Towards an Agent Based Approach for Verification of OWL-S Process Models

Alessio Lomuscio and Monika Solanki

Department of Computing, Imperial College London, UK
{a.lomuscio,m.solanki}@imperial.ac.uk

Abstract. In this paper we investigate the transformation of OWL-S process models to ISPL - the system description language for MCMAS, a symbolic model checker for multi agent systems. We take the view that services can be considered as agents and service compositions as multi agent systems. We illustrate how atomic and composite processes in OWL-S can be encoded into ISPL using the proposed transformation rules for a restricted set of data types. As an illustrative example, we use an extended version of the BravoAir process model. We formalise certain interesting properties of the example in temporal-epistemic logic and present results from their verification using MCMAS.

1 Introduction

The verification of web service behaviour and interaction protocols is now an integral aspect of several frameworks providing service oriented solutions to the IT industry. The Increasing complexities that arise during service composition, make offline verification as model checking [4] crucial in successfully implementing and using services. Model checkers typically use specialised formats for the specification of behaviour, different from those commonly used for describing services. Examples of such system description languages include Promela, used with the checker SPIN [9] and NuSMV, used with the checker NuSMV [3]. However in the web service domain, WSBPEL [16], WSCDL [15] and OWL-S [17], are some popular and widely used standards for describing service behaviour, their composition and interaction protocols.

The languages above work at different levels of abstraction. In order to verify services, an important first step is to investigate how the input language to the model checker can be adapted to encode a suitable abstraction of the service behaviour, which has been described using one of the above standards. Figure 1 illustrates the general architecture of a verification framework for services. As highlighted, a crucial component is the "compiler" that takes as input the service specification, and generates a suitably abstracted model/program, encoded in the system description language for the checker. The model and the desired properties to be verified are fed to the checker. By performing a systematic exploration of the complete set of states that can be generated during an interaction between a service and its clients, the model checker is able to verify desirable properties of the composition. Generating a well abstracted model is thus crucial to the verification of services. However, developing a tool that generates such

L. Aroyo et al. (Eds.): ESWC 2009, LNCS 5554, pp. 578–592, 2009.

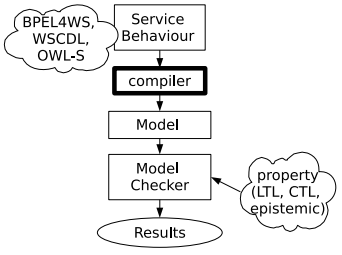

Fig. 1. General architecture for service verification

a model is non trivial. Mapping rules between the languages are required to be established before any automated translation can be undertaken. The rules provide the basis and rationale for development of a compiler providing (semi)automatic compilations from one abstraction to the other.

In this paper, we explore the generation of transformation rules from the process model of OWL-S, a well-established language for the description of web services on the semantic web, to ISPL (Interpreted Systems Programming Language) in view of verifying the results using MCMAS [13]. We are interested in using MCMAS, because it enables the user to verify rich specifications. MCMAS supports not only temporal logic, but also epistemic and deontic modalities. We take the view that a web service can be modelled as an "agent" [6]. Keeping this in perspective, a composition of web services can be viewed as a multi agent system [19]. An OWL-S process model specifies the composition and interaction between agents/services and their clients. Control constructs similar to those found in programming languages can be used to compose services. Earlier work [14,1] on using MCMAS for services has focused on exploiting its verification capabilities. In this paper we provide the transformation rules and a compiler implementation of the rules, that combined with MCMAS, could aid in automated verification of services.

The paper is structured as follows: Section 2 outlines our running example for the paper, a flight booking and managing service, which is an extended version of the BravoAir process from the OWL-S suite of examples. In Section 3 we provide a brief overview of OWL-S, ISPL and MCMAS. Section 4 discusses the mapping rules from OWL-S to ISPL and presents our implementation of the compiler. We present a brief account of the analysis and verification for the case study in Section 5. Finally, we conclude in Section 6.

2 Case Study

As a running example, we use an extended version of the BravoAirProcess model from the OWL-S suite of examples. BravoAir functions as a flight booking agent. It allows a client to perform several tasks such as searching, selecting and booking flights. Bookings can be made as individual or as groups. The top level process, `BravoAir`, is a composite process. It is composed of a *sequence* of processes. Components of the sequence are `GetDesiredFlightDetails` and `SearchAvailableFlight`, and

a composite process, BookFlight. We extend the BookFlight process as a *sequence* whose components are LogIn, followed by a choice between Individual Bookng and GroupBooking and finally ConfirmReservation. Group bookings can be done for a group of more than 10 people and the discount offered is 10% of the total booking fee. However when a group booking is cancelled, the cancellation fee is 15% rather than 10% for individual bookings. We elaborate on the GroupBooking process in Section 4.

