Towards verifying compliance in Semantic Web service compositions

Monika Solanki

Joint work with

Alessio Lomuscio, Hongyang Qu

Imperial College London
London, UK

Outline

1. Background
2. Temporal deontic interpreted systems
3. Specifying behavioural compliance
4. The motivating example
5. Specifications
6. Analysis and verification
7. Conclusions

Background

- Web services (WS) are one of the leading paradigms underlying application integration.
- When several subsystems coordinate in an open environment the end result may be much less predictable.
- Multi-agent systems (MAS) serves as a useful metaphor for reasoning about the services provided by “autonomous components acting rationally to maximise their own design objectives”.
- Contracts are a useful concept to govern and regulate MAS and agent implementations of WS.

Background

- Verification of WS is an active topic of research.
- However it has so far been concerned with checking safety and liveness properties only.
- However when WS are phrased as a contract-regulated MAS there are other properties that seem worth studying i.e.
  - various notions of correctness/violations of the contracts during a run
  - the evolution of the agents' knowledge about themselves the contracts and the expected peers' behaviours, etc.

In this talk...

- We explore the problem of specification and verification of compliance in agent based Web service compositions.
- We use the formalism of temporal-epistemic logic suitably extended to deal with compliance/violations of contracts.
- We illustrate these concepts using a motivating example where the behaviours of participating agents are governed by contracts.
- The composition is specified in OWL-S and mapped to our chosen formalism.
- Finally we use an existing symbolic model checker to verify the example specification whose state space is approximately 2^{21} and discuss experimental results.

The Verification problem

- Given a system S and a property ϕ, ascertain that

$$S \models \phi$$

- All behaviours of S satisfy ϕ
- Some proposed techniques:
  - Simulation and Testing: full coverage not guaranteed.
  - Algorithmic verification.
  - Deductive verification.
  - Abstraction.

Model Checking

An algorithmic verification technique,

- - applied to (not too big) finite state systems.
- - fully automatic, low computational complexity.
- - performs an exhaustive search of the state space of the system in order to establish whether a specification holds.
- Explicit representation of the system as labelled directed graphs leads to “state explosion” - works only for systems with a small state space.
- **Symbolic Model Checking**: the system and the specification formulas to be checked against are represented as Boolean formulas and then encoded canonically as OBDDs.

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 $\tau_i : L_i \times Act \rightarrow L_i$.
- Particularly, the set of local states L_i is 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 defined as $\tau_1 \times, \ldots \times \tau_n \times \tau_e$,
- For each agent i $\sim_i$ is an epistemic indistinguishably 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.

Temporal epistemic logic with correctness

- Syntax

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

- Satisfaction
  - $(M, s) \models p$ 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 $(M, s') \models \phi$.

Epistemic modality

- It is concerned with What does an agent “know”.
- The epistemic accessibility relation of an agent is based on the agent’s extended local states and on the environment local states.
- Intuitively, an agent “knows” something in a state of the system if this something is true in all the states of the system in which its local states and the observable variables of the environment remain the same.

Compliance

- There exists a path in which agent i is always in compliance.

$$EGg_i$$

- In all paths agent i will always be in full compliance.

$$AGg_i.$$

- whenever agent i Whenever agent i is in compliance the state of affairs ϕ holds in the system.

$$AG(g_i \rightarrow \phi).$$

- ϕ holds true whenever all agents in $A' \subseteq A$ are in compliance.

$$AG(\bigwedge_{i \in A'} g_i \rightarrow \phi)$$

- Whenever all agents in the system are in compliance ϕ holds.

$$AG(\bigwedge_{i \in A} g_i \rightarrow \phi)$$

- An agent i knows that as long as agent j is in compliance a certain state of affairs is always reachable in some way.

$$K_i(AG(g_j \rightarrow EX\phi))$$

Consequences of violations

- Following a violation by agent i a certain state of affairs hold indefinitely and that all other agents know this

$$AG(\neg g_i \rightarrow AG\phi) \wedge \bigwedge_{j \neq i} K_j(AG(\neg g_i \rightarrow AG\phi)).$$

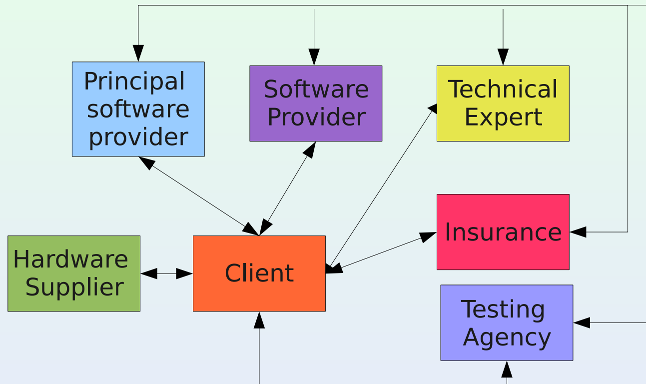
- Following a local violation, perhaps there is a way in the system for the agent i to recover.

$$AG(\neg g_i \rightarrow EFg_i).$$

- All agents always know that there is a way in the system for the agent i to recover from a local violation.

$$AG(\bigwedge_{i \in A} K_i AG(\neg g_i \rightarrow EFg_i)).$$

The motivating example



The “Informal” Contract

- *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 send the software to *Testing agency T* for testing.
- After the software passes the test, *C* orders the hardware from *Hardware supplier* and buy 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 *I* for the compensation.

