Towards verifying contract regulated service composition

Alessio Lomuscio, Hongyang Qu, Monika Solanki

Imperial College London
London, UK

Outline

Outline

Outline

Outline

CONTRACT

Outline

CONTRACT

Outline

CONTRACT

Outline

CONTRACT

General background

- Specifications for MAS very well explored. Range of logics (epistemic, deontic, ATL, ...).
- Model checking relatively much less explored.
- Verification of MAS. Previously standard model checkers employed. Computationally grounded semantics.
- Applications in security protocols, fault-tolerance in automatic vehicles, contract-regulated WS.

CONTRACT project (ends May09)

Some tasks we took on:

- Verifying compliance of contract-regulated WS.
- Local and Global run-time monitoring of contract-regulated WS.

Some tasks we **did not** take on:

- Analysis/negotiation/... of contracts per se.
- Design of a contract language.
- Formalisation of a contract language.
- Model checking/monitoring of particular WS languages naively.

A MAS angle

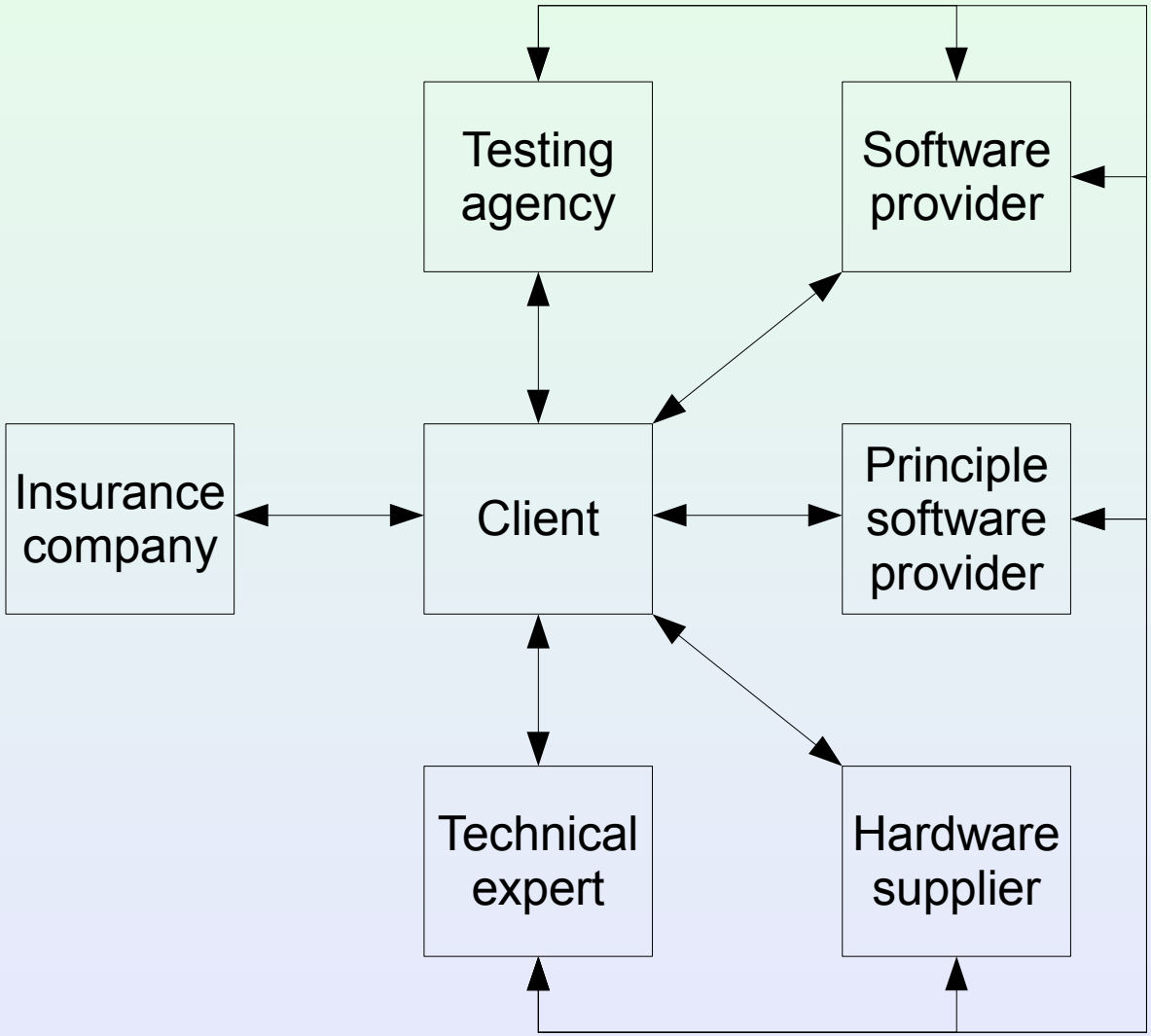
- Contracts as a mechanism for regulating complex interactions between web services implemented as MAS.
- Interest in verifying the MAS behaviours resulting from these interactions. Reason about whether contract violations happen, whether recovery actions are possible, whether recover may can enforced, and knowledge of the agents in these contexts.
- A MAS approach including temporal, epistemic, ATL, and some basic deontic concepts.
- SoA at the time included only reachability analysis via model checking.

