On-the-fly Automata Construction for Dynamic Linear Time Temporal Logic

Laura Giordano[1], Alberto Martelli[2]

[1]Dipartimento di Informatica, Università del Piemonte Orientale, Alessandria

[2]Dipartimento di Informatica, Università di Torino, Torino

Summary of the paper

- We present a tableau-based algorithm for obtaining a **Büchi automaton** from a formula in **Dynamic Linear Time Temporal Logic**.
- Dynamic Linear Time Temporal Logic (DLTL) (Henriksen, Thiagarajan) extends LTL by indexing the until operator with regular programs.
- The construction of the states of the automaton is similar to the standard on-the-fly construction for LTL (Gerth, Peled, Vardi, Wolper), but a different technique must be used to verify the fulfillment of until formulas.

Introduction 1

- The problem of constructing automata from Linear-Time temporal (LTL) formulas has been deeply studied. The interest on this problem comes from the wide use of temporal logic for the verification of properties of concurrent systems. The standard approach to LTL model checking consists of translating the negation of a given LTL formula (property) into a Büchi automaton, and checking the product of the property automaton and the model for language emptiness.
- A tableau-based algorithm for efficiently constructing a Büchi automaton has been presented by Gerth, Peled, Vardi, Wolper. This algorithm allows to build the graph “on the fly” and in most cases builds quite small automata, although the problem is intrinsically exponential.

Introduction 2

- In this paper we present an algorithm for constructing a Büchi automaton from a DLTL formula making use of a tableau-based construction.
- The validity of a formula α can be verified by constructing the Büchi automaton $\mathcal{B}_{\neg\alpha}$ for $\neg\alpha$: if the language accepted by $\mathcal{B}_{\neg\alpha}$ is empty, then α is valid, whereas any infinite word accepted by $\mathcal{B}_{\neg\alpha}$ provides a counterexample to the validity of α.

Motivations

- Dynamic Linear Time Temporal Logic (DLTL) extends LTL by indexing the until operator with programs in Propositional Dynamic Logic, and has been shown to be strictly more expressive than LTL. The satisfiability problem for DLTL can be solved in exponential time, by reducing it to the emptiness problem for Büchi automata.
- In other papers (with C. Schwind) we have developed an action theory based on DLTL and of its product version, and we have shown how to use it to model multi-agent systems and to verify their properties, in particular by using model checking techniques.
- The construction given by Henriksen and Thiagarajan is highly inefficient since it requires to build an automaton with an exponential number of states, most of which will not be reachable from the initial state.

DLTL (Henriksen, Thiagarajan)

Σ be a finite non-empty alphabet of **actions**.
Σ^ω the set of infinite words on Σ.

$Prg(\Sigma)$ the set of **programs** (regular expressions)

$$Prg(\Sigma) ::= a \mid \pi_1 + \pi_2 \mid \pi_1;\pi_2 \mid \pi^*, \text{ where } a \in \Sigma$$

$[[\pi]]$ represents the set of executions of π

The set of formulas of DLTL(Σ):

$$\text{DLTL}(\Sigma) ::= p \mid \neg\alpha \mid \alpha \vee \beta \mid \alpha \mathcal{U}^\pi \beta$$

where $p \in \mathcal{P}$ are atomic propositions

DLTL

A model of DLTL(Σ) is a pair $M = (\sigma, V)$ where $\sigma \in \Sigma^\omega$ and $V : prf(\sigma) \to 2^{\mathcal{P}}$ is a valuation function.

Given a model $M = (\sigma, V)$, and a prefix τ of σ, we can define the satisfiability of a formula at τ in M. In particular:

$M, \tau \models \alpha \mathcal{U}^\pi \beta$ iff there exists $\tau' \in [[\pi]]$ such that

- $\tau\tau' \in prf(\sigma)$
- $M, \tau\tau' \models \beta$
- $M, \tau\tau'' \models \alpha$ for every prefix τ'' of τ'

DLTL

We can define derived modalities

- $\langle \pi \rangle \alpha \equiv \top \mathcal{U}^{\pi} \alpha$
- $[\pi] \alpha \equiv \neg \langle \pi \rangle \neg \alpha$
- $\bigcirc \alpha \equiv \bigvee_{a \in \Sigma} \langle a \rangle \alpha$ (next)
- $\Diamond \alpha \equiv \top \mathcal{U}^{\Sigma^*} \alpha$
- $\Box \alpha \equiv \neg \Diamond \neg \alpha$

Automaton construction

A *Büchi automaton* over Σ is a tuple $\mathcal{B} = (Q, \rightarrow, Q_{in}, F)$ where:

- Q is a finite nonempty set of states;
- $\rightarrow \subseteq Q \times \Sigma \times Q$ is a transition relation;
- $Q_{in} \subseteq Q$ is the set of initial states;
- $F \subseteq Q$ is a set of accepting states.

