# Towards verifying contract regulated service composition

Alessio Lomuscio, Hongyang Qu, Monika Solanki
Department of Computing, Imperial College London
{a.lomuscio, hongyang.qu, m.solanki}@imperial.ac.uk

## Abstract

*We report on a novel approach to (semi-)automatically compile and verify* contract-regulated *service compositions. We specify web services and the contracts governing them as WSBPEL behaviours. We compile WSBPEL behaviours into the specialised system description language ISPL, to be used with the model checker MCMAS to verify behaviours automatically. 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 whose state space is approximately* $10^6$ *and discuss experimental results.*

## 1 Introduction

Web services (WS) are now considered one of the key technologies for building new generations of digital business systems. Industrial strength distributed applications can be built across organisational boundaries using services as basic building blocks. When services are combined, a significant challenge is to regulate the business interactions between them. In an environment where previously unknown services are dynamically discovered and binded, their composition is usually underpinned by binding agreements or "contracts". Should a contract be broken by one of the parties, "legal remedies" may be applicable in the form of penalties, additional rights to some party, and, possibly, additional penalties with respect to third parties.

Conventionally, contracts have been defined and interpreted using natural languages. In electronic business environments, new formal models and tools are needed to enable the successful enforcement of dynamic contractual agreements between services. While designing a contract-regulated composition, an important aspect is the rigorous analysis of possible execution behaviours of individual services as well as the overall behaviour of the composition. A system made of few localised services may only interact in a small number of ways governed by a limited set of contract clauses. However when several subsystems coordinate in an open environment, the contracts binding them are non-trivial and complex, making it difficult to forsee all the possible executions. Additionally, while trying to comply to their respective contractually defined behaviours, certain components may fail, some may be incapacitated to provide the services in the expected timeline, and others still may have to prioritise certain requests.

In this paper, we propose a novel approach towards the verification of services, where transactions are controlled by binding electronic contracts. Verification of WS is an active topic of research (e.g., see [16, 18]). However it has so far been concerned with checking safety and liveness properties only. Our proposed framework, builds upon existing work in the domain of multi agent systems (MAS) [17, 1]. We take the view that a web service can be modelled as an "agent" [5]. When WS are phrased as a contract-regulated MAS, several properties become worth studying, including various notions of correctness and violations of the contracts during a run, the evolution of the agents' knowledge about themselves, the contracts and the expected peers' behaviours, etc.

The specification and analysis of agent behaviour in a MAS has been widely explored. Several formal models have been investigated to specify formally and unambiguously the behaviour of the system. Many of these are based on modal logic, including temporal, epistemic, and deotic logic. Developments in verification of MAS via model checking techniques [15, 4, 9] has kept pace with the advancement in the specification techniques. Along with temporal languages, it is now also possible to verify a variety of modalities describing the informational and intentional state of the agents.

The above leads us to explore the verification of contract-based WS implemented by means of MAS model checkers. To this end, we propose a verification methodology where services or "contract parties" (CP) are specified using WS-BPEL [13]. The contractually correct behaviours for every CP are also specified in WSBPEL. In our approach, a compiler of our design takes as input both these behaviour descriptions, and generates an ISPL program, which is fed to the symbolic model checker MCMAS for verification.

The rest of the paper is organised as follows. In Section 2 we briefly introduce WSBPEL, ISPL and MCMAS. Section 3 introduces a motivating example and some of its key properties. Section 4 presents our proposed framework. Section 5 discusses the implementation of the compiler and Section 6 gives experimental results from verification. We conclude in Section 7.

## 2 Preliminaries

### 2.1 MCMAS and ISPL

MCMAS [11] is a specialised model checker for the verification of multi-agent systems. It builds on symbolic model checking via OBDDs as its underlying technique, and supports CTL, epistemic and deontic logic. The current version of MCMAS [10] has the following features: (1) Support for variables of the following types: Boolean, enumeration and bounded integer. Arithmetic operations can be performed on bounded integers. (2) Counterexample/witness generation for quick and efficient display of traces falsifying/satisfying properties. (3) Support for fairness constraints. This is useful in eliminating unrealistic behaviours. (4) Support for interactive execution mode. This allows users to step through the execution of their model.

