_f(\gamma,t) : s \geq t$ or $s > t$). The future statement is represented in the form of a conditional behaviour:

$$\varphi_f(\gamma,t) \Leftrightarrow \forall t_1 > t[\varphi_{cond}(\gamma,t,t_1) \Rightarrow \varphi_{bh}(\gamma,t_1)],$$

where $\varphi_{cond}(\gamma,t,t_1)$ is an interval statement over the interaction ontology, which describes a condition for some specified action(s) and/or communication(s), and $\varphi_{bh}(\gamma,t_1)$ is a (conjunction of) future statement(s) for t_1 over the output ontology of the form $holds(state(\gamma,t_1+c),output(a))$, for some integer constant c and action or communication a.

A graphical illustration of the structure of the past implies future normal form is given in Figure 11.3. When a past formula $\varphi_p(\gamma,t)$ is true for γ at time t, a potential to perform one or more action(s) and/or communication(s) exists. This potential is realized at time t_1 when the condition formula $\varphi_{cond}(\gamma,t,t_1)$ becomes true, which leads to the action(s) and/or communication(s) being performed at the time point(s) t_1+c indicated in $\varphi_{bh}(\gamma,t_1)$.

For example, the dynamic property $DP7(A1,A2)$ (Information provision reactiveness) from the specification of co-operative information gathering multi-agent system from Section 11.4.1 can be specified in the past implies future normal form $\varphi_p(\gamma,t) \Rightarrow \varphi_f(\gamma,t)$, with $\varphi_p(\gamma,t)$ is a formula

$\exists t2 \leq t$ & $holds(state(\gamma,t2,input(A2)),communicated_from_to(A1,A2,$ $request,IC))$

and $\varphi_f(\gamma, t)$ is a formula

$$\forall t1 > t\ [holds(state(\gamma, t1, input(A2)), observation_result_to_for(A2, IC)) \Rightarrow$$

$$holds(state(\gamma, t1 + c, output(A2)), communicated_from_to(A2, A1, inform, IC))]$$

with $\varphi_{cond}(\gamma, t, t1)$ is a formula

$$holds(state(\gamma, t1, input(A2)), observation_result_to_for(A2, IC))$$

and $\varphi_{bh}(\gamma, t1)$ is $holds(state(\gamma, t1 + c, output(A2)), communicated_from_to(A2, A1, inform, IC))]$,

where t is the present time point with respect to which the formulae are evaluated and c is some natural number.

In general, any TTL formula can be automatically translated into the past implies future normal form. The transformation procedure is based on a number of steps. First, the variables in the formula are related to the given t (current time point) by differentiation. The resulting formula is rewritten in prenex conjunctive normal form. Each clause in this formula is reorganised in past implies future format. Finally, the quantifiers are distributed over and within these implications. Now consider the detailed description of these steps (for a more profound description of the procedure see [418]).

Differentiating Time Variables A formula is rewritten into an equivalent one such that time variables that occur in this formula always either are limited (relativized) to past or to future time points with respect to t. As an example, suppose $\psi(t_1, t_2)$ is a formula in which time variables t_1, t_2 occur. Then, different cases of ordering relation for each of the time variables with respect to t are considered: $t_1 < t$, $t_1 \geq t$ and $t_2 < t$, $t_2 \geq t$, i.e., in combination four cases: $t_1 < t$ and $t_2 < t$, $t_1 < t$ and $t_2 \geq t$, $t_1 \geq t$ and $t_2 < t$, $t_1 \geq t$ and $t_2 \geq t$. To eliminate ambiguity, for $t_i < t$ the variable t_i is replaced by (*past time variable*) u_i, for $t_i \geq t$ by (*future time variable*) v_i.

The following transformation step introduces for any occurring time variable t_i a differentiation into a pair of new time variables: u_i used over the past and v_i used over the future with respect to t.

For any occurrence of a universal quantifier over t_i:

$$\forall t_i\ A \longmapsto [\forall u_i < t\ A[u_i/t_i] \wedge \forall v_i \geq t\ A[v_i/t_i]]$$

For any occurrence of an existential quantifier over t_i:

$$\exists t_i\ A \longmapsto [\exists u_i < t\ A[u_i/t_i] \vee \exists v_i \geq t\ A[v_i/t_i]]$$

Assuming differentiation of time variables into past and future time variables, state-related atoms (in which only one time variable occurs) can be classified in a straightforward manner as a past atom or future atom. For example, atoms of the form $holds(state(\gamma, u_i), p)$ are past atoms and $holds(state(\gamma, v_j), p)$ are future atoms. For non-unary relations, in the special case of the time ordering relation ¡ the ordering axioms are given, e.g., transitivity. Atoms that are mixed (containing both a past and a future variable) are eliminated by the following transformation rules:

$$u_i = v_j \rightarrow false \quad v_j < u_i \rightarrow false \quad u_i < v_j \rightarrow true$$

Obtaining prenex conjunctive normal form This step is performed using a well-known transformation procedure [180].

From a Clause to a Past to Future Implication By partitioning the set of occurring atoms into past atoms and future atoms, it is not difficult to rewrite a clause into a past to future implication format: transform a clause C into an implication of the form $A \rightarrow B$ where A is the conjunction of the negations of all past literals in C and B is the disjunction of all future literals in C. Thus, a quantifier free formula in Conjunctive Normal Form can be transformed into a conjunction of implications from past to future by the transformation rule

$\vee \; PL_i \vee \vee FL_j \;\longmapsto\; \wedge \;\sim PL_i \;\rightarrow \vee \; FL_j$

where the past and future literals are indicated by PL_i and FL_j, respectively, and if a is an atom, $\sim a = \neg a$, and $\sim \neg a = a$.