Let $\sigma \in \Sigma^{\omega}$. Then a run of $\mathcal{B}$ over σ is a map $\rho : prf(\sigma) \rightarrow Q$ such that:

- $\rho(\varepsilon) \in Q_{in}$
- $\rho(\tau) \xrightarrow{a} \rho(\tau a)$ for each $\tau a \in prf(\sigma)$

A run is *accepting* iff if contains infinitely many times an accepting state.
$\mathcal{L}(\mathcal{B})$ is the language of ω-words accepted by $\mathcal{B}$.

Automaton construction

In our construction, we make use of an equivalent formulation of DLTL formulas in which “until” formulas are indexed with finite automata rather than regular expressions. Thus we have $\alpha \mathcal{U}^{\mathcal{A}} \beta$ instead of $\alpha \mathcal{U}^{\pi} \beta$, where $\mathcal{L}(\mathcal{A}) = [[\pi]]$.
The size of the automaton $\mathcal{A}$ can be linear in the size of π.

We will make use of the following notation for automata.
Let $\mathcal{A} = (Q, \delta, Q_F)$ be an ϵ-free nondeterministic finite automaton over the alphabet Σ without an initial state. Given a state $q \in Q$, we denote with $\mathcal{A}(q)$ an automaton $\mathcal{A}$ with initial state q.

Automaton construction

The main procedure to construct the Büchi automaton for a formula ϕ builds a graph $\mathcal{G}(\phi)$ whose nodes are labelled by sets of formulas. Nodes and edges of the graph correspond to the states and the transitions of the Büchi automaton. The procedure makes use of an auxiliary tableau-based function.

The tableau function makes use of *signed formulas*, i.e. formulas prefixed with the symbol **T** or **F**. This function
- takes as input a set of formulas,
- adds to it formula **T** $\bigvee_{a \in \Sigma} \langle a \rangle \top$, and
- returns a set of sets of formulas obtained by expanding the input set according to a set of tableau rules, formulated as follows:

Tableau rules

$$\mathbf{T}(\alpha \wedge \beta) \Rightarrow \mathbf{T}\alpha, \mathbf{T}\beta$$
$$\mathbf{F}(\alpha \vee \beta) \Rightarrow \mathbf{F}\alpha, \mathbf{F}\beta$$
$$\mathbf{F}(\alpha \wedge \beta) \Rightarrow \mathbf{F}\alpha | \mathbf{F}\beta$$
$$\mathbf{T}(\alpha \vee \beta) \Rightarrow \mathbf{T}\alpha | \mathbf{T}\beta$$
$$\mathbf{T}\neg\alpha \Rightarrow \mathbf{F}\alpha$$
$$\mathbf{F}\neg\alpha \Rightarrow \mathbf{T}\alpha$$
$$\mathbf{T}\alpha\mathcal{U}^{\mathcal{A}(q)}\beta \Rightarrow$$
$$\mathbf{T}(\beta \vee (\alpha \wedge \bigvee_{a\in\Sigma}\langle a\rangle \bigvee_{q'\in\delta(q,a)} \alpha\mathcal{U}^{\mathcal{A}(q')}\beta))(q \text{ is a final state})$$
$$\mathbf{T}\alpha\mathcal{U}^{\mathcal{A}(q)}\beta \Rightarrow$$
$$\mathbf{T}(\alpha \wedge \bigvee_{a\in\Sigma}\langle a\rangle \bigvee_{q'\in\delta(q,a)} \alpha\mathcal{U}^{\mathcal{A}(q')}\beta)(q \text{ is not a final state})$$
$$\mathbf{F}\alpha\mathcal{U}^{\mathcal{A}(q)}\beta \Rightarrow$$
$$\mathbf{F}(\beta \vee (\alpha \wedge \bigvee_{a\in\Sigma}\langle a\rangle \bigvee_{q'\in\delta(q,a)} \alpha\mathcal{U}^{\mathcal{A}(q')}\beta))(q \text{ is a final state})$$
$$\mathbf{F}\alpha\mathcal{U}^{\mathcal{A}(q)}\beta \Rightarrow$$
$$\mathbf{F}(\alpha \wedge \bigvee_{a\in\Sigma}\langle a\rangle \bigvee_{q'\in\delta(q,a)} \alpha\mathcal{U}^{\mathcal{A}(q')}\beta)(q \text{ is not a final state})$$