MCMAS uses ISPL as its input language. A system encoded in ISPL is composed of the environment $e$ and a set of agents $A = \{1, \ldots, n\}$. Each agent $i \in A$ has a set of *local states* $L_i$ and a set of local actions $Act_i$. The *protocol function* of agent $i$, $P_i : L_i \to 2^{Act_i}$, defines for each local state $l_i \in L_i$ the set of actions that are allowed to be executed in $l_i$. Similarly, the environment has its local states $L_e$, local actions $Act_e$ and protocol function $P_e$. The transition relation among local states of agent $i$ is defined by the *evolution function* $Ev_i : L_i \times Act_1 \times \cdots \times Act_n \times Act_e \to L_i$. The definition of $Ev_i$ suggests that the local actions of an agent can be observed by other agents. The evolution function $Ev_e$ of the environment is defined in the same way.

To reason about the behaviours of agent $i$ with respect to correctness [12], $L_i$ is further partitioned into two disjoint sets: a non-empty set $G_i$ of allowed (“green”) states and a set $R_i$ of disallowed (“red”) states. In this paper, we use green states to denote the behaviours in compliance with contracts and red states to denote violations, by means of temporal epistemic properties.

ISPL allows user defined atomic propositions over *global states* of the system. A global state is composed of a local state from every agent and the environment. The logic formulae to be checked by MCMAS are defined over the atomic propositions.

### 2.2 WSBPEL

WSBPEL [13] is a popular and de facto industrial standard for describing service composition. The specification has been elaborately explained in several web service based literature [13]. is highly recommended.

WSBPEL defines a model and an XML based grammar for the orchestration of executable and abstract business processes. A BPEL process defines the interaction between partners. The specification provides the control logic to coordinate arbitrarily complex web services, defined in WSDL. A BPEL process can interact synchronously or asynchronously with its partners, i.e., its clients, and with the services the process orchestrates.

The building blocks for a BPEL process are the descriptions of the parties participating in the process, the data that flows through the process and the activities performed during the execution of the process. Some examples of activities include “receive”, “reply”, “assign”, “sequence” and “wait”. WSBPEL also introduces systematic mechanisms for dealing with business exceptions and processing faults. Moreover, WSBPEL introduces a mechanism to define how individual or composite activities within a unit of work are to be compensated in cases where exceptions occur or a partner requests reversal.

## 3 A Motivating Case Study

In this section we present a composition of services, regulated as a pre-defined contract. The case study was first presented in earlier work on verifying service composition with MCMAS [1]. Here, we focus on the automatic compilation of services from WSBPEL into ISPL.

In the example, the participating contract parties, as illustrated in Figure 1, comprise: 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 ($E$). The high-level workflow of the composition is defined as follows: Client $C$ wants to get a software developed and 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 $SP$ delivers its component. $PSP$ and $SP$ twice update each other and $C$ about the progress of the software development. Should the client like any 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.

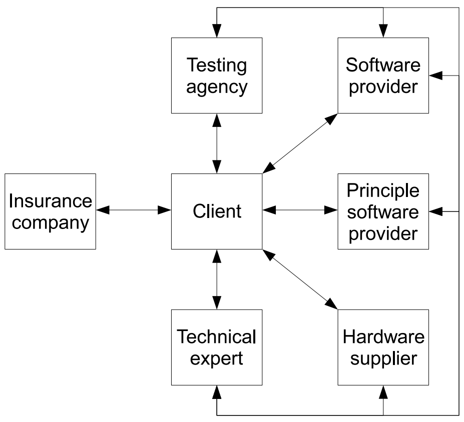

**Figure 1. Interaction between various partners in the composition.**

***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.

**Figure 2. Obligations of Contract parties**

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 integration test fails, the components are revised and tested again. Components can be revised twice. If the third test fails, $C$ 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 $H$. Figure 2 illustrates the obligations of the $PSP$ and $C$.