Distribution of Quantifiers Over Implications The quantifiers can be rewritten to quantifiers with a single implication as their scope, and even one step further, to quantifiers with a single antecedent or a single consequent of an implication as their scope. Notice that quantifiers addressed here are both time quantifiers and non-time quantifiers.

Let φ be a formula in the form of a conjunction of past to future implications $\wedge_{i\in I}[A_i \rightarrow B_i]$ and let x be a (either past or future) variable occurring in φ. The following transformation rules handle existential quantifiers for variables in one or more of the B_i, respectively in one or more of the A_i. Here P denotes taking the power set.

1. if x occurs in the B_i but does not occur in the A_i :
   $\exists x \;\wedge_{i\in I}\; [A_i \rightarrow B_i] \longmapsto \wedge_{j\in P(I)} \;\exists x[\wedge_{i\in j} A_i \rightarrow \wedge_{i\in j} B_i]$
   $\exists x[\wedge_{i\in j} A_i \rightarrow \wedge_{i\in j} B_i] \longmapsto [\wedge_{i\in j} A_i \rightarrow \exists x \wedge_{i\in j} B_i]$
2. if x occurs in the A_i but does not occur in the B_i :
   $\exists x \wedge_{i\in I} [A_i \rightarrow B_i] \longmapsto \wedge_{j\in P(I)} \exists x[\vee_{i\in j} A_i \rightarrow \vee_{i\in j} B_i]$
   $\exists x[\vee_{i\in j} A_i \rightarrow \vee_{i\in j} B_i] \longmapsto [\forall x[\vee_{i\in j} A_i] \rightarrow \vee_{i\in j} B_i]$

The following transformation rules handle universal quantifiers for variables in one or more of the B_i, respectively in one or more of the A_i:

1. if x occurs in the A_i or in the B_i :
   $\forall x \;\wedge_{i\in I} [A_i \rightarrow B_i] \;\longmapsto\; \wedge_{i\in I} \forall x[A_i \rightarrow B_i]$
2. if x occurs in the B_i but does not occur in the A_i :
   $\forall x[A_i \rightarrow B_i] \;\longmapsto\; A_i \rightarrow \forall x B_i$
3. if x occurs in the A_i but does not occur in the B_i :
   $\forall x[A_i \rightarrow B_i] \;\longmapsto\; [\exists x A_i] \rightarrow B_i$

11.6.2 Executable Normal Form

Although the past implies future normal form imposes a standard structure on the representation of TTL formulae, it does not guarantee the executability of formulae, required for automated analysis methods (i.e., some formulae may still contain complex temporal relations that cannot be directly represented in analysis tools). Therefore, to enable automated analysis, normalized TTL formulae should be translated into an executable normal form.

Definition 11.2. Executable Normal Form A TTL formula is in executable normal form if it has one of the following forms, for certain state properties , X and Y with $X \neq Y$, and integer constant c.

1. $\forall t\ holds(state(\gamma,t),X) \Rightarrow holds(state(\gamma,t+c),Y)$ (states relation property)
2. $\forall t\ holds(state(\gamma,t),X) \Rightarrow holds(state(\gamma,t+1),X)$ (persistency property)
3. $\forall t\ holds(state(\gamma,t),X) \Rightarrow holds(state(\gamma,t),Y)$ (state relation property)

For the translation postulated internal states of a component(s) specified in the formula, are used. These auxiliary states include memory states that are based on (input) observations (sensory representations) or communications ($memory : LTIME^{TERMS} \times STATPROP \rightarrow STATROP$). For example, $memory(t, observed(a))$ expresses that the component has memory that it observed a state property a at time point t. Furthermore, before performing an action or communication it is postulated that a component creates an internal preparation state. For example, $preparation_for(b)$ represents a preparation of a component to perform an action or a communication.

In the following a transformation procedure from the normal form $[\varphi_p(\gamma,t) \Rightarrow \varphi_f(\gamma,t)]$ for the property $\varphi_p(\gamma,t)$ to the executable normal form is described and illustrated for the property $DP7(A1,A2)$ (Information provision reactiveness) considered above. For a more profound description of the transformation procedure we refer to [398].

First, an intuitive explanation for the procedure is provided. The procedure transforms a non-executable dynamic property in a number of executable properties. These properties can be seen as an execution chain, which describes the dynamics of the non-executable property. In this chain each unit generates intermediate states, used to link the following unit. In particular, first a number of properties are created to generate and maintain memory states (step 1 below). These memory states are used to store information about the past dynamics of components, which is available afterwards at any point in time. Then, executable properties are created to generate preparation for output and output states of components (steps 2 and 3 below). In these properties temporal patterns based on memory states are identified required for generation of particular outputs of components. In the end all created properties are combined in one executable specification.

More specifically, the transformation procedure consists of the following steps:

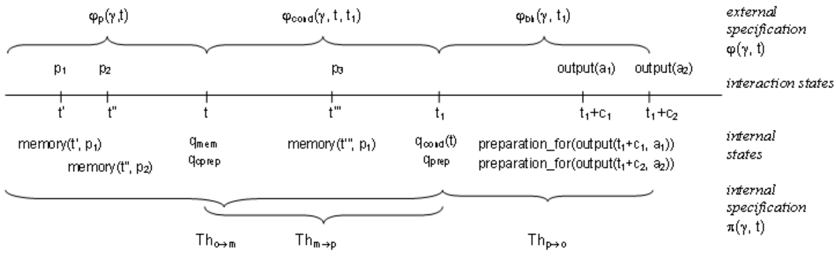

Fig. 11.4 A graphical representation of relations between interaction states described by a non-executable dynamic property and internal states described by executable rules.