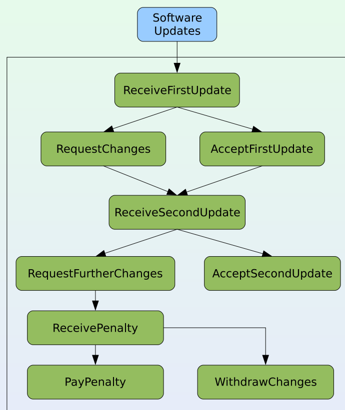
Obligations of contract parties

- *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.

The possible violations

Agent	Violation condition	Recovery
PSP	does not send messages to *SP* and/or *C* in the first and/or second run of update.	pay penalty charge
	does not send payment to *SP*.	no
SP	does not send update messages to *PSP* or *C*.	pay penalty charge
	does not send its components to *PSP*.	no
C	request changes after second update.	pay penalty charge or withdraw changes
	does not send the payment to *PSP*.	no
T	does not send the testing report to *C*, *PSP* and/or *SP*.	no
H	does not deliver the hardware system to *C*.	no
	ignores the deployment failure.	no
E	does not deploy the software on the hardware system.	no
I	does not process the claim of *C*.	no

Modelling the example in OWL-S




```
define composite process SoftwareUpdate
( inputs:  (firstUpdate -  xsd:string
firstUpdateStatus - xsd:boolean
changes - xsd: string
UpdateStatus - xsd:boolean
furtherChanges - xsd: string
penalty - xsd:string
withdrawNotice - xsd:string)
preconditions :( hasContract(contractID)
& receivedFirstUpdate(firstUpdateStatus)
& hasChanges(firstUpdateStatus, changes)
& receivedSecondUpdate(secondUpdateStatus)
& hasFurtherChanges(secondUpdateStatus,furtherChanges)
& receivedPenaltyMessage(penalty))
outputs:( firstUpdateStatus - xsd:boolean
 changes - xsd: string
 secondUpdateStatus - xsd:boolean
 furtherChanges - xsd:string
 withdrawReceipt - xsd:string)
 results :(
 hasFurtherChanges(secondUpdateStatus,furtherChanges)
 |-> output(penalty - xsd:string)
 hasReceivedPenaltyMessage() and paidPenalty(penalty)
 |-> output(penaltyReceipt -xsd:string)
 hasReceivedPenaltyMessage() and withdrawChanges()
 1-> output(withdrawChanges))
 { perform ReceiveFirstUpdate;
  perform RequestChanges;?
  perform AcceptFirstUpdate;
  perform ReceiveSecondUpdate;
  ...}
```

Specifications (I)

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

$$AG(g_{PSP} \rightarrow K_{PSP} EF(contractSucceed)) \quad (1)$$

- In some of the paths where *C* is always in compliance, he eventually receives the software.

$$EG(g_C \wedge EF(receiveSoftware)) \quad (2)$$

- In some of the paths where *PSP* is always in compliance, the software can be eventually integrated and tested.

$$EG(g_{PSP} \wedge g_{SP} \wedge EF(softwareIntegrated) \wedge EF(softwareTested)) \quad (3)$$

- *PSP* knows that for some paths, it is possible that whenever *PSP*, *SP*, *C*, *I*, *H*, *T* and *E* are all in compliance, the software can be eventually delivered.

$$K_{PSP}(EG(g_{all} \rightarrow EF(softwareDelivered))), \quad (4)$$

where g_{all} represents $g_{PSP} \wedge g_{SP} \wedge g_C \wedge g_T \wedge g_H \wedge g_E \wedge g_I$.

Specifications (II)

- It is possible for C to be in compliance until the software is deployed successfully but then entering a violation by not sending the final payment to PSP.

$$E((g_C \wedge EF(softwareDeployed))UEG(\neg g_C \wedge noPayment)) \quad (5)$$

- It is possible that SP is always in compliance before failing to provide the component requested by the PSP.

$$E(g_{SP} UEG(\neg g_{SP} \wedge componentNotProvided)) \quad (6)$$

- It is possible that PSP does not send the first update to C as per schedule and only sends it after paying a penalty to C.

$$E(g_{PSP} U ((\neg g_{PSP} \wedge noFirstUpdate) \wedge EX((g_{PSP} \wedge payPenalty) \wedge EX\ EG(g_{PSP})))) \quad (7)$$

- It is possible that C withdraws the request for change made after the second update.

$$E(g_C\ U\ ((\neg g_C \wedge illegalChangeRequest) \wedge EF (g_C \wedge withdrawChangeRequest \wedge EXEG(g_C)))) \quad (8)$$

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 $PSP_green \wedge SP_green \wedge C_green \wedge T_Green \wedge H_green \wedge E_green \wedge I_green$.