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 be-

| | 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 |

**Figure 3. Agents and their violation conditions.**

haviour identified in the contract is considered a violation. The contract might in some cases also specify mechanisms for recovering from violations.

The contract between various parties can be violated in many ways. Figure 3 illustrates informally some of the conditions under which some local violations may occur.

## 4 Verification framework

In this section we discuss our framework for the verification of contracts. Figure 4 illustrates the proposed architecture. Our approach targets two levels of verification:

- conformance of the behaviour of an individual contract party to its contractually correct behaviour.
- conformance of the combined behaviour of all the contract parties to the overall contract.

For the sake of clarity in the figure and the paper, we elaborate on the components of the architecture and the verification methodology, only for contract party $C_1$. Note that a similar mechanism would be replicated for all the contract parties in the composition.

1. **Natural language contracts:** Conventionally contracts are specified in a natural language. A contract

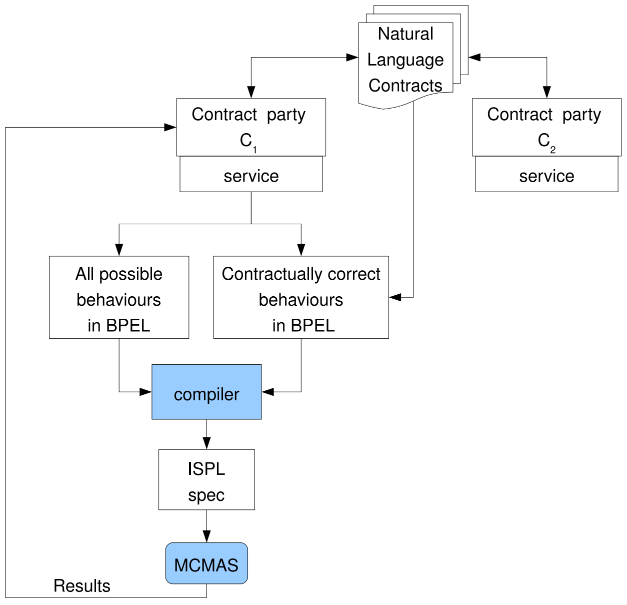
Natural Language Contracts
Contract party C1
service
Contract party C2
service
All possible behaviours in BPEL
Contractually correct behaviours in BPEL
compiler
ISPL spec
MCMAS
Results

**Figure 4. Verification architecture**

stipulates the obligations of parties entering the contract. It defines behaviours that are considered to be violation of some obligations, and may outline penalties and/or recovery actions from the violations. For verification, a conventional contract is encoded as an *e-contract* in WSBPEL.

2. **Contract party:** A contract party (CP) is a service, that is a first class citizen of the contract regulated composition. The behaviour of a CP is governed by the rights, obligations and violations stipulated in the contract, and agreed to by the CP. The overall fulfillment of a contract depends on the adherence of each CP in the composition to its specified behaviour. In our framework, each contract party is an *agent* with well defined *green* and *red* states corresponding to states of compliance and violation respectively. Our proposed methodology aims to verify the adherence of each agent's behaviour to what has been specified as contractually correct behavior for the agent.

3. **Contract party/agent behaviour:** The behaviour of an agent can be defined in terms of a two-part behaviour: all possible behaviours and contractually correct behaviours. In order to automate the verification, we encode both these behaviours in BPEL. Note that it is possible to describe contractually correct behaviours using a specification language, tailor-made for describing contracts e.g., [14]. However, keeping both these behaviours at the same level of abstraction, provides the system designer with the flexibility needed to combine and compile the behaviours into a model suitable for verification.