Some key properties we may wish to reason about...

- what properties are brought about by a run of the system in which no agent violates any of his contracts,
- what properties hold true if some of the agents violate (part of) their contracts,
- what properties (recovery, degraded performance, ...) can agents enforce following some violation and what knowledge is required for this to happen.

A language supporting this sort of specs seems of potential interest.

A running example



A motivating example from Fujitsu's use case

- *Client* C asks *principle software provider* PSP and *software provider* SP to develop the software.
- PSP and SP twice update each other and C about the progress of the software development. C can request some changes in the software before the second round of updates.
- Every update is followed by a payment in part by the client C to the PSP. Payment to SP is handled by PSP.
- PSP integrates the components developed by SP and sends the software to *Testing agency* T for testing.
- After the software passes the test, C orders the hardware from *Hardware supplier* and buys insurance from *Insurance company* I for the software.
- C asks PSP, SP, H and an *Expert* to deploy the software on the hardware.
- If the deployment succeeds, C sends the final payment to PSP and SP for the development.
- If the deployment fails, C asks PSP for compensation.

Some of the parties' obligations

- PSP's obligations
  1. Update SP and C twice about the progress of the software.
  2. Integrate the components and send them to T for testing.
  3. If components fail, integrate the revised software and send them for testing.
  4. Make payment to SP after successful deployment of software.
- C's obligations
  1. Request changes before the second round of updates.
  2. Pay penalty if changes are requested after second round of updates.
  3. Make payment to the PSP after every update.

Temporal Deontic Interpreted Systems

- A system is composed of a set of agents A = $\{1, ..., n\}$ and an environment e.
- Each agent is described by
  - A set of *local states* L_i,
  - A set of *local actions* Act_i,
  - A *local protocol* function $P : L_i \rightarrow 2^{Act_i}$.
  - An *evolution* function $\tau_i : L_i \times Act \rightarrow L_i$.
- For any agent i its local states L_i are partitioned into two subsets: *green states* G_i and *red states* R_i.
- A *path* $\pi = (s_0, s_1, \ldots, s_j)$ is a sequence of possible global states such that $(s_i, s_{i+1}) \in T$ for each $0 \leq i \leq j$.

Models

A model $M = (S, I, T, \sim_1, \ldots, \sim_n, h)$ is a tuple such that:

- $S \subseteq L_1 \times, \ldots \times L_n \times L_e$ is the set of global states for the system,
- $I \subseteq S$ is a set of initial states for the system,
- T is the temporal relation for the system: sTs' if there exist actions $a_1, \ldots, a_n$ such that $a_i \in P_i(l_i(s))$ and $\tau_i(l_i(s), a_1, \ldots, a_n) = l_i(s')$.
- For each agent i $\sim_i$ is an epistemic relation defined by $(l_1, \ldots, l_n, l_e) \sim_i (l'_1, \ldots, l'_n, l'_e)$ if $l_i = l'_i$.
- $h : P \rightarrow 2^S$ is an interpretation for the set of propositional atoms P, including an atom v_i colouring the red states of agent i.