1. Identify executable temporal properties, which describe transitions from the component states specified in $\varphi_p(\gamma,t)$ to memory states (for a graphical representation of relations between the states considered in this procedure see Figure 11.4).
   The general rules that form the basis for the executable properties are the following:
   $\forall t'\ holds(state(\gamma,t'),p) \Rightarrow holds(state(\gamma,t'),memory(t',p))$
   $\forall t''\ holds(state(\gamma,t''),memory(t',p)) \Rightarrow\ holds(state(\gamma,t''+1),memory(t',p))$

   Furthermore, at this step the memory formula $\varphi_{mem}(\gamma,t)$ is defined that is obtained by replacing all occurrences in $\varphi_p(\gamma,t)$ of subformulae of the form $holds(state(\gamma,t'),p)$ by $holds(state(\gamma,t),memory(t',p))$. According to Lemma 1 (given in [398]) $\varphi_{mem}(\gamma,t)$ is equivalent to some formula $\delta^*(\gamma,t)$ of the form $holds(state(\gamma,t),q_{mem}(t))$, where $q_{mem}(t)$ is called *the normalized memory state formula* for $\varphi_{mem}(\gamma,t)$, which uniquely describes the present state at the time point t by a certain history of events. Moreover, q_{mem} is the state formula $\forall u'\ [present_time(u') \rightarrow q_{mem}(u')]$.

   For the property $DP7(A1,A2)$:
   $\forall t'\ holds(state(\gamma,t',input(A2)),communicated_from_to(A1,A2,request,IC)))$
   $\Rightarrow\ holds(state(\gamma,t',internal(B)),$
   $memory(t',communicated_from_to(A1,A2,request,IC))))$
   $\forall t''\ holds(state(\gamma,t'',internal(B)),$
   $memory(t',communicated_from_to(A1,A2,request,IC)))) \Rightarrow$
   $holds(state(\gamma,t''+1,internal(B)),$
   $memory(t',communicated_from_to(A1,A2,request,IC))))$

2. Identify executable temporal properties, which describe transitions from memory states to preparation states for output. At this step the following formulae are defined: *The condition memory formula* $\varphi_{cmem}(\gamma,t,t_1)$ is obtained by replacing all occurrences in $\varphi_{cond}(\gamma,t,t_1)$ of $holds(state(\gamma,t'),p)$ by $holds(state(\gamma,t_1), memory(t',p))$. $\varphi_{cmem}(\gamma,t,t_1)$ contains a history of events, between the time point t, when $\varphi_p(\gamma,t)$ is true and the time point t_1, when the formula $\varphi_{cond}(\gamma,t,t_1)$ becomes true. Again by Lemma 1 $\varphi_{cmem}(\gamma,t,t_1)$ is equivalent to the formula $holds(state(\gamma,t_1),q_{cond}(t,t_1))$, where $q_{cond}(t,t_1)$ is called *the normalized condition state formula for* $\varphi_{cmem}(\gamma,t,t_1)$, and $q_{cond}(t)$ is the state formula $\forall u'$ $[present_time(u') \rightarrow q_{cond}(t,u')]$. The state formula constructed by Lemma 1 for the preparation formula $\varphi_{prep}(\gamma,t_1)$ is called *the (normalized) preparation state formula* and denoted by $q_{prep}(t_1)$. Moreover, q_{prep} is the state formula $\forall u'$ $[present_time(u') \rightarrow q_{prep}(u')]$. The formula $\varphi_{cprep}(\gamma,t_1)$ of the form $holds(state(\gamma,t_1), \forall u1 > t[q_{cond}(t,u1) \rightarrow q_{prep}(u1)])$ is called *the conditional preparation formula* for $\varphi_f(\gamma,t)$. The state formula $\forall u1 > t[q_{cond}(t,u1) \rightarrow q_{prep}(u1)]$ is called *the normalized conditional preparation state formula* for $\varphi_{cprep}(\gamma,t)$ and denoted by $q_{cprep}(t)$. Moreover, q_{cprep} is the formula $\forall u'$ $[present_time(u') \rightarrow q_{cprep}(u')]$.

   The general executable rules that form basis for executable properties are defined as follows:

   $\forall t'\ holds(state(\gamma,t'),p) \Rightarrow holds(state(\gamma,t'), memory(t',p) \wedge stimulus_reaction(p))$

   $\forall t'',t'\ holds(state(\gamma,t''), memory(t',p)) \Rightarrow holds(state(\gamma,t''+1), memory(t',p))$

   $\forall t'\ holds(state(\gamma,t'),q_{mem}) \Rightarrow holds(state(\gamma,t'),q_{cprep})$

   $\forall t',t\ holds(state(\gamma,t'), q_{cprep} \wedge q_{cond}(t) \wedge \wedge_p stimulus_reaction(p)) \Rightarrow holds(state(\gamma,t'),q_{prep})$

   $\forall t'\ holds(state(\gamma,t'), stimulus_reaction(p) \wedge \neg preparation_for(output(t'+c,a))) \Rightarrow holds(state(\gamma,t'+1), stimulus_reaction(p))$

   $\forall t'\ holds(state(\gamma,t'), preparation_for(output(t'+c,a)) \wedge \neg output(a)) \Rightarrow holds(state(\gamma,t'+1), preparation_for(output(t'+c,a)))$

   $\forall t'\ holds(state(\gamma,t'), present_time(t') \wedge \forall u'[present_time(u') \rightarrow preparation_for(output(u'+c),a)]) \rightarrow holds(state(\gamma,t'), preparation_for(output(t'+c),a))$

The auxiliary functions $stimulus_reaction(a)$ are used for reactivation of component preparation states for generating recurring actions or communications.