   For an agent, we refer to its all possible behaviours as *BPEL-behaviour* and the contractually correct behaviours as its *BPEL-contract*. Note that both the behaviours are inter-dependent and replicate information such as variable and action description for the agent, in their specification.

4. **Compiler:** The compiler is a novel and integral component of our architecture. The compiler takes as input the BPEL-behaviour and the BPEL-contract for an agent and combines them to generate an ISPL program. The compiler parses the BPEL-behaviour to generate a partial model that enumerates the local states but abstracts from defining red and green states. The BPEL-contract is then parsed to enumerate the green/red states for the agent. The internal details of the compiler are illustrated in Section 5.

5. **ISPL and MCMAS:** The ISPL program compiled semi-automatically from the BPEL specification, encodes the overall and desired behaviour of an agent. The program is fed to MCMAS for verification of the agent's behaviour.

## 5 Implementation

The core component of our framewrok is the compiler that translates a WSBPEL specification into an ISPL program. It generates basic atomic propositions and properties automatically for verification. The internal architecture of the compiler is illustrated in Figure 5.

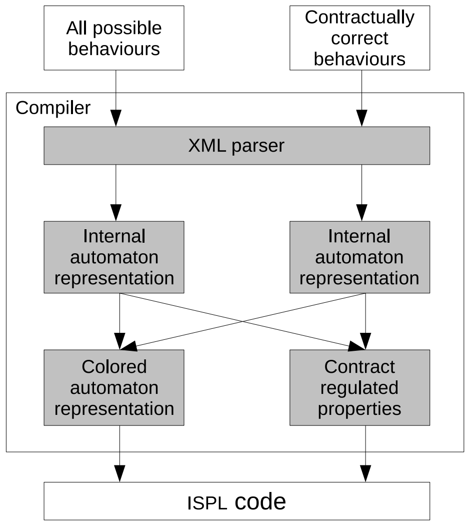
All possible behaviours
Contractually correct behaviours
Compiler
XML parser
Internal automaton representation
Internal automaton representation
Colored automaton representation
Contract regulated properties
ISPL code

**Figure 5. Internal architecture of the compiler**

Given the two specifications (BPEL-behaviours and BPEL-contracts), we propose a three step methodology to generate the corresponding ISPL program:

1. We represent a BPEL process by an automaton. The BPEL-behaviour is first read into memory, followed by the BPEL-contract. Both behaviours are translated into automata. We use *behaviour automata* to denote the automata representing the BPEL-behaviour and *contract automata* for the BPEL-contract.

2. For each state in the contract automata, we look for its counterpart in the behaviour automata and label it as *green*. We then label all other states in the behaviour automata as *red*. Based on these labels, basic properties specified in temporal-epistemic logic are generated.

3. The labelled behaviour automata and the properties are written to the ISPL file input to the checker.

In what follows, we discuss the methodology in detail.

### 5.1 Translating BPEL programs into automata

The compiler uses the following rules to do the translation.

- "Assign", "receive", "invoke" and "empty" activities are translated into transitions connecting the respective *source state* and *target state*. A "sequence" activity is translated into a sequence of transitions.

- An "if" activity is translated into two sequences of transitions, one for the *if*-branch and another for the *else*-branch. The first transition in the *if*-branch uses the condition in the "if" activity as its guard, while the first transition in the *else*-branch uses the negation of the condition as the guard. A "while" activity is translated in the same way as an "if" activity except that the target state of the last transition and the source state of the first transition in the *if*-branch are the same.

- "OnMessage" activities and "onAlarm" activities in a "pick" activity are translated into transitions with a common source state.

- A branch in a "flow" activity is translated into a separate automaton. The beginning and the end of these automata are synchronised with the automaton representing the BPEL process. In doing so, we differ from [8], where a "flow" is translated such that: all branches are executed sequentially and all possible permutations are represented as a single automaton.