Temporal epistemic logic with violations

- Syntax

$$\phi ::= v_i | p | \neg\phi | \phi \wedge \psi | K_i\phi | EX\phi | EFF\phi | E\phi U\psi | EG\phi.$$

- Satisfaction
  - $(M, s) \models v_i$ iff $l_i(s) \notin g_i(s)$;
  - $(M, s) \models p$ iff $s \in h(p)$;
  - $(M, s) \models \neg\phi$ iff $(M, s) \not\models \phi$;
  - $(M, s) \models \phi \wedge \psi$ iff $(M, s) \models \phi$ and $(M, s) \models \psi$;
  - $(M, s) \models EX\phi$ iff there exists a path π starting at s such that $(M, \pi(1)) \models \phi$.
  - $(M, s) \models EG\phi$ iff there exists a path π starting at s such that $(M, \pi(k)) \models \phi$ for all $k \geq 0$;
  - $(M, s) \models E\phi U\psi$ iff there exists a path π starting at s such that for some $k \geq 0$ $(M, \pi(k)) \models \psi$ and $(M, \pi(j)) \models \phi$ for all $0 \leq j < k$;
  - $(M, s) \models K_i\phi$ iff for all possible global states s' if $s \sim_i s'$ then $(M, s') \models \phi$.

Basic approach

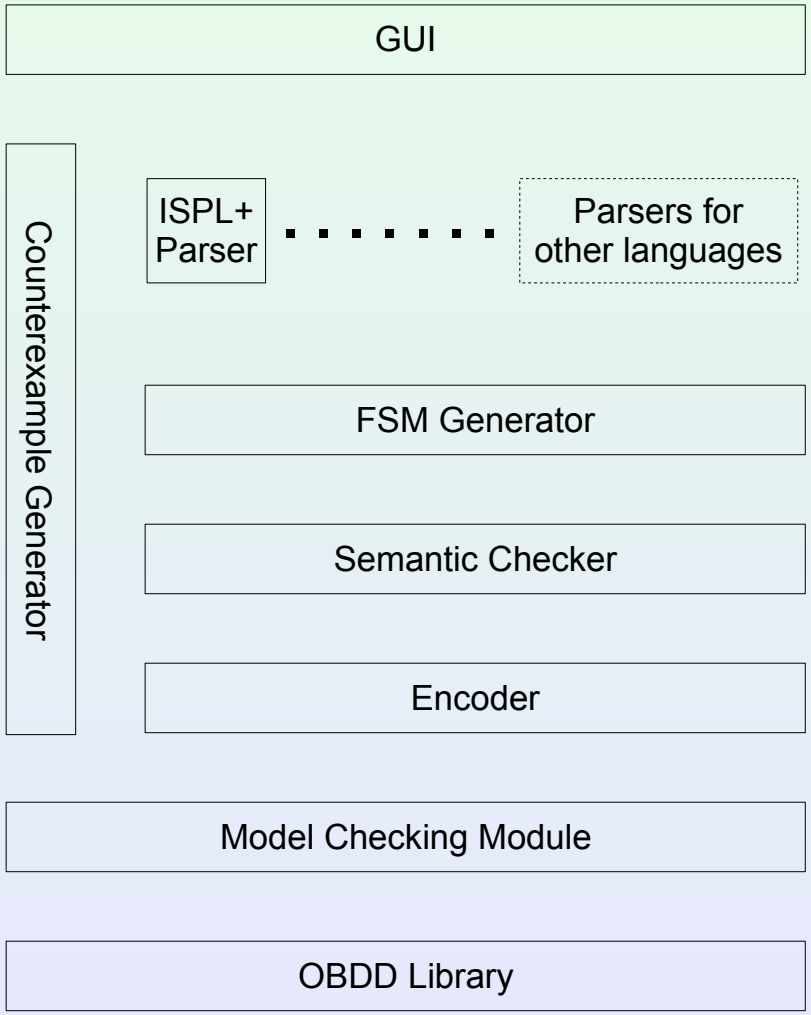
- Given a MAS and a set of contracts, model all possible transitions (via a language), colour the correct/incorrect states.
- Use syntax above to code temporal/epistemic/violation properties of interest.
- Implement a symbolic model checker.
- Test behavioural conformance of single or multiple agents against one or more contracts.