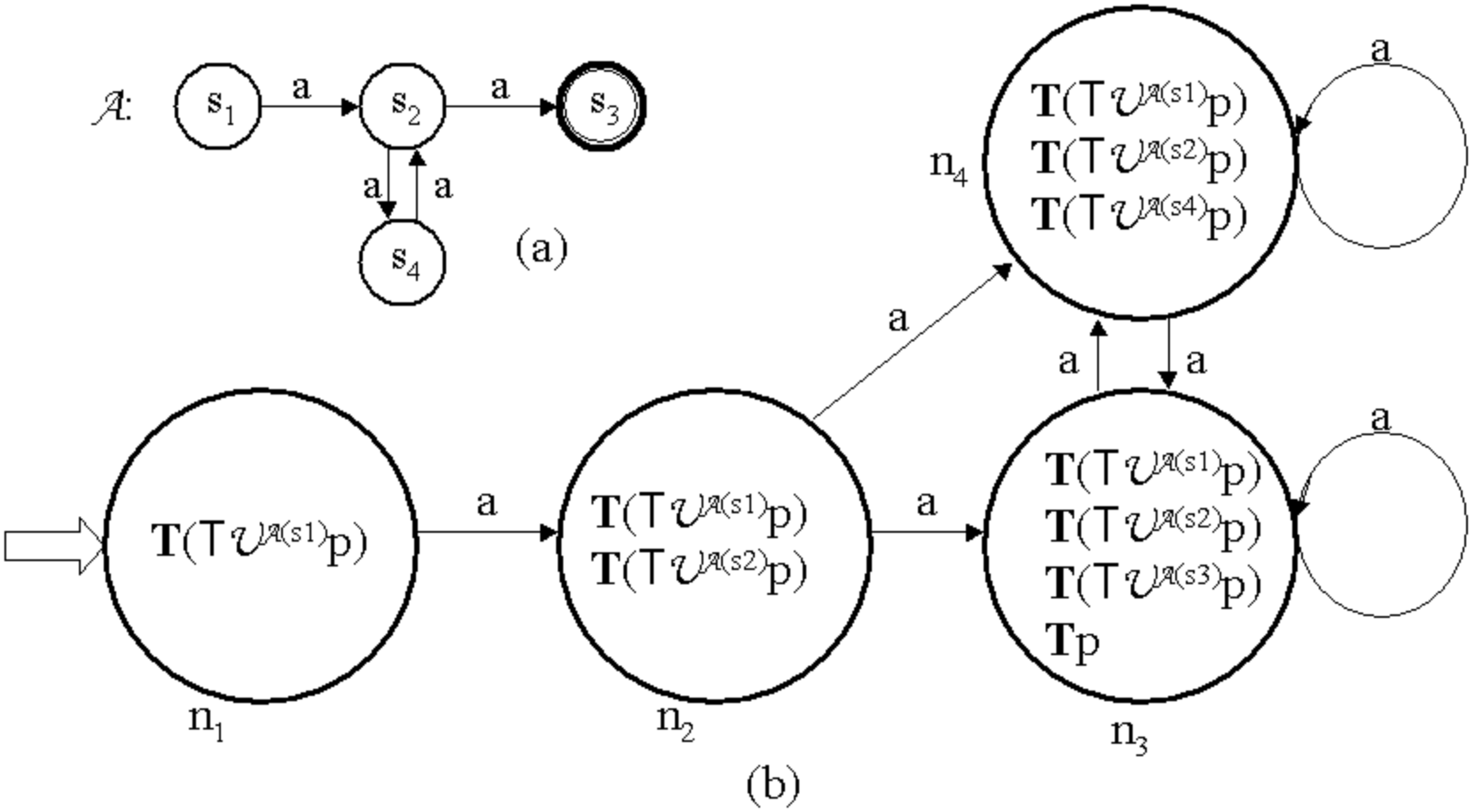
Graph construction

The tableau rules do not expand formulas of the kind $\langle a \rangle \alpha$. Since the operator $\langle a \rangle$ is a *next state* operator, expanding this kind of formulas from a node n means to create a new node containing α connected to n through an edge labelled with a.

