Towards verifying compliance in agent-based Web service compositions

Alessio Lomuscio
Department of Computing
Imperial College London
a.lomuscio@doc.ic.ac.uk

Hongyang Qu
Department of Computing
Imperial College London
hongyang.qu@doc.ic.ac.uk

Monika Solanki
Department of Computing
Imperial College London
m.solanki@doc.ic.ac.uk

ABSTRACT

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.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Model checking; H.3.5 [**Online Information Services**]: Web-based services

General Terms

Verification

Keywords

Web services, Model checking, compliance, epistemic logic.

1. INTRODUCTION

Web services (WS) are one of the leading paradigms underlying application integration and consequently are driving the current IT efforts in the provision of solutions for business and services. While a system made of few and localised services may only interact in a small number of ways, when several subsystems are able to coordinate in an open environment the end result may be much less predictable. Certain components may fail, others may be incapacitated to provide the services in the expected timeline and others still may have to adopt a policy of prioritisation among the requests that are being received. In this context the paradigm of 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" [14]. Indeed the W3C consortium [1] suggested that "A web service is an abstract notion that must be implemented by a concrete *agent*. The agent is the concrete piece of software or hardware that sends and receives messages."

In this context while the designer of the system as a whole cannot *guarantee* an ideal outcome for the service composition — since he or she has no overall control over it — he may still wish to establish verifiable mechanisms to create an incentive in the agents to carry out transactions in a way that is more likely to create an overall positive outcome. One such mechanisms is *service level agreements (SLAs)*. SLAs are rules representing agreed level of service provision to be supplied by the agents when interactions are invoked within certain parameters. For example, a certain SLA may prescribe that all requests from a client are to be answered within 1 min whenever they are requested from 9am to 5pm and within 3 mins from 5pm to 9am. While monitoring and identifying responsibilities in the violation of SLAs is a non-trivial problem whenever service inter dependencies exist, it is well recognised that SLAs can act as a basic regulatory mechanism and may help engineers in predicting the average behaviour of the system. There certainly is an increasing emphasis on reasoning about SLAs in software engineering and in the implementation of platforms supporting them.

Although SLAs are useful, they can represent only basic agreements of service provision. Applications running complex, human-like activities require more general and sophisticated declarative specifications certifying legal-like agreements among the parties. These mechanisms should not only describe the intended timeline for the provision of services but would also have to specify obligations, permissions of states and/or actions in a variety of functioning circumstances including those coming into force as a result of certain agents not performing "as expected". A useful concept from the legal domain in this sense is the one of *contract* as found in human societies. Contracts are legally binding agreements regulating the behaviours between the parties. Should a contract be broken by one of the parties, "legal remedies" may be applicable in the form of penalties, additional rights to some other party, and possibly additional penalties with respect to third parties. Hence, contracts are a useful concept to govern and regulate MAS and agent implementations of WS. A key characteristic of them is that they may still be broken. In this setting system engineers may be interested in investigating what behaviours the MAS implements when all agents are fulfilling their contracts, but also, and perhaps more importantly, what properties the system exhibits when some of the agents are violating their

Cite as: Towards verifying compliance in agent-based Web service compositions, Alessio Lomuscio, Hongyang Qu and Monika Solanki, *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, Padgham, Parkes, Müller and Parsons (eds.), May,12-16.,2008,Estoril,Portugal,pp. 265-272.

contracts in certain or other ways. Particularly, one may want to check if some really unwanted behaviours may result following certain violations or whether the system provides certain elements of regimentation thereby avoiding them.

The points above bring us to consider the issue of verification of MAS implementing WS. Verification of WS is an active topic of research (e.g., see [12,16]). 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, such as 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.