MCMAS and ISPL

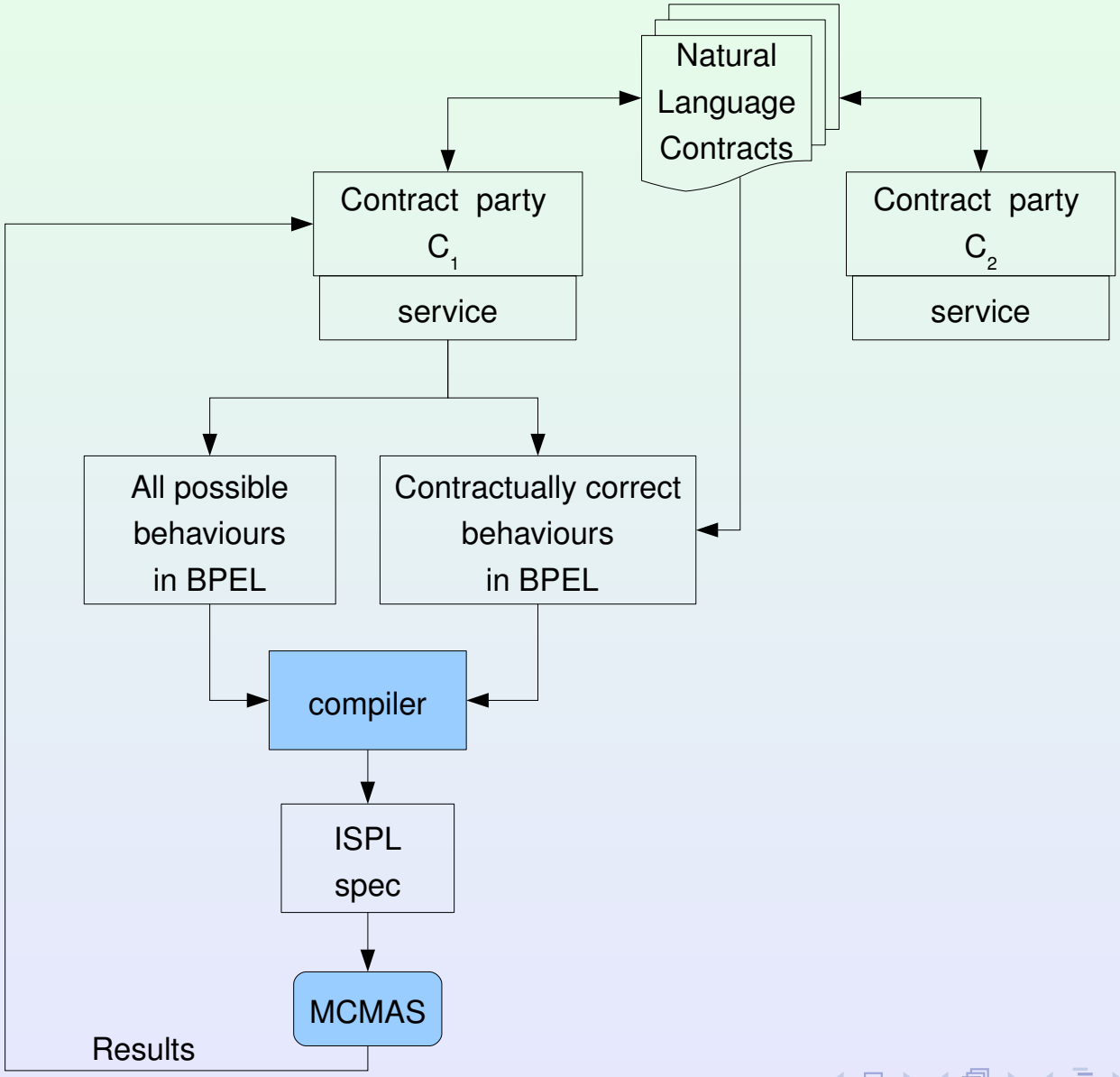
- MCMAS a model checker for multi-agent systems to verify CTL, epistemic, deontic and ATL formulae. It builds upon traditional ideas on symbolic model checking via OBDDs.

- ISPL is the input language of MCMAS. An agent in ISPL is defined as
  1. A set of local states, some of which are initial states;
  2. A set of local actions;
  3. A local protocol specifying for each local state, a subset of local actions that can be performed in that state;
  4. An evolution function defining the transition relation among local states.