We further extend the top level process with a composite process ManageBooking, to be executed in *sequence* with BookFlight. ManageBooking is composed as a *choice* between the atomic process ChooseSeats and the composite processes, ChangeBooking and CancelBooking. If a booking is cancelled the amount which is charged for cancellation depends on whether the booking was made at an individual or group level. ChangeBooking is composed of a *split+join* whose components are the three atomic processes ChargeCard- for economy bookings, AllocateNew Booking- for club and business class bookings and SendUnAvailability- when a change of booking is not possible. The outcome from a choice between the first two processes is composed in *sequence* with SendConfirmation. ChargeCard is also invoked when a booking is cancelled. Figure 2 illustrates the various control constructs and processes used in the composition between BravoAir and a potential client. The composition can be viewed as a multi-agent system where individual processes are abstracted as agents. Within the above settings certain interesting properties of the composition can be verified. We enumerate some of these below and formalise them in temporal-epistemic logic in Section 5.

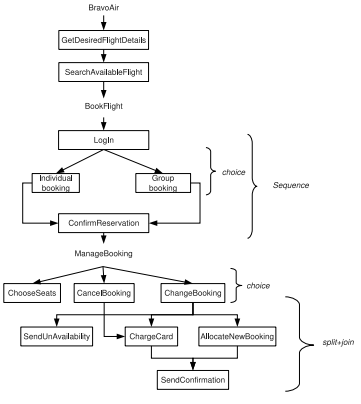

Fig. 2. An extended version of the BravoAir process

- if there is a request confirmation of reservation, the ConfirmReservation agent knows that the booking has been successful and payment has been made.
- If the number of people is more than 10, group booking will always be successful.
- Whenever a booking change is requested, it will always be confirmed.
- If a card is not charged when a booking is changed, the ChangeBooking agent knows that the reference is a business class booking.

- if a confirmation is received, the Customer agent knows that his booking was changed.
- If a card is charged after a booking has been made, it always implies that the booking has been cancelled.

3 Preliminaries

OWL-S [17] is an OWL-based Web service ontology which supplies Web service providers with a core set of constructs for describing the properties and capabilities of their Web services. It defines an upper ontology for services with *Service Profile*, *Service Model* and *Service Grounding* models. The ServiceProfile provides the information needed for an agent to discover a service, while the ServiceModel and ServiceGrounding, taken together, provide enough information for an agent to make use of a service, once found. Due to space restrictions we do not present any further details of the language here and refer the interested reader to [17].

ISPL (Interpreted System Programming Language) is based on the formal semantics of interpreted systems [7] and is the input language for the model checker MCMAS [13,12]. The syntax of ISPL includes the following

- The definition of agents describes the local behaviour of every agent in terms of states, actions, protocols and evolution function. Each agent has a set of local variables. The states of an agent, each of which contains a valuation of its local variables can be further partitioned into two disjoint sets: a non-empty set of allowed ("green") states and a set of disallowed ("red") states. Currently, ISPL allows three types of variables: Boolean, enumeration and bounded integer.
- The global evaluation function of the system defines atomic propositions held over global states which are a combinations of local states of agents defined in the model.
- The local initial state for each agent in the system.
- Specification to be checked defined as formulae in temporal, epistemic and deontic logic and fairness formuale.

MCMAS 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 has the following features:

- Support for variables of the following types: Boolean, enumeration and bounded integer. Arithmetic operations can be performed on bounded integers.
- Counterexample/witness generation for quick and efficient display of traces falsifying/satisfying properties.
- Support for fairness constraints. This is useful in eliminating unrealistic behaviours.
- Support for interactive execution mode. This allows users to step through the execution of their model.
- A graphical interface provided as an Eclipse plug-in which includes a graphical editor with syntax recognition, a graphical simulator, and a graphical analyser for counterexamples.

4 Encoding OWL-S Processes as ISPL Models

In an OWL-S process model, inputs and outputs are process parameters that have concrete datatypes. The current version of ISPL provides support for variables of types *bounded integers*, *boolean* and *enum*. Mapping between ontologies and these types can be done as discussed in [11]. Most existing model checkers including MCMAS, are not equipped to support OWL object types as also highlighted in [2]. We therefore abstract from defining object types for the transformation presented in this paper.

Conditions explicitly occur in OWL-S models as `Precondition`, and as part of the `Result` and `if` statement. In ISPL conditions are defined as formulae when specifying the protocol and evolution functions. In this section we first propose the following step-by-step methodology for transforming an atomic process to an ISPL program. We then show how composite processes can be transformed. These rules also facilitate the generation of a semi-automatic compiler from OWL-S to ISPL.