For the property $DP7(A1, A2)$:

$\forall t' [holds(state(\gamma, t', input(A2)), observation_result_to_for(A2, IC))$
$\Rightarrow holds(state(\gamma, t', internal(A2)),$
$memory(t', observation_result_to_for(A2, IC)) \wedge$
$stimulus_reaction(observation_result_to_for(A2, IC))]])$

$\forall t''\ holds(state(\gamma, t'', internal(A2)),$
$memory(t', observation_result_to_for(A2, IC))) \Rightarrow$
$holds(state(\gamma, t'' + 1, internal(A2)),$
$memory(t', observation_result_to_for(A2, IC)))$

$\forall t'\ holds(state(\gamma, t'), \forall u''[present_time(u'') \rightarrow$
$\exists u2[memory(u2, communicated_from_to(A1, A2, request, IC))]]) \Rightarrow$
$holds(state(\gamma, t'), \forall u'''[present_time(u''') \rightarrow [\forall u1 > u'''$
$[memory(u1, observation_result_to_for(A2, IC)) \rightarrow$
$preparation_for(output(u1 + c,$
$communicated_from_to(A2, A1, inform, IC)))]]])$

$\forall t', tholds(state(\gamma, t'), [\forall u'''[present_time(u''') \rightarrow [\forall u1 > u'''$
$[memory(u1, observation_result_to_for(A2, IC))) \rightarrow$
$preparation_for(output(u1 + c,$
$communicated_from_to(A2, A1, inform, IC)))]]]$
$\wedge \forall u''[present_time(u'') \rightarrow$
$memory(u'', observation_result_to_for(A2, IC)))] \wedge$
$stimulus_reaction(observation_result_to_for(A2, IC)))]) \Rightarrow$
$holds(state(\gamma, t', internal(A2)), \forall u1[present_time(u1) \rightarrow$
$preparation_for(output(u1 + c,$
$communicated_from_to(A2, A1, inform, IC))))])$

$\forall t'\ holds(state(\gamma, t'), stimulus_reaction(observation_result_to_for(A2, IC)) \wedge$
$not(preparation_for(output(t' + c,$
$communicated_from_to(A2, A1, inform, IC)))]) \Rightarrow$
$holds(state(\gamma, t' + 1), stimulus_reaction(observation_result_to_for(A2, IC)))$

$\forall t'\ holds(state(\gamma, t', internal(A2)),$

$[preparation_for(output(t' + c, observation_result_to_for(A2, IC)))$

$\wedge\ not(output(observation_result_to_for(A2, IC)))]) \Rightarrow$

$holds(state(\gamma, t' + 1, internal(A2)),$

$preparation_for(output(t' + c, observation_result_to_for(A2, IC))))$

3. Specify executable properties, which describe the transition from preparation states to the corresponding output states.

   The preparation state $preparation_for(output(t1 + c, a))$ is followed by the output state, created at the time point t_{1+c}. The general executable rule is the following:

   $\forall t'\ holds(state(\gamma, t'), preparation_for(output(t' + c, a))) \Rightarrow holds(state(\gamma, t' + c), output(a))$

   For the property $DP7(A1, A2)$:

   $\forall t'\ holds(state(\gamma, t', internal(A2)),$

   $preparation_for(output(t' + c,$
   $communicated_from_to(A2, A1, inform, IC)))) \Rightarrow$

   $holds(state(\gamma, t' + c, output(A2)),$

   $output(communicated_from_to(A2, A1, inform, IC)))$

To automate the proposed procedure the software tool was developed in $Java^{TM}$. The transformation algorithm searches in the input file for the standard predicate names and the predefined structures, then performs string transformations that correspond precisely to the described steps of the translation procedure, and adds executable rules to the output specification file.

11.6.3 Abstraction of executable specifications

Sometimes (executable) specifications of multi-agent systems may be very detailed, with opaque global dynamics. To establish higher level dynamic properties of such systems, abstraction of specifications can be performed. In particular, internal dynamics of agents described by executable cognitive specifications may be abstracted to behavioural (or interaction) specifications of agents as shown in [399]. To express properties of behavioural and cognitive specifications past and past-present statements are used.

Definition 11.3. (Past-Present Statement) A past-present statement (abbreviated as a pp-statement) is a statement φ of the form $B \Leftrightarrow H$, where the formula B, called the body and denoted by $body(\varphi)$, is a past statement for t, and H, called the head and denoted by $head(\varphi)$, is a statement of the form $holds(state(\gamma,t),p)$ for some state property p.

It is assumed that each output state of an agent A specified by an atom $holds(state(\gamma,t),\psi)$ is generated based on some input and internal agent's dynamics that can be specified by a set of formulae over $\varphi(\gamma,t) \Rightarrow holds(state(\gamma,t),\psi)$ with φ a past statement over $InOnt(A) \cup IntOnt(A)$. Furthermore, a completion can be made (similar to Clark's completion in logic programming) that combines all statements $[\varphi_1(\gamma,t) \Rightarrow holds(state(\gamma,t),\psi), \varphi_2(\gamma,t) \Rightarrow holds(state(\gamma,t),\psi), ..., \varphi_n(\gamma,t) \Rightarrow holds(state(\gamma,t),\psi)]$ with the same consequent in the specification, into one past-present-statement $\varphi_1(\gamma,t) \vee \varphi_2(\gamma,t) \vee ... \vee \varphi_n(\gamma,t) \Leftrightarrow holds(state(\gamma,t),\psi)$. Sometimes this statement is called *the definition* of $holds(state(\gamma,t),\psi)$.