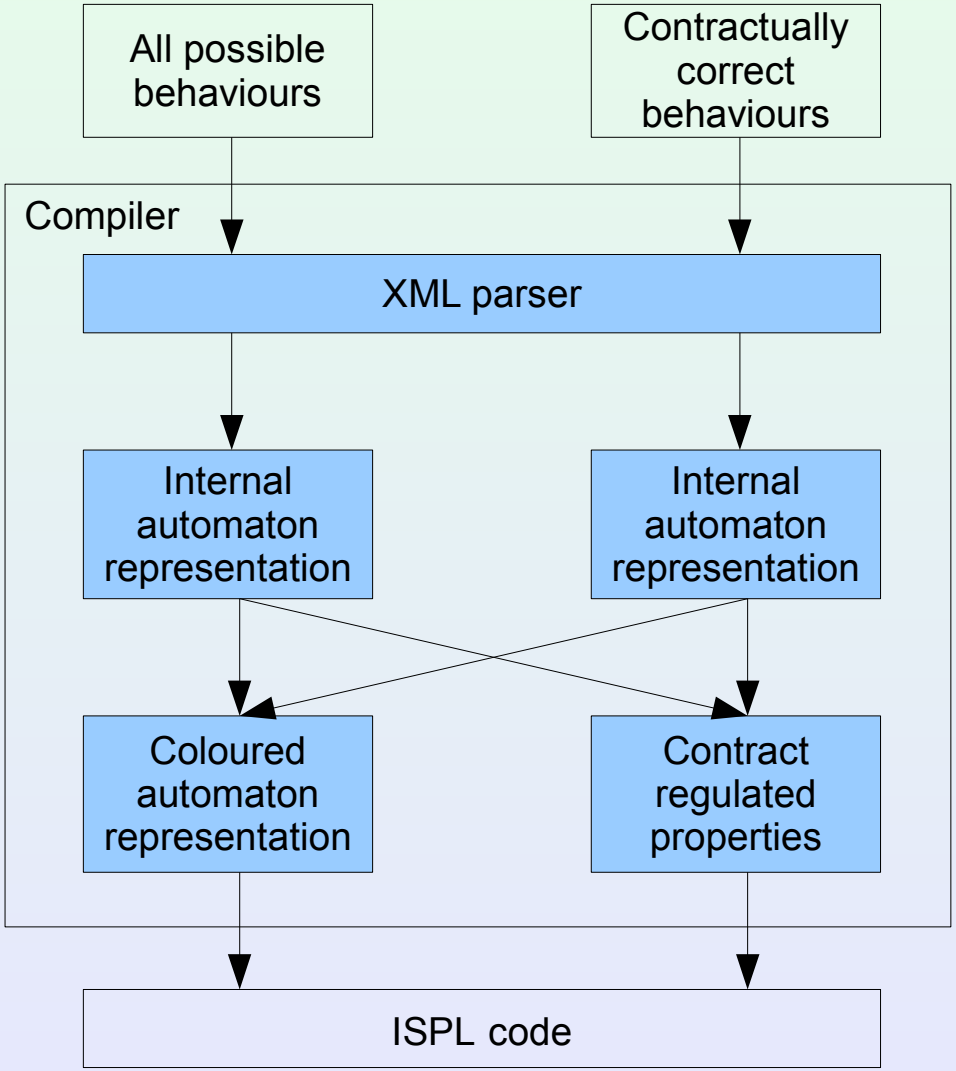
Architecture of MCMAS



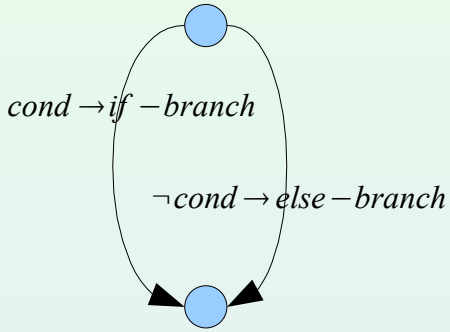
Towards automatisation: an experiment



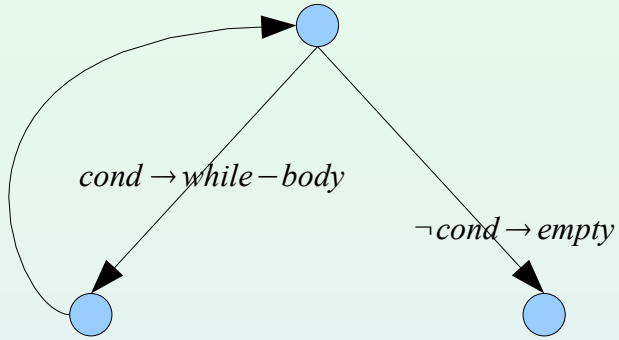
Compiler


All possible behaviours
Contractually correct behaviours
Compiler
XML parser
Internal automaton representation
Internal automaton representation
Coloured automaton representation
Contract regulated properties
ISPL code

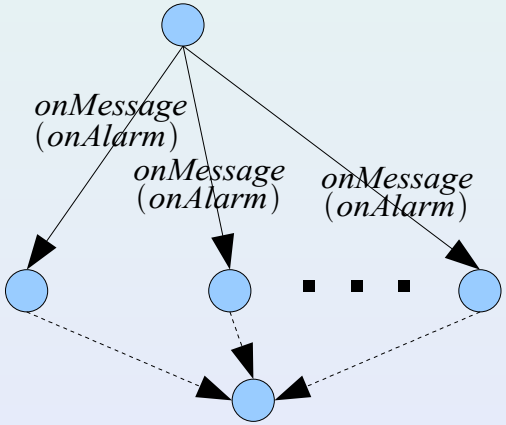