There is a tradition in the MAS community to use rich logic-based languages to analyse the behaviour of agents in the system. In particular, not only is temporal logic used but also, among others, epistemic (to reason about knowledge of the processes), deontic (to reason about obligation of the processes), cooperation (to reason about strategies of the agents), and other modalities. These logic-based languages can be used to specify formally and unambiguously the behaviour of the system. Recent developments in the verification of MAS via model checking techniques [2,11,15] permit the verification of not only plain temporal languages but also a variety of modalities describing the informational and intentional state of the agents. The above leads us to believe that verifying contract-based WS implemented by MAS is a worthwhile programme of research. In this paper we set out to conduct an exercise in modelling WS as MAS and analyse them by means of concept-rich, yet fully-formal specification languages. We also report on how an existing symbolic model checker can be used to verify such rich specifications. As a first step in the direction highlighted above we do not represent the contracts explicitly. Instead we focus on the resulting correctness of the states of execution of the agents with respect to the contracts that are currently active at the time. This colouring of states is the result of the interpretation of the relevant active contracts with respect to the executions of the agents.

The rest of the paper is organised as follows. In Section 2 we introduce the trace-based semantics of interpreted systems and discuss various notions of violations. Section 3 introduces a motivating example, including part of its OWL-S specification, and some of its key properties. Section 4 presents the formalisation of the example in ISPL and results obtained from verifying the example using MCMAS. We conclude in Section 5 also comparing our results to existing work.

2. TEMPORAL DEONTIC INTERPRETED SYSTEMS

We introduce here a formalism to express notions pertaining to the temporal evolution of MAS, the knowledge of agents in the system as well as the correctness and violations of states and runs with respect to a predetermined set of contracts regulating the interaction among the agents. Intuitively we would like to be able to express specifications capable of expressing:

- what properties are brought about by a run of the system in which no agents violates any of his contracts,
- what properties hold true if some of the agents violate (part of) their contracts,
- what knowledge the agents have about the consequences of some other agents violating some of their contracts and how this knowledge evolves over time.

These can serve as intuition, but we refer to Section 3 for concrete examples.

Although some preliminary attempts to introduce contract-modelling languages are being put forward we were not able to identify one that would suit our intention of performing automatic verification. As a consequence we will not model the contracts themselves but we will simply use a flag reflecting whether or not the agent in question is at that time in compliance (respectively in violation) of the set of contracts applicable to them. Clearly there is scope for further work in this direction. Further note that we do not discuss the issue of contract negotiation here. We assume all contracts have been negotiated at the beginning of the execution of MAS. Contract negotiation does generate interesting issues but we are not able to discuss them here.

2.1 Semantics

We model a MAS as composed of a set of agents and environment. We assume that each agent is implementing a web service providing particular functionalities. We follow the interpreted system model [4] and assume that at any given time each agent in the system is in a particular local state. This local state can be a state of compliance with respect to the agent's contracts or of violation. We will call the former allowed (or green) states, and disallowed (or red) the latter.

Each agent has a repertoire of actions available; the action selection mechanism is given by the notion of local protocol, effectively a function giving the set of possible actions that may be performed when in a given local state. The system evolves by means of transitions from a collection of (instantaneous) local states to another following the execution of actions for all the agents in the system.

The notions of entitlement, penalty, etc., given by compliance with respect to a given set of contracts is incorporated in the notions of protocol and transition. This will be exemplified in the example of the next section.

For the above purposes we adopt the model of deontic interpreted systems [8] as extended to temporal models as in [10]. Formally, we assume a set of agents $A = \{1, \ldots, n\}$ and an environment e.

To each agent i we associate a set of instantaneous *local states* L_i and a set L_e to the environment. For each agent i we assume the set of local states is partitioned into two subsets $L_i = G_i \cup R_i$: G_i represents *green* (or *ideal*) local states, R_i represent the *red* (or *non-ideal*). Intuitively G_i represent states of compliance with respect to the contract the agent i is subjected to, whereas R_i represents states of violation.

To represent the instantaneous configuration of the whole MAS at a given time we use the notion of global state. A global state $s \in S$ is a tuple $s = (l_1, \ldots, l_n, l_e)$ where each component $l_i \in L_i$ represents the local state an agent i is in (these may be either a green or a red state), together with the environment state. The set of all global states $S \subseteq L_1 \times \cdots \times L_n \times L_e$ is a subset of the Cartesian product of all local states and the local states for the environment.