4.1 Encoding Atomic Processes

Agent. For every atomic process in OWL-S, we define an agent, qualified as `ProcessName` in ISPL. Recall, that the definition of an agent in ISPL includes: local variables, red states, actions, protocol functions and evolution functions.

Variables and Local States. The local states of an agent in ISPL are defined in terms of valuation of the local variables. We define the set of local variables for an agent by transforming the ontological inputs and outputs in the process model, to variables with the same identifiers and datatypes in the ISPL model. Bounds for integer variables are interactively assigned keeping the domain and context of the process model in perspective. For an atomic process, we identify two kinds of states: (1) An "Input" or initial state and (2) several "Result" states depending on the number of results defined for the process.

Let V_I denote the set of integer variables, V_B the set of Boolean variables and V_E the set of variables of type enum. We define the set of their valuations as Val_I, Val_B and Val_E respectively. The set of local variables for an agent is therefore $V = V_I \cup V_B \cup V_E$. The set of local states, L_{lstate}, of an agent can now be defined as

$$L_{lstate} : (V_I \rightarrow Val_I) \cup (V_B \rightarrow Val_B) \cup (V_E \rightarrow Val_E).$$

We enumerate L_{lstate} for an agent as follows -

- the initial state (l_0) where local variables are assigned initial values. We denote the set of variables at the initial state as $V_0 \subseteq V$.
- the set of states L_{result}, where each $l \in L_{result}$ corresponds to a non deterministic `Result` state, defined for the process. For example, a credit card validating service may produce two results: *ValidationSuccess* with boolean output $validated$ as $true$, and $ValidationnFailed$ with boolean output $validated$ as $false$. The set of variables at each of the result state is denoted as $V_i \subseteq V$, $i = 1 \ldots \mid L_{result} \mid$. The valuations for the variables are computed as per the evolution function described below.

– finally, l_f, a failure state which is reached when the preconditions for the process evaluates to $false$. We denote the set of variables at this state as $V_f \subseteq V$. Valuations for the variables are again computed as per the evolution function.

Red States. They are reached when an agent performs an undesirable action. This feature of ISPL is most useful while encoding faults and recovery in complex systems. The red states of an agent are represented by a Boolean formula, f^{red}, over its local variables.

Actions. The internal actions taken by a service cause a transition from the input state to one of the several result states. Actions for the agent are enumerated as follows:

– the null action ϵ,
– the set of internal actions, $A_{int} = \{a_i | i = 1 \ldots n\}$, the agent takes at the input state to reach one of the several result states.
– the internal action a_f taken when the precondition fails, to reach state l_f.
– the set of actions $A_{send} = \{s_i | i = 1 \ldots n\}$. The agent takes an action $s \in A_{send}$ at each $l \in l_{result}$ respectively to send the corresponding results to the client.
– the action s_f which the agent takes to send the precondition failure message at l_f.
– It follows that the total number of actions is:

$$N_{actions} =\mid A_{int} \cup \{a_f\} \mid + \mid A_{send} \cup \{s_f\} \mid + \mid \{\epsilon\} \mid$$

For simplicity we assume, $\mid A_{int} \cup \{a_f\} \mid = \mid A_{send} \cup \{s_f\} \mid$, and simplify the above to,

$$N_{actions} = 2\times \mid A_{int} \cup \{a_f\} \mid +1$$

Protocols. Protocols for the agent can be enumerated as follows:

$$\begin{aligned} f^{pre} &: \{a_i | i = 1 \ldots \mid A_{int} \mid\} \\ !f^{pre} &: a_f \\ f_i^{res} &: \{s_i | i = 1 \ldots \mid A_{send} \mid\} \cup s_f \end{aligned}$$

where f^{pre} (precondition), f_i^{res}, $i = 1 \ \ldots \ \mid A_{send} \mid$ (condition in results) and $!f^{pre}$ are Boolean formulae over the set of local variables at the input state, result states[1] and the failure state respectively. Note that ISPL and MCMAS allow non determinism in the specification of protocols.

At execution time an agent at l_0 takes an action, $a \in A_{int}$ if f^{pre}, i.e., the precondition holds and action a_f if $!f^{pre}$ holds. This causes a transition to one of the result states $l \in L_{result} \cup l_f$, where the conditionals from the results, f_i^{res}, $i = 1 \ldots \mid A_{send} \mid$ are required to hold. At l, the agent take an action, $s \in A_{send} \cup \{s_f\}$.