Translating BPEL programs into automata



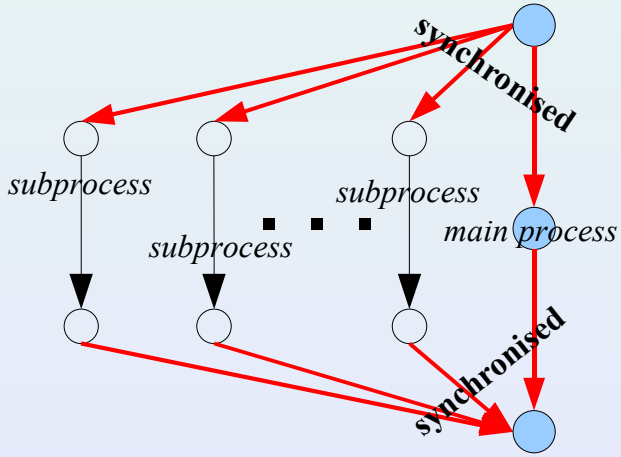
if



while

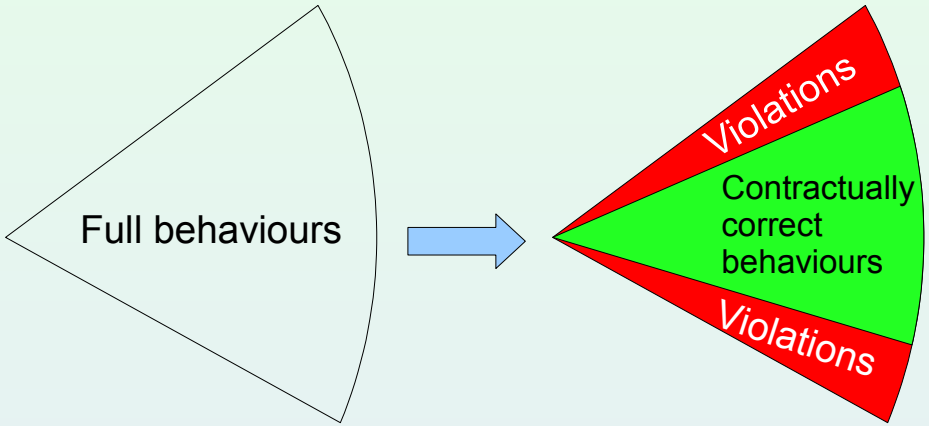


pick



flow

Colouring the state space



1. The initial state of a behaviour automaton is labelled as green.
2. For every transition in the contract automata, we find the same transition in the behaviour automata and label its target state as green.
3. For all states that are not green, we label them as red.