Furthermore, the procedure is applicable only to cognitive specifications that can be stratified.

Definition 11.4. (Stratification of a Specification) An agent specification Π is stratified if there is a partition $\Pi = \Pi_1 \cup ... \cup \Pi_n$ into disjoint subsets such that the following condition holds: for $i > 1$: if a subformula $holds(state(\gamma,t),\varphi)$ occurs in a body of a statement in Π_i, then it has a definition within $\cup_{j \leq i} \Pi_j$.

The notation $\varphi[holds_1,...,holds_n]$ is used to denote a formula φ with $holds_1$, ..., $holds_n$ as its atomic subformulae.

The rough idea behind the procedure is as follows. Suppose for a certain cognitive state property the pp-specification $B \Leftrightarrow holds(state(\gamma,t),p)$ is available; here the formula B is a past statement for t. Moreover, suppose that in B only two atoms of the form $holds(state(\gamma,t1),p1)$ and $holds(state(\gamma,t2),p2)$ occur, whereas as part of the cognitive specification also specifications $B1 \Leftrightarrow holds(state(\gamma,t1),p1)$ and $B2 \Leftrightarrow holds(state(\gamma,t2),p2)$ are available. Then, within B the atoms can be replaced (by substitution) by the formula $B1$ and $B2$. This results in a

$$B[B1/holds(state(\gamma,t1),p1), B2/holds(state(\gamma,t2),p2)] \Leftrightarrow holds(state(\gamma,t),p)$$

which again is a pp-specification. Here for any formula C the expression $C[x/y]$ denotes the formula C transformed by substituting x for y. Such a substitution corresponds to an abstraction step. For the general case the procedure includes a sequence of abstraction steps; the last step produces a behavioural specification that corresponds to a cognitive specification.

Let us describe and illustrate the procedure for a simple executable pp-specification that corresponds to the property $DP7(A1,A2)$ considered in Section 11.6.2:

CP1(A1, A2) (memory state generation and persistence)

$holds(state(\gamma,t1,internal(A2)),$
$memory(t2,communicated_from_to(A1,A2,request,IC))) \Leftrightarrow$
$\exists t2\ t2 < t1\ \&\ holds(state(\gamma,t2,input(A2)),$
$communicated_from_to(A1,A2,request,IC))$

CP2(A1, A2) (conclusion generation)

$holds(state(\gamma,t3,output(A2)),communicated_from_to(A2,A1,inform,IC)) \Leftrightarrow$
$\exists t4,t5\ t4 < t3\ \&\ t5 < t4\ \&\ holds(state(\gamma,t4,internal(A2)),$
$memory(t5,communicated_from_to(A1,A2,request,IC)))\&$
$holds(state(\gamma,t4,input(A2)),observation_result_to_for(A2,IC))$

To obtain an abstracted specification for a specification X the following sequence of steps is followed:

1. Enforce temporal completion on X.
2. Stratify X:
   a. Define the set of formulae of the first stratum ($h = 1$) as: $\{\varphi_i : holds(state(\gamma,t),a_i) \leftrightarrow \psi i_p(holds_1,...,holds_m) \in X | \forall k\ m \geq k \geq 1\ \ holds_k$ is expressed using $InOnt\}$;

      proceed with $h = 2$.

      In the considered example $CP1(A1,A2)$ belongs to the first stratum.
   b. The set of formulae for stratum h is identified as $\{\varphi_i : holds(state(\gamma,t),a_i) \leftrightarrow \psi i_p(holds_1,...,holds_m) \in X | \forall k\ m \geq k \geq 1\ \exists l$ $l < h\ \exists \psi \in STRATUM(X,l)\ AND\ head(\psi) = holds_k\ AND\ \exists j\ m \geq j \geq 1\ \exists \xi \in STRATUM(X,h-1)\ AND\ head(\xi) = holds_j\}$;

      proceed with $h = h+1$.

      In the considered example $CP2(A1,A2)$ belongs to the stratum 2.
   c. Until a formula of X exists not allocated to a stratum, perform 2b.
3. Replace each formula of the highest stratum n $\varphi_i : holds(state(\gamma,t),a_i) \leftrightarrow \psi i_p(holds_1,...,holds_m)$ by $\varphi_I \delta$ with renaming of temporal variables if required, where $\delta = \{holds_k \backslash body(\varphi_k)$ such that $\varphi_k \in X$ and $head(\varphi_k) = holds_k\}$. Further, remove all formulae $\{\varphi \in STRATUM(X,n-1) | \exists \psi \in STRATUM(X,n)$ $AND\ head(\varphi)$ is a subformula of the $body(\varphi)\}$).

   In the considered example the atom $holds(state(\gamma,t4,internal(A2)),memory(t5,communicated_from_to($ $A1,A2,request,IC)))$ in $CP2$ is replaced by its definition given by $CP1$:

   $BP1 : holds(state(\gamma,t3,output(A2)),$
   $communicated_from_to(A2,A1,inform,IC))$

$\Leftrightarrow \exists t4, t5\ t4 < t3\ \&\ t5 < t4 \&\ holds(state(\gamma, t5, input(A2)),$
$communicated_from_to(A1, A2, request, IC)) \&\ holds(state(\gamma, t4, input(A2)),$
$observation_result_to_for(A2, IC))$

Furthermore, both $CP1$ and $CP2$ are removed from the specification. Thus, the obtained property is a behavioural specification expressed using *InOnt* and *OutOnt* only that corresponds to the considered cognitive specification.