$I \subseteq S$ is a set of initial states for the system.

The formal model we use accounts for the temporal evolution of the system. To do this we further assume that each agent i has a repertoire of actions ACT_i at his disposal, similarly the environment. It is assumed $null \in ACT_i$ for each agent i where $null$ is the null action. Actions are selected by means of action selection mechanisms local to the agents; this is formalised by protocol functions $P_i : L_i \to 2^{Act_i}$ for any $i \in A$. In other words $P_i(l_i)$ represents the actions that may be performed in the state l_i (irrespective as to whether l_i is a red or green state). Some of these actions will lead to green states for the agent, others to red ones. A tuple $(a_1, \ldots, a_n, a_e)$ in which every component represents the action carried out by an agent (the environment for the last component) is called a *joint action.*

The evolution of the system is given by locked transitions for all the agents and the environment. The model assumes that each agent moves from local state to local state at each time tick. The transitions between local states depend on which actions have been performed by all agents in the system. So an agent's action may affect another agent's resulting next state. Although this is not enforced in the semantics, in any concrete example we will impose that the colour of the resulting target state will be green or red depending on the local action the agent himself has performed; if necessary we can have two copies for a certain local state, one green and one red, to differentiate outcomes depending on the agent's latest action and the ones of the rest of the system. Formally, for each agent we assume a local transition function $\tau_i : L_i \times Act_1 \times \ldots \times Act_n \times Act_e \to L_i$ defining the local state for agent i resulting from a local state and and a joint action.

Local transitions may be combined together (the model checker presented later will do precisely this) to give a joint transition function $\tau : S \times Act_1 \times \ldots \times \ldots Act_n \times Act_e \to S$ giving the overall transition function for the system. We write $(s, s') \in T$ if $\tau(s, a_1, \ldots, a_n, a_e) = s'$ for some joint action $(a_1, \ldots, a_n, a_e)$.

We introduce paths as standard to give an interpretation to a branching time language. 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 $0 \leq i \leq j$. For a path $\pi = (s_0, s_1, \ldots)$, we take $\pi(k) = s_k$.

Definition 2.1 (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 above,*
- *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 $l_i = l'_i$.*
- *$h : P \to 2^S$ is an interpretation for the set of propositional atoms P.*

The above models allow us to interpret a temporal epistemic language. The relation T will be used to interpret temporal operators whereas $\sim_i$ will be used to interpret epistemic modalities as standard [4].

2.2 Syntax

Our formal language is a multi-modal logic including operators for branching time, epistemic operators and specialised local variables expressing correctness and violations. We will see that by combining local propositions for violations and correctness with temporal and epistemic operators we can express a variety of notions of compliance.

Definition 2.2 (Syntax) *The syntax of the specification language is given by the following BNF syntax:*

$$\phi ::= p | g_i (i \in A) | \neg\phi | \phi \wedge \psi | K_i\phi | EX\phi | EF\phi | E\phi U\psi | EG\phi.$$

In the above definition, p is an atomic proposition and g_i is an i-local atomic proposition expressing that "agent i is presently in compliance (with respect to a set of contracts)". We use g_i as we will often say that in this case "agent i is in a green state". We sometimes write r_i for $\neg g_i$ expressing that "agent i is presently not in compliance", or "agent i is in a red state". The formula $EG\phi$ stands for "there exists a path accessible from the present state in which ϕ holds globally", i.e., ϕ holds in every future state in at least a path; $EF\phi$ stands for "there exists a path in which ϕ holds at some future state"; $EX\phi$ stands for "ϕ holds in the next state in at least one path accessible from the present state"; $E\phi\mathcal{U}\psi$ stands for "there exists at least one path where ψ holds at some point in the future and ϕ holds in all states until then". $K_i\phi$ means that "agent i knows ϕ". For examples and interpretation of the temporal epistemic fragment we refer to specialised literature on the subject [4].

