Web Service Composition using DLTL

Laura Giordano[1] and Alberto Martelli[2]

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

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

DLTL

- The action theory is based on Dynamic Linear Time Temporal Logic ($DLTL$), an extension of LTL (the propositional linear time temporal logic).
- $DLTL$ extends LTL by strengthening the *until* operator by indexing it with the regular programs of dynamic logic. It is, essentially, a dynamic logic equipped with a linear time semantics.
- $DLTL$ has an exponential time decision procedure based on Büchi automata.

Action theory

For each (communicative) action a we can define
Action laws

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

Precondition laws

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

Causal laws

$$\Box((\alpha \wedge \bigcirc\beta) \rightarrow \bigcirc\gamma)$$

Persistency is modeled by a completion construction.

Deterministic actions

- We assume **actions** to be **deterministic**.
- Nondeterminism in protocols is represented by the possibility of executing different actions in a given state.

Specifying web services

We adopt a social approach to protocol specification, where communicative actions affect the "social state" of the system, rather than the internal states of the participating 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.

Incomplete knowledge

Each agent participating in a conversation is only aware of some of the effects of communicative actions in the conversation (namely those it is involved in as a sender or as a receiver).

We use a knowledge operator $\mathcal{K}$ to describe the knowledge shared by groups of agents.

An *epistemic state* is a *complete and consistent set of epistemic fluent literals*, and it provides a *three-valued* interpretation in which each literal l is *true* when $\mathcal{K}l$ holds, *false* when $\mathcal{K}\neg l$ holds, and *undefined* when both $\neg\mathcal{K}l$ and $\neg\mathcal{K}\neg l$ hold.

Example: Purchase service

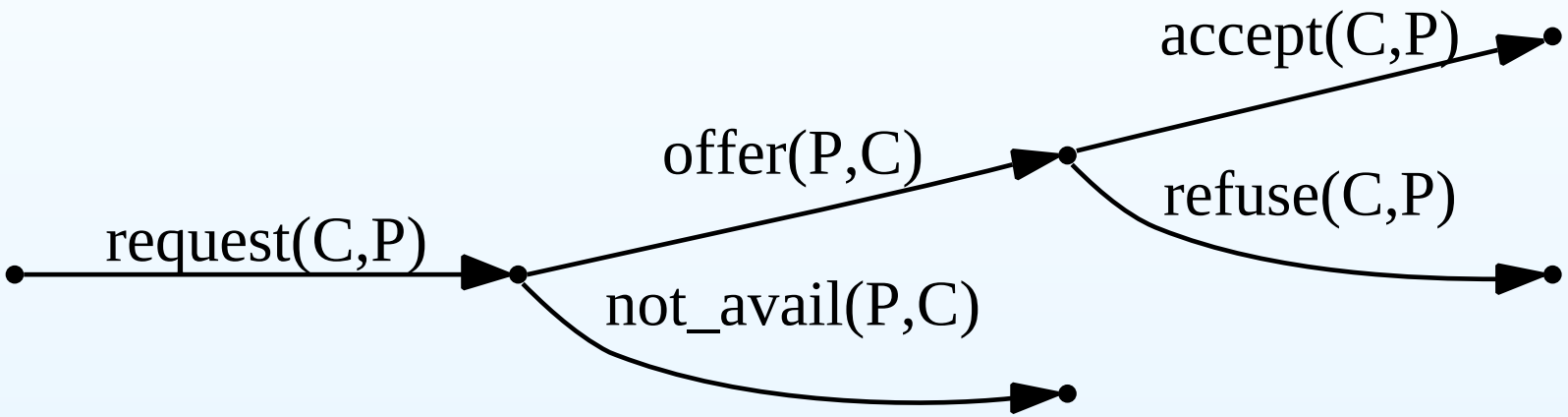
Two **roles**: P, the producer, and C, the customer.

Actions:
$request(C, P)$ (the customer sends a request for a product), $offer(P, C)$ and $not\_avail(P, C)$ (the producer sends an offer or says that the product is not available), $accept(C, P)$ and $refuse(C, P)$ (the customer accepts or refuses the offer), $begin(C, P)$ and $end(C, P)$ (to start and finish the protocol).

Fluents:
$requested$, the product has been requested, $offered$, the product is available and an offer has been sent ($\neg offered$ means that the product is not available), $accepted$, the offer has been accepted, Pu, the protocol is being executed.
Commitments are special fluents.

Purchase service

C: customer
P: producer



Action laws

Some action laws

$\Box[begin(C,P)]CC(Pu,P,C,\mathcal{K}(requested),\mathcal{K}^w(offered))$
$\Box[offer(P,C)]\mathcal{K}(offered)$
$\Box[not\_avail(P,C)]\mathcal{K}(\neg offered)$
$\Box[accept(C,P)]\mathcal{K}(accepted)$
$\Box[refuse(C,P)]\mathcal{K}(\neg accepted)$

$\mathcal{K}^w(f)$ is a shorthand of the formula $(\mathcal{K}(f) \vee \mathcal{K}(\neg f))$

In the initial state, all fluents are undefined.