4. Append the formulae of the stratum n to the stratum $n-1$, which now becomes the highest stratum (i.e, $n = n-1$).

   For the example, BP1 becomes the only property that belongs to the stratum 1.

5. Until $n > 1$, perform steps 3 and 4.

The algorithm has been implemented in $Java^{TM}$. The worst case time complexity is $O(|X|^2)$. The representation of a higher level specification Φ is more compact than of the corresponding lower level specification Π. First, only *IntOnt* is used to specify the formulae of Φ, whereas $InOnt \cup OutOnt \cup IntOnt$ is used to specify the formulae of Π. Furthermore, only a subset of the temporal variables from Π is used in Φ, more specifically, the set of temporal variables from

$\{body(\varphi_i)|\varphi_i \in \Pi\} \cup \{head(\varphi_i)|\varphi_i \in \Pi\ AND\ head(\varphi_i)$ is expressed over $InteractOnt\}$.

11.7 Verification of Specifications of Multi-Agent Systems in TTL

In this Section two verification techniques of specifications of multi-agent systems are considered. In Section 11.7.1 a verification approach of TTL specifications by model checking is discussed. Checking of TTL properties on a limited set of traces obtained empirically or by simulation is considered in Section 11.7.2.

11.7.1 Verification of interlevel relations in TTL specifications by model checking

The dynamic properties of a component of a higher aggregation level can be logically related by an interlevel relation to dynamic properties of components of an adjacent lower aggregation level. This interlevel relation takes the form that a number of properties of the lower level logically entail the properties of the higher level component.

Identifying interlevel relations is usually achieved by applying informal or semi-formal early requirements engineering techniques; e.g., i^* [120] and SADT [300]. To formally prove that the identified interlevel relations are indeed correct, model checking techniques [100, 310] may be of use. The idea is that the lower level properties in an interlevel relation are used as a system specification, whereas the higher level properties are checked for this system specification. However, model checking techniques are only suitable for systems specified as finite-state concurrent systems. To apply model checking techniques it is needed to transform an original behavioural specification of the lower aggregation level into a model based on a finite state transition system. To obtain this, as a first step a behavioural description for the lower aggregation level is replaced by one in executable temporal format using the procedure described in Section 11.6.2. After that, using an automated procedure an executable temporal specification is translated into a general finite state transition system format that consists of standard transition rules. Such a representation can be easily translated into an input format of one of the existing model checkers. To translate an executable specification into the finite state transition system format, for each rule from the executable specification the corresponding transition rule should be created. For translation the atom *present_time* is used, which is evaluated to true only in a state for the current time point. For example, consider the translation of the memory state creation and persistence rules given in Table 11.3. The translation of other rules is provided in [398].

Table 11.3 Translation of the memory state creation and persistence rules into the corresponding finite state transition rules

Rule from the executable specification	Corresponding transition rules
Memory state creation rule	
$\forall t'\ holds(state(\gamma,t'),p) \Rightarrow$	
$holds(state(\gamma,t'),memory(t',p))$	$present_time(t) \wedge p \longrightarrow memory(t,p)$
Memory persistence rule	
$\forall t''\, holds(state(\gamma,t''),memory(t',p)) \Rightarrow$	
$holds(state(\gamma,t''+1),memory(t',p))$	$memory(t,p) \longrightarrow memory(t,p)$

The executable properties obtained in Section 11.6.2 for the property $DP7(A1,A2)$ from the running example were translated into the transition rules as follows:

$$present_time(t) \wedge communicated_from_to(A,B,request,IC) \longrightarrow$$
$$memory(t,communicated_from_to(A,B,request,IC))$$

$$present_time(t) \wedge observation_result_to_for(B,IC) \longrightarrow$$
$$memory(t,observation_result_to_for(B,IC)) \wedge$$
$$stimulus_reaction(observation_result_to_for(B,IC))$$

$$memory(t, communicated_from_to(A, B, request, IC)) \longrightarrow$$
$$memory(t, communicated_from_to(A, B, request, IC))$$

$$memory(t, observed(observation_result_to_for(B, IC)) \longrightarrow$$
$$memory(t, observed(observation_result_to_for(B, IC))$$

$$present_time(t) \wedge$$
$$\exists u2 \leq t\ memory(u2, communicated(request_from_to_for(A, B, IC))) \longrightarrow$$
$$conditional_preparation_for($$
$$output(communicated_from_to(B, A, inform, IC)))$$

$$present_time(t) \wedge$$
$$conditional_preparation_for($$
$$output(communicated_from_to(B, A, inform, IC))) \wedge$$
$$memory(t, observed(observation_result_to_for(B, IC))) \wedge$$
$$stimulus_reaction(observed(observation_result_to_for(B, IC))) \longrightarrow$$
$$preparation_for(output(t + c, communicated_from_to(B, A, inform, IC)))$$

$$present_time(t) \wedge$$
$$stimulus_reaction(observed(observation_result_to_for(B, IC))) \wedge$$
$$not(preparation_for($$
$$output(t + c, communicated_from_to(B, A, inform, IC))))$$
$$\longrightarrow stimulus_reaction(observed(observation_result_to_for(B, IC)))$$

$$preparation_for(output(t + c, communicated_from_to(B, A, inform, IC))) \wedge$$
$$not(output(communicated_from_to(B, A, inform, IC))) \longrightarrow$$
$$preparation_for(output(t + c, communicated_from_to(B, A, inform, IC)))$$

$$preparation_for(output(t + c, communicated_from_to(B, A, inform, IC))) \wedge$$
$$present_time(t + c - 1) \longrightarrow$$
$$output(communicated_from_to(B, A, inform, IC))$$

The obtained general representation for a finite state transition system was used further as a model for the model checker SMV [310]. SMV was used to perform the automatic verification of relationships between dynamic properties of components of different aggregation levels. For this purpose a procedure was developed for translating the general description of a transition system into the input format of the SMV model checking tool. For the description of the translation procedure and the complete SMV specification for the considered example we refer to [398].