We can now interpret our logical language.

Definition 2.3 (Satisfaction) *Satisfaction for a formula ϕ in a model M at a global state $s = (l_1, \ldots, l_n, l_e)$, denoted as $(M, s) \models \phi$, is defined recursively as follows:*

- *$(M, s) \models p$ iff $s \in h(p)$;*
- *$(M, s) \models g_i$ iff $l_i \in G_i$;*
- *$(M, s) \models \neg\phi$ 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 $(M, s') \models \phi$.*

The other connectives, such as AX, AG, AF and AU, are defined via the above as standard [3]. For example, $AX\phi = \neg EX(\neg\phi)$.

The definition of satisfaction above is standard in temporal epistemic logic and only extends the literature by adding propositional constants g_i for compliance of agent i. There are clear correspondences between these and what is discussed in [9].

Often we are interested in establishing whether a model M representing a whole system satisfies a specification ϕ, represented as $M \models \phi$. In this case we will check whether $(M, s) \models \phi$ for all $s \in I$.

2.3 Expressivity

We now formalise various notions of behavioural compliance with respect to a set of contracts by using the semantics above. There are many dimensions of possible investigation here: compliance may be *local* or *global*, it may hold for portions or for the whole length of a path, etc. We only focus on some of these here.

Compliance. We begin by analysing the notion of local compliance of agent i over a whole path. In particular we can distinguish between possible local compliance over the whole path ("there exists a path in which agent i is always in a green state") and inevitable local compliance ("in all paths agent i will always be in full compliance"). The former can be expressed in the syntax above by

$$EGg_i$$

and the latter by

$$AGg_i.$$

The above allows us to specify easily the consequence of full local compliance. For instance, if we need to express that "whenever agent i is in compliance the state of affairs ϕ holds in the system," we could state:

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

Should we need to refer to states resulting from more than one agent being in compliance we can obviously take the conjunction of the respective g_i. For instance

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

represents the fact that ϕ holds true whenever all agents in $A' \subseteq A$ are in compliance.

By allowing A' to grow to represent the whole set of agents we can refer to "full global compliance"

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

representing "whenever all agents in the system are in compliance ϕ holds."

We can combine the above with knowledge modalities. For example we may want to express that an agent i knows that as long as agent j is in compliance a certain state of affairs is always reachable in some way, expressible by $K_i(AG(g_j \to EX\phi))$. Obviously more complex specifications are possible.

Consequences of violations. In addition to the above, we may be interested in what consequences arise should one agent not be in compliance. For instance we may wish to express the fact that following a violation by agent i a certain state of affairs hold indefinitely and that all other agents know this. This is expressible by the formula

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

Most often we are interested in the notion of "recovery". Following a local violation, perhaps there is a way in the system for the agent to recover (often contracts prescribe penalties to be payable following certain violations). This is expressible as:

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

In the language above we also can easily express that all agents always know this:

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

Similar formulas may be introduced regarding recovery for global violations as well.

The above are only some examples of the possibilities of the language. We analyse a concrete WS example in the next section and analyse it in Section 4 in view of the above.

3. A MOTIVATING CASE STUDY

In this section we present a composition of services which are composed as per a pre-defined contract, negotiated between services. Note that we use the notion of contracts only as an illustrative mechanism to exemplify the concepts of correctness and violations. We do not go into details or formalise the intricacies of each of the processes in the composition.

In our example the participating agents or services, as illustrated in Figure 1, are: a principal software provider (PSP), a software provider (SP), a software client (C), an insurance company (I), a testing agency (T), a hardware supplier (H) and a technical expert (E).

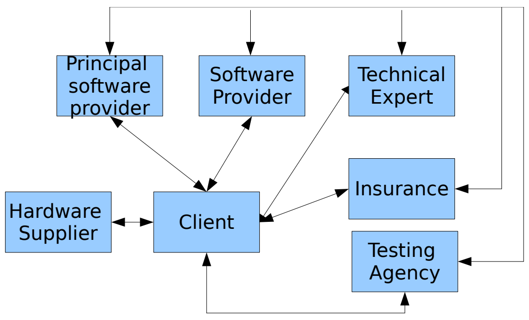

Figure 1: Interaction between various partners in the composition.