Permissions

The permissions to execute communicative actions in each state are represented by *precondition laws*. For instance,

$$\Box((\neg\mathcal{K}(Pu) \vee \neg\mathcal{K}(offered) \vee \mathcal{K}^w(accepted)) \rightarrow [accept(C,P)]\bot)$$

an offer cannot be accepted if

- Pu does not hold or
- an offer has not been made or
- the truth value of $accepted$ is known (the customer has already accepted or refused an offer).

Reasoning about protocols using automata

The satisfiability problem for DLTL can be solved in deterministic exponential time, as for LTL, by constructing for each formula α a Büchi automaton $\mathcal{B}_\alpha$ such that there is a one to one correspondence between models of the formula and infinite words accepted by $\mathcal{B}_\alpha$ (we have developed an “on-the-fly” algorithm which extend the one for LTL).

For instance, given a domain description

$$(Comp(\Pi) \wedge Init \wedge \bigwedge_i (Perm_i \wedge Com_i)) \wedge Constr$$

where $Constr$ describes a set of temporal constraints, we can construct the corresponding Büchi automaton, such that all runs accepted by the automaton represent runs of the protocol satisfying the given constraints.

Composing web services

We want to compose a service Sh for shipping goods with the producer service Pu previously described.
The Sh protocol describes the interactions between a customer C and a shipper S.
For simplicity we assume that the protocol of the shipping service has the same actions as the producer service.

The aim of the customer is to extract from the domain description of PS a *plan* allowing it to interact with the two services.

Reasoning about service composition

The domain description D_{PS} of the composed service can be obtained by taking the union of the sets of formulas specifying the two protocols: $D_{PS} = D_{Pu} \cup D_{Sh}$. The runs of the composed service PS give all the interleavings of the runs of the two protocols (parallel composition).

The **goal** of the plan will be specified by means of a set of constraints $Constr$ which will take into account the properties of the composed service.

The specification of the interaction protocol of the composed service is given by $D_{PS} \cup Constr$, from which the customer will extract the plan.

Examples of constraints

The customer cannot request an offer to the shipping service until it has not received an offer from the producer:

$$\Box(\neg \mathcal{K}_C(offered\_Pu) \rightarrow [request\_Sh(C,S)]\bot)$$

The customer must accept both offers or none of them:

$$\Diamond\langle accept\_Pu(C,P)\rangle \leftrightarrow \Diamond\langle accept\_Sh(C,S)\rangle$$

How to get a plan

A **linear plan** can be easily obtained by checking **satisfiability** of the formula $D_{PS} \cup Constr$.
If the formula has a model, this model represents a run of the two services satisfying the given constraints (**plan**).

To do this, we can build the Büchi automaton derived from $D_{PS} \cup Constr$, and search it for an accepting run.

Unfortunately linear plans are not adequate.

Problems

The run

begin_Pu; request_Pu; offer_Pu; accept_Pu; begin_Sh; request_Sh; offer_Sh; accept_Sh; end_Pu; end_Sh

is correct with respect to the above constraints, since both offers are accepted.

However, if the shipping service replies with *not_avail_Sh* instead of *offer_Sh*, there is no correct run with the prefix

begin_Pu; request_Pu; offer_Pu; accept_Pu; begin_Sh; request_Sh; not_avail_Sh

No possibility of replanning to find a different plan.

Dealing with nondeterminism

If a protocol contains a point of choice among different communicative actions, the sender of these actions can choose freely which one to execute.

From the viewpoint of the receiver, that point of choice is a point of nondeterminism to care about. For instance, the customer cannot know whether the service *Pu* will reply with *offer_Pu* or *not_avail_Pu* after receiving the request.

Therefore the customer cannot simply reason on a single choice of action, but he will have to consider all possible choices of the two services, thus obtaining alternative runs, corresponding to a *conditional plan*.

Conditional plans

An example of conditional plan is the following

begin_Pu; request_Pu;
(offer_Pu; begin_protocol_Sh; request_Sh;
(offer_Sh; accept_Pu; accept_Sh; end_Pu; end_Sh +
not_avail_Sh; refuse_Pu; end_Pu; end_Sh)) +
(not_avail_Pu; end_Pu)

This plan is represented as a regular program, where, in particular, “+" is the choice operator.

All runs of this plan satisfy the formula $D_{PS} \cup Constr$.
This can be proved in $DLTL$, since in this logic it is possible to deal with regular programs.

How to build a conditional plan

- Build the Büchi automaton obtained from the domain description D_{PS} and the constraints *Constr*.
- Mark as AND states those states of the automaton whose outgoing arcs are labeled with actions whose sender is one of the services.
- Extract from the automaton a conditional plan, such that all outgoing arcs from an AND state belong to an accepting run.

Approach similar to Pistore, Traverso et al., but with different formalisms and tools.