- There is a trace in which the client is always in contract compliant states until the software is delivered (while the client remains compliant) before the client enters a violation.

$$E(C_green \; U \; E((C_green \wedge softwareDeployed) \; U \; \neg C_green))$$

MCMAS

- MCMAS is a symbolic model checker developed particularly for multi-agent systems (MAS) to verify CTL, epistemic, deontic and ATL formulae.
- It takes as input a MAS specification and a set of formulae to be verified
- It evaluates the truth value of these formulae using algorithms based on **OBDDs** (Ordered Binary Decision Diagrams).
- Whenever possible MCMAS produces counterexamples for false formulae and witnesses for true formulae.

Interpreted Systems Programming Language

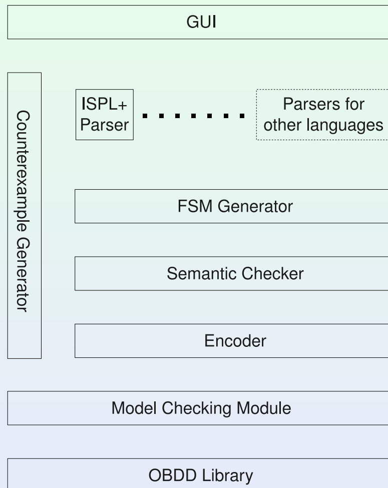
- MAS are described in MCMAS using a dedicated programming language derived from the formalism of **interpreted systems**.
- ISPL- resembles the SMV language, characterises agents by mans of variables and represents their evolution using Boolean expression.
- Two kinds of Agents: Standard and Environment.
- Environment agent: similar to standard agents and used to describe boundary conditions and infrastructure shared by standard agents. They are not always needed to be defined.

Interpreted Systems Programming Language

In MCMAS each agent (including the environment) is characterized by:

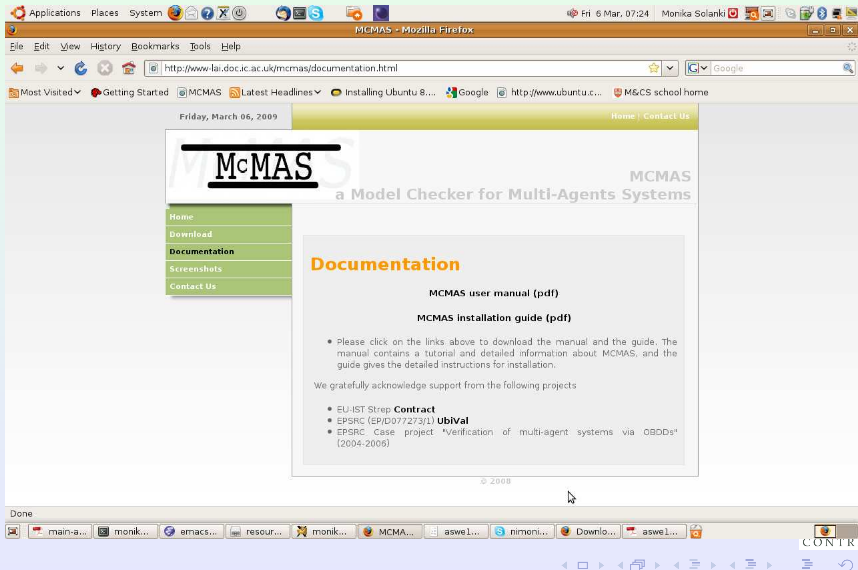
- A set of local states (for instance the states “ready” or “busy” for a re- ceiver) - defined in terms of *local variables*.
- A set of actions (for instance “sendmessage” or “open channel”.
- A rule describing which action can be performed by an agent in a given local state. We call this rule a protocol.
- An evolution function, describing how the local states of the agents evolve based on their current local state and on other agents’ actions.

Architecture of MCMAS



CONTRACT

MCMAS Home Page



MCMAS Demo

Verifying the illustrative example

Conclusions

- MAS serves as a useful metaphor for reasoning about the services provided by "*autonomous* components acting rationally to maximise their own design objectives".
- Contracts are a useful concept to govern and regulate MAS and agent implementations of WS.
- We address the issue of violation or non-compliance of pre-defined behaviour when specified as SLAs, contracts or protocols - specifically in a multi-agent scenario.
- We used ISPL along with MCMAS to show the verification of a service composition specified in OWL-S.
- We used a reasonably complex example (2^{21} states) to verify at design time, temporal-epistemic properties of services that capture the compliance levels of their implementing agents.