The high-level workflow of the composition can be defined as follows: Client C wants to get a software developed. The software is to be deployed on hardware supplied by H. To deploy the software, the technical expert E is needed. Components of the software are provided by different software providers. We consider two software providers here: PSP and SP. The components need to be integrated by the providers before the software is delivered to C.

The software integration is carried out by PSP when SP sends its component to PSP. PSP and SP twice update each other and C about the progress of the software development. If the client would like some changes in the software he can request them before the second round of updates. Any change suggested by the client after the second update is considered a violation and the client is charged a penalty. The client can recover from this violation by paying the penalty or by withdrawing the request for changes. If PSP and SP do not send their updates as per schedule, this is also considered a violation and they are charged a penalty. Every update is followed by a payment in part by the client C to the PSP. Payment to SP is handled by PSP and is done once the software is deployed successfully.

PSP integrates the components and sends the integrated component to T for testing. Results from testing are made available to all the parties, i.e., PSP, SP and C. If the test fails, the components are revised and tested again. Components can be revised twice. If the third test fails C always cancels the contract with PSP. If the testing succeeds, C invokes I to get the software insured. C then invokes H to order the hardware. Finally C invokes E to get the software deployed. If the software cannot be deployed then the hardware and the components have to be re-evaluated. Components can be revised twice. If the third test fails C always cancels the contract with PSP and H.

From the above scenario it can be seen that contracts between services can be usefully employed to illustrate the notion of correctness in behaviour. Any deviation from the behaviour identified in the contract is considered a violation. The contract might in some cases also specify mechanisms for recovering from violations. Keeping this in perspective we require that before the composition can be realised, contracts are established and negotiated between the different parties involved.

The contract between various parties can be violated in many ways. Table 1 illustrates informally some of the conditions under which some local violations may occur.

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

Table 1: Agents and their violation conditions. .

3.1 Specifying the "Client" as an OWL-S service

Although several standards are now available for the specification of services and their composition, we use OWL-S [13] below.

While verifying behaviour of services for contract compliance, it is crucial to verify their accompanying pre and post conditions. The techniques proposed in our framework, allow us to easily model the pre and post conditions using the input language of the model checker below and thus facilitate their verification.

In OWL-S a service is identified as a process which has properties defined in terms of IOPR (inputs, outputs, preconditions and results). Processes can be atomic or composite. Since in this paper, our objective is to model and verify correctness of behaviour and its violations, we do not emphasise on the process description capabilities of OWL-S and present only a high level representation of the composition.

In our example the client C is a composite process. We assume the contracts to have been negotiated and be active at the beginning of the run. The process is a composition of the following sub processes:
`SoftwareUpdates`, `TestResults`, `SoftwareReceipt`, `SoftwareInsurance`, `HardwareOrder`, `SoftwareDeployment`. Each of these processes can be further defined as being atomic or composite depending on the choice of granularity in process description, i.e., black, grey, or white box. One possible composition of these processes for the parent client process is as illustrated in Figure 2. Specifying informally, the client monitors updates for the software, receives the test results and then receives the software. These processes are performed in *sequence* followed by the parallel execution of insuring the software and ordering the hardware (*spilt+join*). Finally the software is deployed.

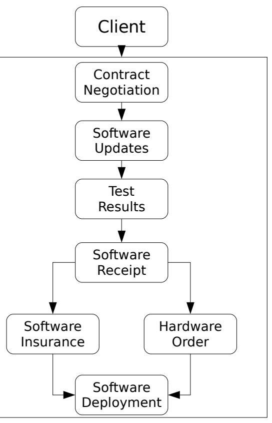

Figure 2: Composition of services for the "Client".