[1] Note that for simplicity we do not consider the case, when due to some internal failure of the service, the result conditions do not hold, but this may well be possible and additional transitions would have to be defined to consider such scenarios.

Evolutions (Transitions). The evolution function determines how local states evolve based on the agent's current local state and a set of actions. An evolution consists of a set of assignments of local variables in V and an enabling condition which is a Boolean formula, over local variables and actions of all agents.

l_0 if f_i^{res} and ProcessName.Action $= s_i$ or ProcessName.Action $= s_f, i = 1 \ldots \mid A_{send} \mid$
l_i if f^{pre} and ProcessName.Action $= a_i, i = 1 \ldots \mid A_{int} \mid$
l_f if $!f^{pre}$ and ProcessName.Action $= a_f$

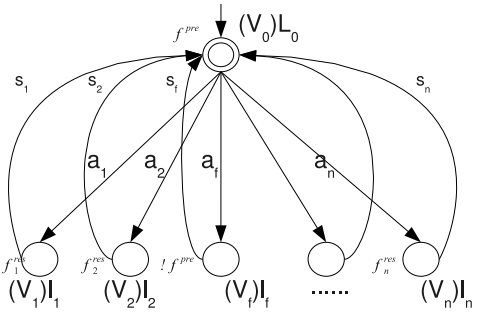

Fig. 3. Mapping between an OWL-S atomic process and ISPL

We have implemented a semi automatic compiler for the transformation, by extending the CMU OWL-S API [5]. Given an OWL-S process model, the compiler extracts the agent name, inputs and outputs from the process model as ISPL variables and enumerates the actions for the agent. Currently the definition of the red states, protocol and evolution function are interactively given, but we hope to automate the process in future versions of the paper.

The psuedocode of an algorithm, which we implemented as part of our compiler for compiling an atomic process to ISPL is presented as algorithm 1 towards the end of the paper.

4.2 ISPL Encoding of the Atomic Process: GroupBooking

Figure 4 illustrates an atomic process "Group Booking " from the case study presented in Section 2. The process takes $noOfPeople$, $flightdetails$, $carddetails$ and $loggedInStatus$ as inputs. In order to perform a group booking, the preconditions on the process are that the payment card details must be provided, the number of people in a group must be atleast 10 and the booking client must be a logged in. It returns as output, a $successMsg$ message, an $invalidCardMsg$ or an $invalidNumMsg$ message depending on the conditions $isBookingSuccessful$, $isValidCard$ and $isValidNumberOfPeople$ being $true$ or $false$. For a successful booking it also returns the discounted booking cost.

We specify the "GroupBooking" agent using the presentation syntax of OWL-S along with its corresponding ISPL code in Table 1. The inputs and outputs are mapped

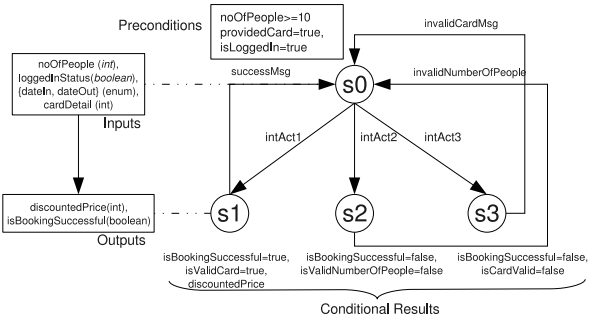

Fig. 4. Atomic Process: Group Booking

Table 1. The Group Booking Atomic Process

OWL-S atomic process	ISPL Agent
define atomic process Group (inputs:(noOfPeople - xsd:integer cardDetails - xsd: string dout - xsd:date din - xsd:date), exists:(loggedInStatus - xsd:boolean), preconditions :((loggedInStatus) & (noOfPeople>=10) & providedCard(cardDetails), outputs:(invalidCardMsg - xsd:string successMsg - xsd:string invalidNumMsg -xsd:string discountedPrice - xsd:String) results :(isValidCard & noOfPeople>=10)|-> output(isBookingSuccessful - xsd:boolean, discountedPrice - xsd:integer), isValidCard |-> output(invalidCardMsg-xsd:string), isValidNumberOfPeople |-> output(invalidNumMsg),)	Agent GroupBooking Vars: noOfPeople:1..20; isValidNumberOfPeople:boolean; loggedIn:boolean; providedCard:boolean; isValidCard:boolean; isBookingSuccessful:boolean; price:1000..200000; discountedPrice:100..20000; dates:{dout, din}; successMsgSent:boolean; cardFailureMsgSent:boolean; numberFailureMsgSent:boolean; end Vars RedStates: end RedStates Actions = {intAct1, intAct2, intAct3, intAct4, invalidCardMsg, invalidNumMsg, successMsg, nothing}; Protocol: loggedIn=true and providedCard=true and noOfPeople >=10 : {intAct1, intAct3}; loggedIn=true and noOfPeople <10 : {intAct2}; isValidCard=false:{invalidCardMsg}; isValidNumberOfPeople=false:{invalidNumMsg}; isBookingSuccessful=true:{successMsg}; end Protocol Evolution: isBookingSuccessful=true and isValidCard=true and discountedPrice=price -(price * 1/10) if loggedIn=true and providedCard=true and noOfPeople>=10 and GroupBooking.Action=intAct1; isBookingSuccessful=false and isValidCard=false if providedCard=true and GroupBooking.Action=intAct3; isBookingSuccessful=false if noOfPeople<=10 and GroupBooking.Action=intAct2; successMsgSent=true if isBookingSuccessful=true and GroupBooking.Action=successMsg; cardFailureMsgSent=true if isBookingSuccessful=false and GroupBooking.Action=invalidCardMsg; end Evolution end Agent