  The automata generated from "if", "while", "pick" and "flow" activities are illustrated in Figure 6.

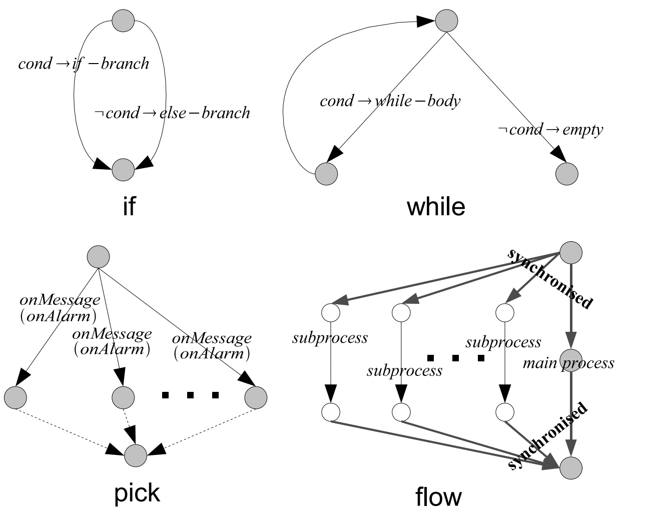

**Figure 6. Translated automaton**

- Fault handlers and exceptions are translated into transitions as well. The latter transition assigns a specific value to a variable and the guard of the former transition tests if the variable has this value. Other kinds of handlers are dealt with in the same way. Theoretically, in every state where an exception could happen a copy of the exception/handler transition is produced using this state as its source state (note that these copies have the same target state). Thus one transition would be replicated many times. In practice, however, we have a succinct way to implement it due to the flexibility of ISPL, as discussed later.

As remarked in the literature review, much work has been appeared on translating BPEL into model checkers' input languages, e.g., [8, 7, 2]. However, only few of them can process all BPEL structures. A detailed discussion can be found in [2].

### 5.2 Colouring the model

We use the green and red of labelling in ISPL code to differentiate between contractually correct and incorrect behaviours, as shown in Figure 7. This is possible be-

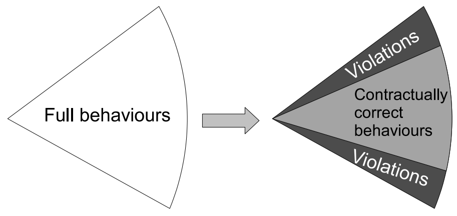

**Figure 7. Labelling behaviours**

cause the BPEL-contract specification defines behaviours included also in the BPEL-behaviour specification. Labelling the states in the behaviour automata is done as follows:

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.

We do not look for matched states directly because the states are named in a numerical way and, therefore, the same state in the behaviour automata and the contract automata might have different names. However, transitions get their name from the BPEL activities, each of which has a unique name.

After the labelling process finishes, the compiler encodes three kinds of atomic propositions, which are used to define basic formulae to be checked in the following way. For each BPEL process $p$, we define

- an atomic proposition $p_{green}$ holding in all green states of the process;
- an atomic proposition $p_{end}$ holding in the last state of process $p$;
- an atomic proposition $p_{red_i}$ holding in the corresponding red state $i$.

Two kinds of basic properties are generated based on the atomic properties. For each BPEL process $p$, define

$$E\ (p_{green}\ U\ p_{end}). \tag{1}$$

This property specifies that $p$ has a way to conduct a whole run in compliance with its contract obligations. For each atomic proposition $p_{red} \in \{p_{red0}, p_{red1}, \ldots\}$, define

$$EF\ p_{red}. \tag{2}$$

This property represents a test to check whether a agent may violate its contractual behaviours.

The above properties verify the basic behaviours of contract parties. More properties can be manually added to the automatically generated ISPL code in order to test other interesting behaviours (see below).