As an example we specify "`SoftwareUpdates`" in detail. As illustrated in Figure 3, the process `SoftwareUpdate` is composed of the following sub processes:
`ReceiveFirstUpdate`, `RequestChanges`, `AcceptFirstUpdate`, `ReceiveSecondUpdate`, `RequestFurtherChanges`, `ReceivePenaltyMessage`, `AcceptSecondUpdate`, `PayPenalty`, `WithdrawChanges`

Processes `ReceiveFirstUpdate`, `ReceiveSecondUpdate` and `ReceivePenalty` are composite processes composed using the "choice" control construct while the other processes are atomic. The client receives the first update following which he can request changes or accept the update. This is followed by the second update. If the client requests any changes to the software at this stage, he is required to pay the penalty. This is considered as violation. By withdrawing the request or by paying the penalty the agent may recover from this violation. Note that this behaviour is in accordance with the workflow defined earlier and forms a part of the contract between PSP and C.

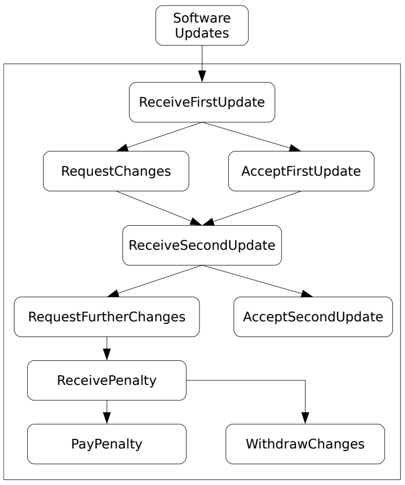

Figure 3: Composition of services for "SoftwareUpdates".

The IOPRs for the process are the computed IOPRs for the individual atomic processes. We specify some snippets of the `SoftwareUpdate` process using the presentation syntax of 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;
  perform RequestFurtherChanges;?
  perform AcceptSecondUpdate;
  perform ReceivePenaltyMessage;
  perform PayPenalty;?
  perform WithdrawChanges;}
```

4. ANALYSIS AND VERIFICATION

As we show below, properties specifying behavioural correctness can be verified using MCMAS [7]. Implemented in C++ and based on OBDD technique, MCMAS is a model checker developed particularly for multi-agent systems to verify CTL, epistemic, deontic and ATL formulae. The procedure of verifying properties is briefly described as follows: A system and its properties are fed to MCMAS in the format of ISPL [7], the input language of MCMAS. The ISPL file is parsed using standard tools, and parameters such as agent names and states, are stored in temporary lists. The lists are traversed to build the OBDDs for the verification algorithm. An OBDD representing the set of states in which a specification formula holds is computed. The OBDD for the set of reachable states is then compared with the OBDD corresponding to each formulae. In case of an equivalence the tool reports a positive output, otherwise a negative output is produced.

In what follows, we encode the process outlined in section 3.1 in ISPL. We then formalise a few properties using the formal model specified in section 2 and verify them using MCMAS.

4.1 Encoding specifications in ISPL

There is a precise correspondence between the semantics of section 2 and the language ISPL. An agent i 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 protocol specifying for each local state, a subset of local actions that can be performed in that state, and (4) an evolution function defining the transition relation among local states. The properties to be verified are encoded as temporal-epistemic formulae and are defined over atomic propositions, each of which is mapped to a set of compound states across some agents.

To model the complete system as illustrated in Section 3, we define seven agents, each of which represent the main thread of a service. We begin by representing the composition as a state transition diagram such that we put local states before and after atomic processes and these atomic processes become actions connecting local states. For example, Figure 4 shows that atomic process `receiveFirstUpdate` in `SoftwareUpdate` is translated into an action *receiveFirstUpdate* starting from a local state s to another local state s'. Since OWL-S does not define processes directly in terms of

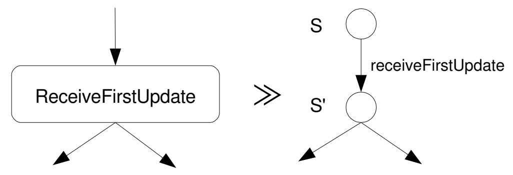

Figure 4: State transition diagram for `receiveFirstUpdate`.

the primitives of interpreted systems, we need to extract these from the IOPRs and the process definition. For the agents running parallel processes, such as PSP, SP and C, extra agents or processes are generated. For example, three extra agents are constructed for PSP to model its concurrent executions of sending update messages to SP and C, and receiving update messages from SP. These three agents are synchronised with the main thread of agent PSP. In total, 19 agents are generated. The following piece of ISPL code[1] shows the definition of agent C.