as "Vars" in ISPL. Actions are interactively enumerated in accordance with the operation names specified in the grounding model defined for the process. Preconditions such as $loggedIn$, $providedCard$ and $noOfPeople \leq 10$ are specified as Boolean formula on the LHS of the protocol function. For the precondition, $loggedIn = true$ and $providedCard = true$ and $noOfPeople \geq 10$ the internal actions $intAct1$ or $intAct3$ would be non deterministically chosen by MCMAS as specified in the protocol function. The conditional part of results is specified as Boolean formula on the LHS of the "if" in the evolution function.

4.3 Encoding Composite Processes

A composite process may use one of several control constructs such as *sequence*, *if-then-else*, found in programming languages. In what follows, we discuss the modelling of composite processes, for some of the control constructs. Transformation to other constructs follows intuitively from those presented below.

Sequence. The *sequence* specifies a list of processes to be executed in a certain order. The modelling of OWL-S sequence requires explicit synchronisation. In ISPL, the definition of evolution for an agent encodes this synchronisation. We illustrate sequential composition through an example of processes composed in sequence.

Consider the process `BookFlight`, from the BravoAir model, which is a sequential composition of three atomic processes, `Login`, `GroupBooking` and `ConfirmReservation`. After receiving the result of a successful booking from the `GroupBooking` process, the client invokes the `ConfirmReservation` process with inputs *isbookingSuccessful=true* and *confirmFlight=true*. The precondition for the execution of `ConfirmReservation` is *isbookingSuccessful=true*. Note that this was also the result condition of the `GroupBooking` process. The `ConfirmReservation` process returns a single result as a complex message consisting of a *reservationID* and *seatNumber*. The processes are synchronised for these inputs on the final state of `GroupBooking` and the initial state of `ConfirmReseravation`. It may also be the case that the client provides all the inputs for both the processes in the initial state of the `GroupBooking` process. In such a scenario the `GroupBooking` process invokes the `ConfirmReseravation` process at its final state using those inputs. We encode both the atomic processes in ISPL using the approach outlined in Section 4.1. We then define synchronisation between the processes for the sequential composition as outlined above. Figure 5 illustrates the composition. A typical evolution function for the "GroupBooking" agent would now be:

```
Evolution:
isBookingSuccessful=true and isValidCard=true and discountedPrice=price -(price * 1/10) if
loggedIn=true and providedCard=true and noOfPeople>=10 and GroupBooking.Action=act1;
isBookingSuccessful=false and isValidCard=false if
 providedCard=true and GroupBooking.Action=act3;
isBookingSuccessful=false if
noOfPeople<=10 and GroupBooking.Action=act2;
sucessMsgSent=true if
isBookingSuccessful=true and GroupBooking.Action=successMsg and
ConfirmReseravation.Action=recBookingSuccessMsg;
cardFailureMsgSent=true if
isBookingSuccessful=false and GroupBooking.Action=invalidCardMsg;
numberFailureMsgSent=true if
isBookingSuccessful=false and GroupBooking.Action=invalidNumMsg;
end Evolution
```

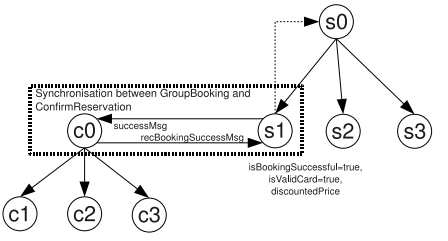

Fig. 5. Sequential composition of GroupBooking and ConfirmReservation

Split, Split+Join. OWL-S provides two types of constructs for concurrent execution: *split* and *split+join*. The components of a split process are a set of processes to be executed concurrently. Split completes when all its component processes have been scheduled for execution whereas split+join completes when all of its component processes have been completed. In both these types of constructs there is a parent process that spawns off the component processes. Split is encoded as illustrated in Figure 6.