Given a node n containing a formula $\mathbf{T}\langle a \rangle \alpha$, then the set of nodes connected to n through an edge labelled a can be obtained by $tableau(\{\mathbf{T}\alpha | \mathbf{T}\langle a \rangle \alpha \in n\} \cup \{\mathbf{F}\alpha | \mathbf{F}\langle a \rangle \alpha \in n\})$.

However this construction does not allow to define correctly accepting conditions for *until* formulas.

graph for $\Box\langle\mathcal{A}(s_1)\rangle p \equiv \mathbf{F}(\top\mathcal{U}^{\mathcal{A}_1(s_0)}\neg(\top\mathcal{U}^{\mathcal{A}(s_1)}p))$



Problem

We cannot find an accepting set of nodes in this graph: Every node of this graph contains a formula $\mathbf{T}(\top \mathcal{U}^{\mathcal{A}(s_1)} p)$, and the only node which might fulfill the until formulas is node n_3, since it contains $\mathbf{T}(\top \mathcal{U}^{\mathcal{A}(s_3)} p)$, with s_3 final, and $\mathbf{T}p$.

However it is easy to see that not all infinite paths through n_3 will be accepting. For instance, in the path $n_1, n_2, n_3, n_4, n_3, n_4, n_3, n_4, \ldots$ no occurrence of n_3 fulfills the formula $\mathbf{T}(\top \mathcal{U}^{\mathcal{A}(s_1)} p)$ in n_2, since the distance in this path between node n_2 and any occurrence of n_3 is odd, while all strings in $\mathcal{L}(\mathcal{A}(s_1))$ have even length.

Gerth, Peled, Vardi, Wolper use *generalized Büchi automata*, but this does not work here.

Construction of the graph

We adopt a different solution, derived from Henriksen and Thiagarajan, where some of the nodes will be duplicated to avoid the above problem.

Each node of the graph is a triple $(\mathcal{F}, x, f)$, where $\mathcal{F}$ is an expanded set of formulas, $x \in \{0, 1\}$, and $f \in \{\downarrow, \checkmark\}$.

All true until formulas have a label 0 or 1, i.e. they have the form $\mathbf{T}^l \alpha \mathcal{U}^{\mathcal{A}(q)} \beta$ where $l \in \{0, 1\}$.

For each node $(\mathcal{F}, x, f)$, the label of an until formula in $\mathcal{F}$ will be assigned as follows. If it is derived from an until formula in a predecessor node, then its label is the same as that of the until formula it derives from. Otherwise, if the formula is new, it is given the label $1 - x$.