```
Agent I
  -- Local states
  Lstate={c0, c1, c2, c3, c4, c5,...};
  -- local actions
  Action={waiting, start, endSuccess, endFail,
          contractFail, null...};
  -- Local protocol
  Protocol :
    c0 : {waiting};
    c1 : {start };
    c2 : {waiting};
    c3 : {endSuccess, endFail};
    c4 : {contractFail};
    c5 : {null};
    ...
  end Protocol
  -- Evolution function
  Ev :
    c1 if (Lstate = c0 and Action = waiting);
    c2 if (Lstate = c1 and Action = start and
           C_1.Action=start and C_2.Action=start);
    c3 if (Lstate = c2 and Action = waiting);
    c4 if (Lstate = c3 and Action = endFail and
           (C_1.Lstate=c1_7 or C_2.Lstate=c2_7));
    c5 if (Lstate = c4 and Action = contractFail) or
           (Lstate = c20 and Action = contractFail);
    ...
  end Ev
end Agent
```

4.2 Specifications

In this subsection we formalise various properties of compliance (or lack of) for the motivating case study outlined in Section 3 using the Syntax defined in Section 2.

- Whenever PSP is in a green state (i.e., is in a state of compliance), he knows the contract can be eventually fulfilled successfully.

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

  Intuitively this property should not hold because even though PSP is in compliance, the software might not pass testing or cannot be deployed.

- 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 $g_{PSP} \wedge g_{SP} \wedge g_C \wedge g_T \wedge g_H \wedge g_E \wedge g_I$.

- 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.

$$E((g_C \wedge EF(softwareDeployed)) \; U \; EG(\neg g_C \wedge noPayment)) \quad (5)$$

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

$$E(g_{SP} U \; EG(\neg g_{SP} \wedge componentNotProvided)) \quad (6)$$

- 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.

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

4.3 Model properties in ISPL

Before we check the temporal-epistemic specifications above, it is necessary to define an evaluation function in the ISPL model. This function maps atomic propositions to states and specifies for every atomic proposition the set of states in which the proposition holds. Conditions in OWL-S, e.g., *preconditions*, can be specified in terms of atomic propositions. In order to model compliance, we define an atomic proposition g_i that holds on all green states of agent i. For example, g_C for the client is defined in ISPL as follows:

```
Evaluation
...
g_C if C.Lstate=c0 or C.Lstate=c1 or C.Lstate=c2
       or C.Lstate=c3 or C.Lstate=c4, ...;
...
end Evaluation
```

Specification formulae are specified in "Formulae" section in ISPL. For example, formula (5) is defined in ISPL as follows:

```
Formulae
...
E ((g_C and EF softwareDeployed) U
   EG (!g_C and noPayment));
...
end Formulae
```

4.4 Experimental verification Results

We encoded the scenario and the specification above in ISPL and verified it using MCMAS. Our system was running on Linux Fedora 8 (kernel 2.6.23) on Intel Core 2 Duo E6750 2.66GHz with 2GB memory. 68 BDD variables were generated to encode the local states of agents and 66 BDD

[1]The complete code of the system is available from the authors upon request.

variables to encode actions. In order to encode the transition relation, an additional copy of BDD variables for local states was constructed. In total, 202 BDD variables were used. The overall state space is estimated to have 2^{21} global states and uses 111 MB memory space. It took 41 minutes for MCMAS to verify the properties in the previous subsection. Table 2 presents the results. It can be seen that they are in line with what expected.

Property	Satisfaction	Property	Satisfaction
1	no	5	yes
2	yes	6	no
3	yes	7	yes
4	yes	8	yes

Table 2: Verification of the properties.

5. CONCLUSION

Extensive research has been done on the automated verification of Web service composition. Pistore et al [12] present a technique based on "Planning as Model Checking" for planning under uncertainty for composition and monitoring of BPEL4WS processes. Fu et al [5] presents a framework where BPEL specifications are translated to an intermediate representation, using guarded automata as XPath expressions. This is followed by the translation of the intermediate representation to a verification language "Promela", input language of the model checker SPIN. Hu Huang et al [6] presents an approach using the BLAST model checker to verify the process models of OWL-S.

None of the above efforts, however, 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.

In this paper, we used a reasonably complex example (2^{21} states) to show that it is possible to verify at design time, temporal-epistemic properties of services that capture the compliance levels of their implementing agents.

Much work still needs to be done in this line, e.g., there is presently no automatic way to generate the green/red states for a language representing contracts. Also it remains to be seen how the input language of the checker can be adapted to accept a suitable abstraction of the service composition so that a tool can be used to provide automatic compilation.

Acknowledgements. *The research described in this paper is partly supported by the European Commission Framework 6 funded project CONTRACT (IST Project Number 034418).*

6. REFERENCES

[1] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard. Web service architecture. w3c working group note 11 february 2004, 2004. http://www.w3.org/TR/ws-arch/.

[2] R. Bordini, M. Fisher, C. Pardavila, W. Visser, and M. Wooldridge. Model checking multi-agent programs with CASP. In *CAV'03*, volume LNCS 2725, pages 110–113. Springer-Verlag, 2003.

[3] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[4] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about Knowledge*. MIT Press, Cambridge, 1995.

[5] X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *13th international conference on World Wide Web*, pages 621–630. ACM Press, 2004.

[6] H. Huang, W. Tsai, R. Paul, and Y. Chen. Automated model checking and testing for composite web services. In *Proc. of ISORC'05*, pages 300–307. IEEE Computer Society, 2005.

[7] A. Lomuscio and F. Raimondi. MCMAS: A model checker for multi-agent systems. In *Proceedings of TACAS 2006*, volume 3920, pages 450–454. Springer Verlag, 2006.

[8] A. Lomuscio and M. Sergot. Deontic interpreted systems. *Studia Logica*, 75(1):63–92, 2003.

[9] A. Lomuscio and M. Sergot. A formalisation of violation, error recovery, and enforcement in the bit transmission problem. *Journal of Applied Logic*, 2(1):93–116, Mar. 2004.

[10] A. Lomuscio and B. Woźna. A complete and decidable axiomatisation for deontic interpreted systems. In *Proceedings of the 8th International Workshop on Deontic Logic in Computer Science (DEON'06)*, volume 4048, pages 238–254. Springer-Verlag, July 2006.

[11] W. Penczek and A. Lomuscio. Verifying epistemic properties of multi-agent systems via bounded model checking. *Fundamenta Informaticae*, 55(2):167–185, 2003.

[12] M. Pistore, F. Barbon, P. Bertoli, D. Shaparau, and P. Traverso. Planning and monitoring web service composition. In *AIMSA*, pages 106–115, 2004.

[13] The OWL-S Coalition. OWL-S 1.1 Release., 2004. http://www.daml.org/services/owl-s/1.0/.

[14] M. Wooldridge. *An introduction to multi-agent systems*. John Wiley, England, 2002.

[15] M. Wooldridge, M.-P. Huget, M. Fisher, and S. Parsons. Model checking for multiagent systems: the mable language and its applications. *International Journal on Artificial Intelligence Tools*, 15(2):195–226, 2006.

[16] X. Fu T. Bultan and J. Su. Conversation Protocols: A Formalism for Specification and Verification of Reactive Electronic Services. In *CIAA*, volume LNCS 2759, pages 188–200. Springer-Verlag, 2003.