One of the possible dynamic properties of the higher aggregation level that can be verified against the generated SMV specification is formulated and formalized in CTL as follows:

GP (Concluding effectiveness): If at some point in time environmental component E generates all the correct relevant information, then later agent C will receive a correct conclusion.

$\mathbf{AG}\ (E_output_observed_provide_result_from_to_E_A_info\ \&$
$E_output_observed_provide_result_from_to_E_B_info$

$\rightarrow \mathbf{AF}\ \ input_C_communicated_send_from_to_A_C_info),$

where **A** is a path quantifier defined in CTL, meaning "for all computational paths", **G** and **F** are temporal quantifiers that correspond to "globally" and "eventually" respectively.

The automatic verification by the SMV model checker confirmed that this property holds with respect to the considered model of the multi-agent system as specified at the lower level.

11.7.2 Verification of Traces in TTL

This section introduces a technique for verification of TTL specifications. Using this technique TTL properties are checked upon a limited set of traces. On the one hand, this set can be obtained by performing simulation of particular scenarios based on the TTL specification. In this case only a relevant subset of all possible traces is considered for the analysis. On the other hand, a set of traces can be obtained by formalising empirical data. Then, both verification of TTL properties on these traces and validation of TTL specifications by empirical data can be performed. For this type of verification a dedicated algorithm and the software tool TTL Checker have been developed [80] (see Figure 11.5) [1] .

As an input for this analysis technique either a simulation or a formalized empirical trace(s) is/are provided. A trace is represented by a finite number of state atoms, changing their values over time a finite number of times, i.e., complies with the finite variability property defined in Section 11.3. The verification algorithm is a backtracking algorithm that systematically considers all possible instantiations of variables in the TTL formula under verification. However, not for all quantified variables in the formula the same backtracking procedure is used. Backtracking over variables occurring in *holds* predicates is replaced by backtracking over values occurring in the corresponding *holds* atoms in traces under consideration. Since there are a finite number of such state atoms in the traces, iterating over them often will be more efficient than iterating over the whole range of the variables occurring in the holds atoms.

[1] The TTL Checker tool can be downloaded at http://www.few.vu.nl/ wai/TTL/

As time plays an important role in TTL-formulae, special attention is given to continuous and discrete time range variables. Because of the finite variability property, it is possible to partition the time range into a minimum set of intervals within which all atoms occurring in the property are constant in all traces. Quantification over continuous or discrete time variables is replaced by quantification over this finite set of time intervals.

In order to increase the efficiency of verification, the TTL formula that needs to be checked is compiled into a Prolog clause. Compilation is obtained by mapping conjunctions, disjunctions and negations of TTL formulae to their Prolog equivalents, and by transforming universal quantification into existential quantification. Thereafter, if this Prolog clause succeeds, the corresponding TTL formula holds with respect to all traces under consideration.

The complexity of the algorithm has an upper bound in the order of the product of the sizes of the ranges of all quantified variables. However, if a variable occurs in a holds predicate, the contribution of that variable is no longer its range size, but the number of times that the holds atom pattern occurs (with different instantiations) in trace(s) under consideration. The contribution of an isolated time variable is the number of time intervals into which the traces under consideration are divided.

The specific optimisations discussed above make it possible to check realistic dynamic properties with reasonable performance. To illustrate this technique the specification of the co-operative information gathering multi-agent system from Section 11.4 was instantiated for the case, when agents *A* and *B* collect and combine information about orthogonal projections of a three-dimensional shape: *A* collects information about the side view and *B* collects information about the bottom view. For example, if *A* observes a triangle and *B* observes a circle, then the shape is a cone. Using the simulation software environment LeadsTo [81] a number of simulation traces were generated and loaded into the TTL Checker. Then, a number of TTL properties were checked automatically on the traces, among which:

P1 (Successfulness of the cone determination)

$\forall \gamma\ \exists t\ \exists V : COMPONENT\ holds(state(\gamma, t, input(C)),$
$communicated_from_to(V, C, inform, conclusion(cone)))$

P2 (Successfulness of the projection acquisition for a cone)

$\forall \gamma\ \forall t1, t2\ holds(state(\gamma, t1, input(A)),$
$observation_result_to_for(A, side_view(triangle)))\&$
$holds(state(\gamma, t2, input(B)), observation_result_to_for(B, bottom_view(circle)))$

Checking the property P2 took 0.46 sec. on a regular PC. With the increase of the number of traces with similar complexity as the first one, the verification time grows linearly: for 3 traces - 1.3 sec., for 5 traces - 2.25 sec. However, the verification time is polynomial in the number of isolated time range variables occurring in the formula under verification.