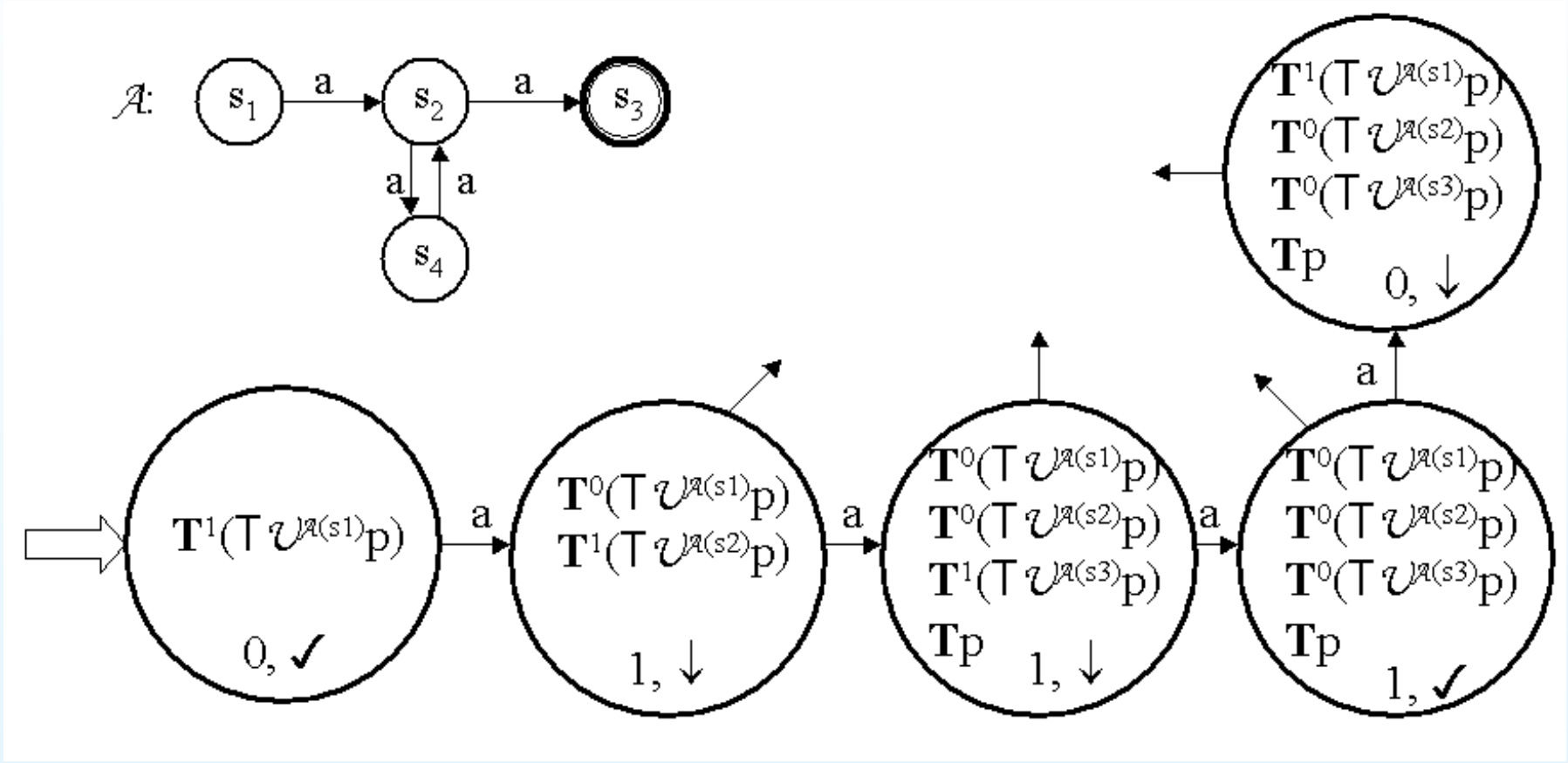
Construction of the graph

The x and f values of the nodes of the graph are assigned as follows:

- the initial node contains $(0, \checkmark)$
- if a node n contains $(x, \checkmark)$ then its successors contain $(1 - x, \downarrow)$,
- if a node n contains $(x, \downarrow)$ then its successors contain
  - $(x, \checkmark)$ if n does not contain any until formula with label x
  - $(x, \downarrow)$ otherwise

The set of **accepting states** consists of all states whose associated node contains $f = \checkmark$.

Correct graph for $\Box\langle\mathcal{A}(s_1)\rangle p$



Correctness of the procedure

Let ρ be a run of $\mathcal{B}(\phi)$.
According to the construction of the graph, the x and f values of the nodes of ρ will be as follows:
$(0, \checkmark), (1, \downarrow), \ldots, (1, \downarrow), (1, \checkmark), (0, \downarrow), \ldots, (0, \downarrow), (0, \checkmark), \cdots$

Let us call *0-sequences* or *1-sequences* the sequences of nodes of ρ with $x = 0$ or $x = 1$ respectively. If ρ is an *accepting run*, these (finite) sequences will alternate infinitely many times.

It is possible to see that every until formula contained in a node of a 0-sequence must be fulfilled within the end of the next 1-sequence, and vice versa.

Correctness of the procedure

Let ρ be a (non necessarily accepting) run, and let s be a state of ρ containing an until formula ϕ with label l. Then, either

1. all states following s in ρ contain an until formula derived from ϕ with label l, or
2. there is a state following s where ϕ is fulfilled

Condition 2 holds iff ρ is an accepting run.

Theorem. Let $M = (\sigma, V)$ and $M, \varepsilon \models \phi$. Then $\sigma \in \mathcal{L}(\mathcal{B}(\phi))$.

Theorem. Let $\sigma \in \mathcal{L}(\mathcal{B}(\phi))$. Then there is a model $M = (\sigma, V)$ such that $M, \varepsilon \models \phi$.

As usual, validity of a formula ϕ in DLTL can be checked by checking emptiness of $\mathcal{L}(\mathcal{B}(\neg\phi))$.