The parent process is agent P which spawns three child processes A, B, C at the initial state. Synchronisation between the processes is defined at the initial states of the parent and child processes. Note that in ISPL the parent and child processes are encoded as agents. The number of child agents to be defined can be extracted automatically from the OWL-S definition of composite process.

Similar to split, split+join is encoded as illustrated in Figure 7. In addition to the synchronisation on the initial states of the child processes, the parent process is also synchronised on their final states.

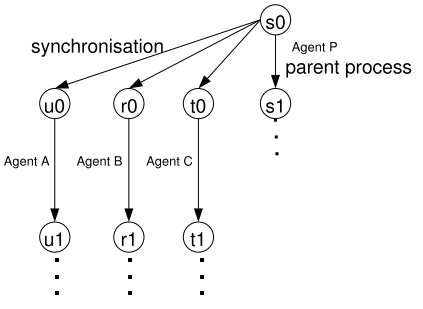

Fig. 6. Encoding split

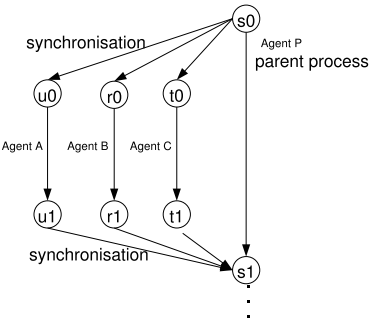

Fig. 7. Encoding split+join

Consider, our extended model for BravoAir. The parent process `ChangeBooking` is composed of two child atomic processes, `ChargeCard` and `AllocateNew Booking` using *split+join*. The processes are synchronised at the states indicated in Figure 8. `ChangeBooking` sends an input message to `ChargeCard` which includes the client's details and the extra payment to be charged. As output `ChargeCard` returns transaction details. Concurrently, `ChangeBooking` sends the client's details, original booking and the requested new booking to `AllocateNewBooking`. The process returns new booking for the client. All messages are encoded as propositions as discussed in section 4.1.

Choice. The *choice* construct allows the invoking process to choose one process non deterministically, from a set of processes. Once the choice is made, the composition essentially resolves to a sequential composition between the invoking process and the chosen process. The parent process as well as the set of processes are encoded as agents in ISPL, as illustrated in Figure 9.

In our extended example for BravoAir, `ManageBooking` is a process, composed as a choice between `ChooseSeats` and the `ChangeBooking` and `CancelBooking`. Due to space restrictions, we do not discuss this in the paper.

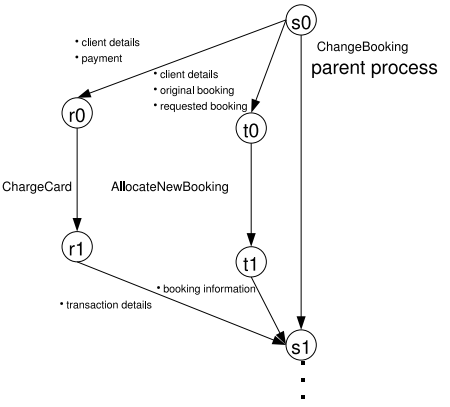

Fig. 8. The ChangeBooking process

If-then-else. The *if-then-else* construct allows to conditionally choose a process from a pair of processes. We define three agents, corresponding to the invoking process and the pair of processes. Since the condition to be checked is a boolean formulae, similar to checking preconditions, we define two states, one where the condition holds and the other where it does not. At these two states we define the synchronisation between the invoking process and the processes in the pair respectively as illustrated in Figure 10.

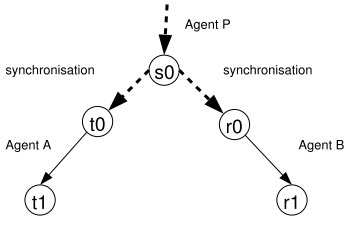

Fig. 9. Encoding choice

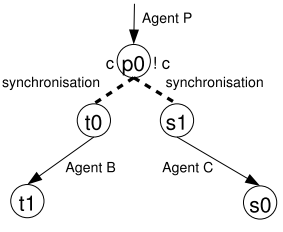

Fig. 10. Encoding if-then-else

Iterate, Repeat-while, Repeat-Until. The *iterate* construct allows the unconditional repeated invocation of a process. Two agents are defined corresponding to the invoking and the iterating process. *Repeat-while* and *Repeat-Until* allow repeated and conditional invocation of a process Synchronisation between the processes is illustrated in Figure 11 and 12.