## 5.3 Generating an ISPL program

Once the behaviour automatas are labelled, they are ready to be written to an ISPL file for verification. Each automaton is mapped to an agent in the file. Let $\mathcal{A} = \{1, \ldots, n\}$ be the set of automata and $A = \{1, \ldots, n\}$ the set of agents. Here we only enumerate the key steps to generate an agent $i \in A$ from an automaton $\mathcal{A}_i \in \mathcal{A}$.

1. Local states generation. A local state $l \in L_i$ is a valuation for the set of *local variables* $Var_i$. Thus, the generation of $L_i$ is performed through the generation of $Var_i$. If $\mathcal{A}_i$ is generated from a BPEL process $p$, then
$$Var_i = Var_p \cup \{state\},$$
where $Var_p$ is the set of variables defined in $p$ and $state$ is an additional enumeration variable. Each value of $state$ represents a unique state of $\mathcal{A}_i$. If $\mathcal{A}_i$ is a "flow" branch in $p$, then
$$Var_i = Var'_p \cup \{state\},$$
where $Var'_p \subseteq Var_p$ is the set of variables used by $\mathcal{A}_i$. In order to reduce the agent's state space, the compiler monitors the usage of every variable $v \in Var_p$. If $v$ is never read by any transitions in $\mathcal{A}_i$, then it is discarded.

2. Local actions generation. $Act_i$ is obtained from the transitions of $\mathcal{A}_i$. Each transition is mapped into an action; additionally if two transitions have the same name, they are mapped into the same action.

3. Protocol generation. Let $l(state)$ be the value of variable $state$ in state $l \in L_i$ and $E_l$ the set of allowed actions in $l$. For any transition $t$ whose source state is represented by $l(state)$, the action to which $t$ is mapped is included in $E_l$. Obviously, two states $l_1, l_2 \in L_i$ have the same set of allowed actions if $l_1(state) = l_2(state)$.

4. Evolution function generation. Each transition in $\mathcal{A}_i$ is translated to an evolution item. For a transition $t$ with source state $s_1$, target state $s_2$, and guard $c$, the evolution item is defined to be of the following form:

   state=$s_2$ if state=$s_1$ and c and Action=t.

   This item means that if in the current state, the variable $state$ has value $s_1$ and the guard $c$ is satisfied, the execution of $t$ makes agent $i$ move to a state where $state$ has value $s_2$. If $t$ is synchronised with another transition $t'$ in the automaton $\mathcal{A}_j \in \mathcal{A}$, then the evolution item is

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

   If $t$ assigns a value $expr$ to a variable $v$, the assignment is translated on the left side of "if", i.e.

   state=$s_2$ and v=$expr$ if $\cdots$.

   If there are multiple copies of $t$, e.g., $t$ represents a fault handler, we use the following form to specify an evolution item for all copies:

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

where $s_1$ and $s_2$ are the source states of these copies and $s$ is their target state. If $t$ is allowed in all states, the above form can be simplified to

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

## 6 Experimental Analysis

We evaluated the compilation and verification mechanism on the case study illustrated in Section 3. We represented the composition in terms of a WSBPEL orchestration. The following BPEL code represents the full behaviour of the client $C$, when receiving updates from $PSP$ and $SP$. Note that for brevity, only essential information is shown. The BPEL-contract is the same as BPEL-behaviour except that it defines only contractually correct and therefore limited behaviours.

```
<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>
```

The translation generates the following ISPL program for the client.

```
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
```

The following listing gives an example about how to define atomic propositions and properties in ISPL.

```
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
```

In addition to the basic properties automatically generated by the compiler, we manually added a few more complex properties to the model. Those properties were also studied in [1]. Some atomic propositions, e.g., "receiveSoftware" and "softwareTested", are also added to the ISPL code manually. In particular, we considered the following:

- 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.

  $$E(C\_green\, U$$
  $$E((C\_green \wedge softwareDeployed)\, U\, \neg C\_green))$$

The generated ISPL model was encoded automatically by MCMAS by using 134 BDD variables: 49 BDD variables for local states (the same number of BDD variables are constructed for the transition relation) and 36 for local actions. The total number of global states is approximately

$10^6$. On a machine running Linux Fedora 8 x86_64 version (kernel 2.6.24.3-50) on Intel Core 2 Duo E4500 2.2GHz with 4GB memory, it took about 24 seconds with 34 MB memory space for MCMAS to verify 25 properties.