Action theory

Action laws

$$\Box(\alpha \rightarrow [a]\beta)$$

Causal laws

$$\Box([a]\alpha \rightarrow [a]\beta)$$

Persistency is modelled by a completion construction (Reiter) of action and causal laws.

Specifying protocols (social approach)

In a social approach communicative actions affect the "social state" of the system, rather than the internal states of the agents. The social state records the social facts, like the permissions and the commitments of the agents, which are created and modified in the interactions among them.

This protocol does not require the rigid specification of all the allowed action sequences, e.g. by means of finite state diagrams.

The action theory can provide a high level specification of the protocol.

Permissions

Permissions are represented by *Precondition laws* of the form: $\alpha \rightarrow [a]\bot$,
meaning that the execution of an action a is not possible if α holds (i.e. there is no resulting state following the execution of a if α holds).

$$\Box(\neg paid \rightarrow [sendReceiptMrCt]\bot)$$

$Perm_i$ (permissions of agent i): the set of all the precondition laws of the protocol pertaining to the actions of which agent i is the sender.

Commitments

Commitments are special fluents

- **base-level commitments**: $C(ag_1, ag_2, action)$ (agent ag_1 is committed to agent ag_2 to execute the $action$)
- **conditional commitments**: $CC(ag_1, ag_2, p, action)$ (agent ag_1 is committed to agent ag_2 to execute $action$, if the condition p is brought about)

Specifying protocols

Action laws from the point of view of the merchant (the same for the customer):

$$\Box([sendQuoteMrCt](CC(mr, ct, accepted, sendGoodsMrCt) \wedge CC(mr, ct, paid, sendReceipt)))$$
$$\Box(requested \rightarrow [sendQuoteMrCt]\neg requested)$$
$$\Box([sendPaymentCtMr]paid)$$
$$\ldots$$

Rules for commitments

$$\Box([a]\neg C(i,j,a))$$
$$\Box([a]\neg CC(i,j,p,a))$$
$$\Box((CC(i,j,p,a) \wedge \bigcirc p) \rightarrow \bigcirc(C(i,j,a) \wedge \neg CC(i,j,p,a)))$$

where we assume that the action a is shared by the agents i (debtor) and j (creditor)

Fulfillment of commitments

An agent i satisfies its commitments when, in all the runs of the system, for all the possible commitments $C(i, j, a)$, the formula

$$\Box(C(i, j, a) \rightarrow \Diamond\langle a\rangle\top)$$

holds: when an agent is committed to execute action a, then it must eventually execute a.
We denote with Com_i the set of all the formulas describing the satisfaction of the commitments of agent i.

Specifying a protocol

A protocol can be described as:

- A **domain description** D consisting of the action and causal laws of the protocol (suitably completed to cope with persistency).
- **Precondition laws** $Perm_i$ (permissions) for each agent i.
- **Commitment constraints** Com_i for each agent i.

Reasoning about protocols

Given a **domain description** D we can build a Büchi automaton whose runs are all possible executions of the actions (model).

Given the formula the formula:

$$D \wedge \bigwedge_j (Perm_j \wedge Com_j)$$

we can build a Büchi automaton whose runs are all correct executions of the protocol.

Verifying the compliance of agents to the protocol

We are given a history $\tau = a_1, \ldots, a_n$ of the communicative actions executed by the agents, and we want to verify that the history τ is the prefix of a run of the protocol. This means that

$$(D \wedge \bigwedge_i (Perm_i \wedge Com_i)) \wedge < a_1; a_2; \ldots; a_n > \top$$

must be satisfiable. In fact, the above formula is satisfiable if it is possible to find a run of the protocol starting with the action sequence $a_1, \ldots, a_n$.
This verification is carried out at runtime.
We can carry out the construction on-the-fly whenever a new action is executed by some agent, until either the protocol is completed or the construction cannot go on, in case of violation of the protocol.

Specifying rigid protocols

A rigid protocol can be formalized as a regular program, or a finite automaton.

Contract Net:

$$cfp(M, all); (refuse(i, M)+ \\ (propose(i, M); (reject(M, i)+ \\ accept(M, i); inform(i, M, Done(i, task)))))$$

In this case the domain description specifies the semantics of the actions.

Verifying rigid protocols

We can verify the correctness of a protocol, specified by a regular program $Prog$, with respect to the semantics of the actions.
This can be formalized as the validity check of the formula

$$(D \wedge \langle Prog \rangle \top) \rightarrow \bigwedge_i (Perm_i \wedge Com_i)$$