5 Analysis and Verification

In this section we show the results of verification of some interesting properties of the BravoAir process. Properties are specified as CTL and epistemic formulae[2].

[2] For further details on how such properties could be specified and verified using MCMAS, the interested reader is referred to [1].

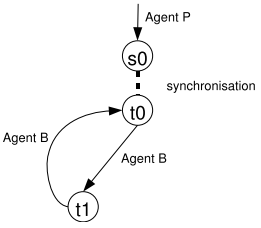

Fig. 11. Encoding iterate

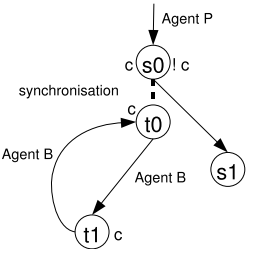

Fig. 12. Encoding repeat-while-repeat-until

– if there is a request for flight booking confirmation, the ConfirmReservation agent (CR) knows that the customer (C) is an authorised customer.

$$EF((confirmBookingRequest) \rightarrow K_{CR}(authorisedCustomer))$$

– If the number of people is more than 10, group booking will always be successful.

$$AF((noOfPeople > 10) \rightarrow EF(isBookingSuccessful))$$

Intuitively the property does not hold because if the card details provided are not valid, the booking will not be successful.

– Whenever a booking change is requested, it will eventually be confirmed.

$$EF((bookingChangeRequest) \rightarrow EF(sendConfirmation))$$

Intuitively the property does not hold because if there are no alternative bookings available, the change will not be confirmed.

– If a card is not charged when a booking is changed, the `ChangeBooking` agent CB knows that the reference is a business class booking.

$$EF(bookingChanged \wedge \neg cardCharged \rightarrow K_{CB}(businessBooking))$$

– if a confirmation is received, the Customer agent(C) knows that his booking was changed.

$$EF(receivedConfirmation \rightarrow K_C(bookingChanged))$$

We encoded the scenario and the specification above in ISPL and verified it using MCMAS. Our system was running on Linux Ubuntu 8.10 (kernel 2.6.27) on Intel Core 2 Duo T5500 1.66GHz with 2GB memory. We encoded 20 ISPL agents by using 120 BDD variables: 43 BDD variables for local states (the same number of BDD variables are constructed for the transition relation) and 34 for local actions. The total number of global states is approximately 10^5. it took about 41 seconds with 34 MB memory space for MCMAS to verify 15 properties. The verification results were in accordance with what expected.

6 Conclusions

Although extensive research has been done on the automated verification of web service composition using BPEL4WS, work on verification of OWL-S process models is relatively scant. Approaches closely related to our work are [2] and [10]. In [2] the mapping rules are defined for Promela to be used with SPIN, an explicit model checker. In [10], the rules are defined for a C-like specification language to be used with BLAST [8]. The limitation in both cases is that only LTL (SPIN) and LTL-like (BLAST) properties can be verified. Using our approach, it is possible to verify LTL, CTL, epistemic and deontic properties with MCMAS. For example, for the case study in Section 2 the composition can be viewed as a multi-agent system where individual processes are abstracted as agents.

In this paper we have proposed mapping rules from the process model of OWL-S to ISPL. We believe the rules are sound as the semantics of ISPL are based on standard kripke semantics and it has been shown that OWL-S processes can be encoded as transition systems [18]. We have shown the mapping for atomic processes and for certain control constructs used for composing them. Our approach provides the first steps necessary to compile automatically OWL-S process models to ISPL. We have developed a compiler that implements the proposed mapping rules. The compiler generates ISPL agents for atomic processes and processes composed in sequence. We are now in the process of enhancing the compiler for other control constructs such as *choice* and *if-then-else*.

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. Lomuscio, A., Qu, H., Solanki, M.: 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 2008). ACM Press, New York (2008)
2. Ankolekar, A., Paolucci, M., Sycara, K.P.: Towards a formal verification of owl-s process models. In: Gil, Y., Motta, E., Benjamins, V.R., Musen, M.A. (eds.) ISWC 2005. LNCS, vol. 3729, pp. 37–51. Springer, Heidelberg (2005)
3. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NuSMV: A new symbolic model verifier. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 495–499. Springer, Heidelberg (1999)
4. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
5. CMU: An API for OWL-S. CMU OWL-S API, `http://www.daml.ri.cmu.edu/owlsapi/index.html`
6. Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., Orchard, D.: Web service architecture. W3c working group note (February 11, 2004), `http://www.w3.org/TR/ws-arch/`
7. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning about Knowledge. MIT Press, Cambridge (1995)

8. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software verification with BLAST. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 235–239. Springer, Heidelberg (2003)
9. Holzmann, G.J.: SPIN Model Checker, The: Primer and Reference Manual. Addison Wesley Professional, Reading (2003)
10. Huang, H., Tsai, W., Paul, R., Chen, Y.: Automated model checking and testing for composite web services. In: Proc. of ISORC 2005, pp. 300–307. IEEE Computer Society, Los Alamitos (2005)
11. Kalyanpur, A., Jimnez, D.: Automatic mapping of owl ontologies into java. In: Proceedings of Software Engeering and Knowledge Engeering (SEKE 2004) (2004)
12. Lomuscio, A., Qu, H., Raimondi, F.: Mcmas 0.9 alpha (2008), `http://sourceforge.net/projects/ist-contract/`
13. Lomuscio, A., Raimondi, F.: MCMAS: A model checker for multi-agent systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 450–454. Springer, Heidelberg (2006)
14. Lomuscio, A., Qu, H., Sergot, M.J., Solanki, M.: Verifying temporal and epistemic properties of web service compositions. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 456–461. Springer, Heidelberg (2007)
15. Kavantzas, N., Burdett, D., Ritzinger, G., Fletcher, T., Lafon, Y.: Web Services Choreography Description Language Version 1.0: W3C Working Draft (December 17, 2004)
16. OASIS Web service Business Process Execution Language (WSBPEL) TC. Web service Business Process Execution Language Version 2.0 (2007)
17. The OWL-S Coalition. OWL-S 1.1 Release (2004), `http://www.daml.org/services/owl-s/1.0/`
18. Traverso, P., Pistore, M.: Automated composition of semantic web services into executable processes. In: McIlraith, S.A., Plexousakis, D., van Harmelen, F. (eds.) ISWC 2004. LNCS, vol. 3298, pp. 380–394. Springer, Heidelberg (2004)
19. Wooldridge, M.: An introduction to multi-agent systems. John Wiley, England (2002)

Transformation Algorithm from OWL-S to ISPL

Algorithm 1. Psuedocode: OWL-S - ISPL mapping

read `ProcessName.owl` {parse the atomic process}
print Agent `ProcessName` {define the agent}
print Vars: {begin the extraction and definition of local variables}
for all hasInput in ProcessName.owl **do**
    print InputID: datatype;
end for
for all hasOutput in ProcessName.owl **do**
    print OutputID: datatype;
    print end Vars {end of variable definition}
end for
$N_{hasResult} \Leftarrow$ count(<process:hasResult>){count the number of <process:hasResult> elements}
$N_{actions} \Leftarrow 2 \times N_{hasResult}$+2 {determine the number of actions for the process}
print RedStates:{begin the definition the red states}
f^{red} {define the Boolean formula for the red states}
print end RedStates {end definition of red states}
print Actions ={ϵ, a_f {begin enumeration of actions}
for all i such that $1 \leq i \leq \mid A_{int} \mid$ **do**
    print a_i, {enumeration of actions taken at states l_0}
end for
for all i such that $1 \leq i \leq \mid A_{send} \mid$ **do**
    print s_i, {enumeration of actions taken at states $\{l_i \mid i = 1 \ldots \mid A_{send} \mid\}$}
end for
print s_f} {end enumeration of actions}
print Protocol: {begin enumeration of the protocols}
printf^{pre} : {define the Boolean formula for the precondition}
for all i such that $1 \leq i \leq \mid A_{int} \mid$ **do**
    print $\{a_i | i = 1 \ldots \mid A_{int} \mid\}$ {enumerate the actions}
end for
print;{end of line}
print$!f^{pre} : a_f$; {define the protocol for precondition failure}
for all i such that $1 \leq i \leq \mid A_{send} \mid$ **do**
    print $f_i^{res} : \{s_i\}$
    print;{end of line}
end for
print end protocol {end enumeration of the protocol}
print Evolution: {begin enumeration of the evolutions}
print l_0 if (
for all i such that $1 \leq i \leq \mid A_{send} \mid$ **do**
    print (l_i and ProcessName.Action=s_i)
end for
print;{end of line}
for all i such that $1 \leq i \leq \mid A_{int} \mid$ **do**
    print l_i if (l_0 and ProcessName.Action=a_i);
end for
print l_f if (Lstate =l_0 and ProcessName.Action=a_f);
end Evolution; {end enumeration of evolutions}
end Agent {end agent definition}