In this example, all basic properties hold on the model, which means not only all parties can fulfil their contractual obligations successfully, but also that all the violations shown in Figure 3 can actually happen. Amongst the manually added properties, the first one does not hold. The reason is that even though $PSP$ fulfills its contractual obligations, the software might not pass testing hence not be deployed. For a similar reason, the third one does not hold either.

## 7 Conclusions

In this paper we presented a novel technique for the verification of contract-regulated service compositions. In our approach, services and contracts are specified as WSBPEL behaviours. We showed how these behaviours could be semi-automatically compiled into ISPL, and then verified using the symbolic model checker MCMAS. The salient feature of the approach is the possibility of checking agent compliance with respect to contracts and the potential of compiling a fairly large subset of BPEL constructs to ISPL. We illustrated the methodology using a realistic case study with a reasonably large state space.

It is worth mentioning that there are two limitations in the current framework: (1) Since MCMAS cannot handle real-time systems, some BPEL constructs such as *deadline* and *timeout* have to be translated into non-deterministic behaviours. For real-time properties, a secondary model checker, such as UPPAAL [3] or Verics[6], can be integrated into the framework. (2) The contracts that can be dealt with are written in natural languages and translated into BPEL code manually. Nowadays, some contracting languages, e.g., [14], have been proposed in order to construct electronic contracts to be processed by computers. Currently, we are working on compiling electronic contracts into ISPL to allow more automation.

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

## References

[1] A. Lomuscio and H. Qu and M. Solanki. Towards verifying compliance in agent-based web service compositions. In *Proceedings of The Seventh International Joint Conference on Autonomous Agents and Multi-agent systems (AAMAS-08).* ACM Press, 2008.

[2] L. Baresi, D. Bianculli, C. Ghezzi, S. Guinea, and P. Spoletini. Validation of web service compositions. *IET Softw.*, 1(6):219–232, December 2007.

[3] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, W. Yi, and C. Weise. New generation of UPPAAL. In *Proceedings of the International Workshop on Software Tools for Technology Transfer*, 1998.

[4] 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.

[5] 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/.

[6] P. Dembiński, A. Janowska, P. Janowski, W. Penczek, A. Pólrola, M. Szreter, B. Woźna, and A. Zbrzezny. VerICS: A tool for verifying Timed Automata and Estelle specifications. In *Proc. of the 9th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *LNCS*, pages 278–283. Springer-Verlag, 2003.

[7] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *Proceedings of the 10th IEEE International Conference on Automated Software Engineering*. IEEE Press, 2003.

[8] 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.

[9] P. Gammie and R. van der Meyden. MCK: Model checking the logic of knowledge. In *Proceedings of 16th International Conference on Computer Aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 479–483. Springer-Verlag, 2004.

[10] A. Lomuscio, H. Qu, and F. Raimondi. Mcmas 0.9 alpha. http://sourceforge.net/projects/ist-contract/, 2008.

[11] 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.

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

[13] OASIS Web service Business Process Execution Language (WSBPEL) TC. Web service Business Process Execution Language Version 2.0, 2007.

[14] S. Panagiotidi, J. Vazquez-Salceda, S. Alvarez-Napagao, S. Ortega-Martorell, S. Willmott, and P. S. R. Confalonieri. Contracting agent language. In *Symposium on Behaviour Regulation in Multi-Agent Systems*, 2008.

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

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

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

[18] 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.