Basic properties

For each BPEL process, agent i, we define end_i holding in the end state of agent i;
We compile the following basic properties to check

$E(\neg v_i U end_i)$ (Is there a contract compliant run for agent i?)

EFv_i (Can agent i violate any of its contracts?)

Other properties may be added.

Generating ISPL programs

1. Local states generation: A local state $l \in L_i$ is a valuation for the set of *local variables* Var_i:

$$Var_i = Var_p \cup \{state\} \text{ or } Var_i = Var'_p \cup \{state\}.$$

2. Local actions generation: Act_i is obtained from the transitions of $\mathcal{A}_i$.
3. Protocol generation: For any transition t whose source state is represented by $l(state)$, the action to which t is mapped is included in E_l.
4. Evolution function generation: Each transition in $\mathcal{A}_i$ is translated to an evolution item:

state=s_2 if state=s_1 and c and Action=t; or

state=s_2 if state=s_1 and c and Action=t and $\mathcal{A}_j$.Action=t'; or

state=s_2 and v=$expr$ if $\cdots$; or

state=s if (state=s_1 or state=s_2 or ...) and c and Action=t and $\cdots$; or

state=s if c and Action=t and $\cdots$.

Modelling the example in BPEL

```
<pick name="Update1">
  <onMessage partnerLink="PSP_C"
   operation="recPSP" portType="ns1:recMsg"
   variable="RecPSPIn">
    <empty name="Empty1"/>
  </onMessage>
  <onMessage partnerLink="PSP_C_int"
   operation="recPSP" portType="ns1:recMoney"
   variable="SendSPIn1">
    <receive name="recUpdate1"
     createInstance="no" partnerLink="PSP_C1"
     operation="recPSP" portType="ns1:recMsg"
     variable="RecPSPIn">
    </receive>
  </onMessage>
  <onMessage partnerLink="PSP_NoC"
   operation="recNoPSP" portType="ns1:recMsg"
   variable="RecPSPIn">
    <exit name="Exit347"/>
  </onMessage>
</pick>
```

Modelling the example in ISPL (1)

```
Agent Client
  Vars:
    state : { Client_0, Client_1, ...};
    count : 0 .. 3;
    ...
  end Vars
  Actions={Client_Upd1_0, Client_Upd1_1,...};
  Protocol :
    state=Client_0:{Client_Upd1_0, Client_Upd1_1,
                    Client_Upd1_2, Client_While1};
    state=Client_1:{Client_Empty1};
    ...
    end Protocol
  Evolution :
    state=Client_0 and count=count+1 if
      state=Client_24 and Action=Client_Assign375;
    state=Client_1 if state = Client_0 and
      count<2 and Action = Client_Upd1_0 and
      PSP.Action = PSP_updateClient;
    ...
  end Evolution
end Agent
```

Modelling the example in ISPL (2)

```
Evaluation
  Client_green if Client.state = Client_0 or
                  Client.state = Client_1 or ...;
  Client_end if Client.state = Client_51;
  Client_red0 if Client.state = Client_11;
  ...
end Evaluation
Formulae
 E ( Client_green U Client_end );
 EF Client_red0;
  ...
end Formulae
```

Advanced properties

- Whenever PSP is in a compliance state, he knows the contract can be eventually fulfilled successfully.

$$AG(PSP\_green \rightarrow K_{PSP}EF(PSP\_end))$$

- There exists a path where C is always in compliance with the contract until he eventually receives the software.

$$E(C\_green\ U\ receiveSoftware)$$

- PSP knows that it is possible that PSP, SP, C, I, H, T and E are all in compliance until the software is delivered.

$$K_{PSP}E(all\_green\ U\ softwareDelivered),$$

where all_green represents
$PSP\_green \wedge SP\_green \wedge C\_green \wedge T\_Green \wedge H\_green \wedge E\_green \wedge I\_green$.

- For a certain violation v_a of agent a, b knows that a certain agent c may have violated its contract with a such that a becomes unable to satisfy its contractual obligations with b.

$$AG(K_bE(v_cUv_a))$$

- There is a trace in which the client is always in contract compliant states

Conclusions

- A general interest in representing/verifying MAS against rich logics.
- MCMAS, a BDD-based model checker.
- Still not found an adequate contract language.
- Runtime monitoring on contract-based WS, service discovery.