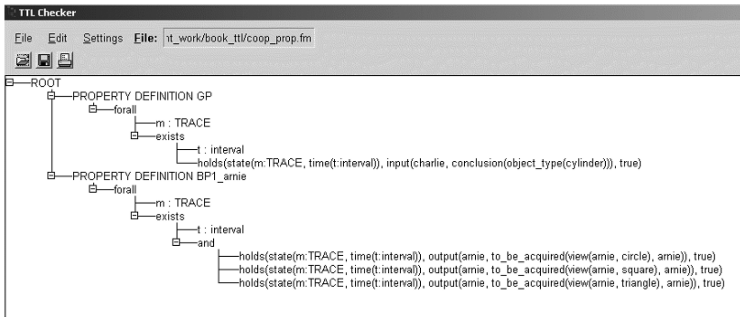

Fig. 11.5 Screenshot from the TTL Checker Tool

11.8 Conclusions

This chapter presents the predicate logical Temporal Trace Language (TTL) for formal specification and analysis of dynamic properties. TTL allows the possibility of explicit reference to time points and time durations, which enables modelling of the dynamics of continuous real-time phenomena. Although the language has a logical foundation, it supports the specification of both qualitative and quantitative aspects of a system, and subsumes specification languages based on differential equations.

Sometimes dynamical systems that combine both quantitative and qualitative aspects are called hybrid systems [133]. In contrast to many studies on hybrid systems in computer science, in which a state of a system is described by assignment of values to variables, in the proposed approach a state of a system is defined by (composite) objects using a rich ontological basis (i.e., typed constants, variables, functions and predicates). This provides better possibilities for conceptualizing and formalizing different kinds of systems (including those from natural domains). Furthermore, by applying numerical approximation methods for continuous behaviour of a system, variables in a generated model become discrete and are treated in the same manner as finite-state transition system variables. Therefore, so-called control points [297], at which values of continuous variables are checked and changes in a system's functioning mode are made, are not needed.

Furthermore, more specialised languages can be defined as a sublanguage of TTL. For simulation, the executable language LEADSTO has been developed [81]. For verification, decidable fragments of predicate logics and specialized languages with limited expressivity can be defined as sublanguages of TTL. TTL has similarities (as well as important conceptual distinctions) with (from) situation and event calculi. A proper subclass of TTL formulae can be directly translated into formulae of temporal logics (e.g., LTL and CTL).

In this chapter an automatically supported technique for verifying TTL properties on a limited set of simulation or empirical traces was described. Furthermore, it was shown how model checking techniques can be used for verification of TTL

specifications. To enable model checking, a model should be provided in the form of a finite state transition system. In this chapter it was shown how a TTL specification that comprises formulae in the executable normal form can be automatically translated into a finite state transition system. Using such an approach relations between dynamic properties of adjacent aggregation levels of a multi-agent system can be checked automatically, as also demonstrated in this chapter. The proposed approach has similarities with compositional reasoning and verification techniques [238] in the way how it handles complex dynamics of a system. Compositional reasoning approaches developed in the area of software engineering are based on one common idea that the analysis of global properties of a software system can be reduced to the analysis of local properties of system components. More specifically, the problem of satisfaction of global properties of a complex software system can be reduced to two (easier) problems: (i) identifying and justifying relations between global properties of the system and local properties of its components (parts); (ii) verifying local properties of system components with respect to components specifications.

In [338] formal methods for the analysis of hardware specifications expressed in the language PSL (an extension of the standard temporal logics LTL and CTL), are described. By means of the suggested property assurance technique supported by a tool, different global system properties (e.g., consistency) can be verified on specifications and in such a way the correctness of specifications can be established. The verification is based on bounded model checking techniques. Besides the specification language, an essential difference between this analysis method and the approach described in this chapter is that the latter provides means for the multi-level (or compositional) representation and verification of properties in specifications. This allows system modelling at a necessary level of abstraction and the reduction of the complexity of verification of system dynamics.

Similar differences can be identified in comparison with the approach proposed in [186]. This approach allows semi-automatic formalization of informal graphical specifications of multi-agent systems with the subsequent verification of dynamic properties using model checking techniques. Formalized specifications comprise descriptions of classes that describe components of a multi-agent system and relations between them, constraints over these components, assertions and possibilities. Although the first-order temporal logic that is used for formalizing these specifications is expressive enough to define complex temporal relations, it is does not provide the complete expressivity allowed by TTL (e.g., arithmetical operations, references to multiple traces in the same formula). Furthermore, although such specifications can be built and analyzed in parts, the idea of compositional verification, central in our approach, is not elaborated in this approach.

Compositional verification may be used for analysis of dynamics of large socio-technical systems (e.g., in the area of incident management). Such systems are characterized by a large complexity of internal dynamics of and interaction among diverse types of agents, including human and artificial intelligent agents (e.g., ambient devices). It is expected that in the future the complexity of such systems will increase considerably with a further development and implementation of ambient

intelligence technologies. Formal analysis of such systems presents a big conceptual and computational challenge for existing verification tools in the area of multi-agent systems. To enable effective and efficient analysis of systems of such type, new methods based on appropriate (dynamic) abstraction mechanisms need to be developed. For this the idea of compositional verification may serve as the starting point. Further, findings from the area of nonlinear system analysis, control theory and complex systems in general could be used.

Finally, TTL and the related analysis techniques proved their value in a number of research projects in such disciplines as artificial intelligence, cognitive science, biology, and social science. In particular, the analysis of continuous models (i.e., based on differential equations) is illustrated by the case study on trace conditioning considered in [83]. In [79] TTL is used for modelling and analysis of adaptive agent behaviour specified by complex temporal relations. The use of arithmetical operations in TTL to perform statistical analysis is illustrated by a case study from the criminology [78]. More examples of applications of TTL are described in [80].