Ad-Hoc networking with OWL-S and CSP

This item was submitted to Loughborough University’s Institutional Repository by the/an author.


Additional Information:

- This is a conference paper [© IEEE]. It is also available from: http://ieeexplore.ieee.org/xpl/RecentCon.jsp?punumber=4155388. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

Metadata Record: https://dspace.lboro.ac.uk/2134/4158

Version: Published

Publisher: © IEEE

Please cite the published version.
This item was submitted to Loughborough’s Institutional Repository (https://dspace.lboro.ac.uk/) by the author and is made available under the following Creative Commons Licence conditions.

For the full text of this licence, please go to: http://creativecommons.org/licenses/by-nc-nd/2.5/
Ad-Hoc Networking with OWL-S and CSP

J. I. Rendo Fernandez, I. W. Phillips

Abstract—In order to achieve a ubiquitous ad-hoc environment suitable for any kind and number of compute devices, information concerning device usability must be stored and manipulated. Take, for example the home where a large number of devices - heating, cooking, lighting, entertainment, security all cooperate to provide a suitable environment for a home resident. This paper proposes a representation of home devices as OWL-S (Web Service Ontology) services, capable of being implemented by means of the formal algebra CSP (Communication Sequential Process). Because of the ontological nature of OWL-S and the possibility of translating CSP equations to lightweight implementations, this proposal allows a rich semantic description of services capable of being hosted by a wide range of devices, including such ones with low computational resources. The paper describes the procedure of developing a service in OWL-S, its translation to CSP and its implementation in occam, an efficient CSP-based language.

Index Terms—ubiquitous, ad-hoc, networking, OWL-S CSP, occam, JCSP.

I. INTRODUCTION

OWL-S is an upper ontology for modelling web-service composition which offers a process-based perspective. In connection with the ontological essence of OWL-S, research in ubiquitous computing points to service oriented architectures in which device capabilities are exposed through ontologies [1]. In light of this, several projects have achieved pervasiveness by completing classical service discovery protocols such as Universal Plug and Play (UPnP) [2], Web Services [3] and Jini [4] by means of OWL-S [5]. However, these technologies demand devices to participate in the service discovery protocol, which might be complicated for simple devices. For example, UPnP and Web Services require devices capable of interpreting SOAP messages.

A ubiquitous home environment implies the participation of several devices, involving sensors, actuators and intelligent systems. For example, imagine a user watching his favourite TV program in his lounge, completely unaware of what is happening in the rest of his house. Suddenly, the security system detects that someone is trespassing the garden. An intelligent system should alert the home resident about this event by redirecting the output of the security camera situated in the garden to the lounge TV. The intelligent system may offer the home resident to automatically close all home windows and phone the local police.

The nature of a ubiquitous computing scenario, particularly in a home environment, suggests the study of such kind of system under the discipline of distributed and embedded systems. On one hand, the necessity of home devices for cooperate, to offer helpful services for the user, requests the adoption of distributed computing techniques.

On the other hand, most of these devices participating in such kind of cooperation are controlled by embedded systems. During the 70s, both topics, distributed and embedded systems, started to be analysed with formal methods. Specially interesting are those methods that propose the study of systems as processes that might communicate among them. This is the case of CSP (Communicating Sequential Process), a notation for modelling concurrent systems devised by C.R.Hoare [6]. CSP specifications are easily implemented as it is supported by several popular programming languages, such as Java [7], [8] and C++ [9] or specific CSP based languages such as occam.

Exploiting the common semantic of a process-based service implementation and its ontological representation brings coherence between these two service perspectives. The service development stage requires designing device services as a set of OWL-S processes, suitable for being converted to CSP [10]. At this point, a formal analysis of the produced service can be done in terms of deadlock, livelock and determinism. After this analysis, the developer may choose the desired implementation, based on the target platform. For example, a switch power service for controlling the power on/off device functionality can be represented in OWL-S. Once the service has been proved correct, the developer may choose between a hardware close implementation in occam, or to a software one in Java.

The next section introduces the reader to OWL-S and CSP, focusing in the OWL-S process model and its correspondence with CSP. Further sections present the service implementation procedure and a probe plugin to automate all this procedure.

II. BACKGROUND

A. Web Service Ontology (OWL-S)

OWL-S is a proposal based on OWL (Ontology Web Language) [11] which specifies an upper ontology for service composition, providing three different knowledge types about a service.

The information about the service capabilities is provided by class service:ServiceProfile 1. This class specifies a service as a set of inputs, outputs, preconditions and results. With this information, web services are composed by means of class service:serviceModel which gives a process view of services. Once services are composed, class service:ServiceGrounding offers all details about their invocation.

Among other information related to a service, class service:ServiceProfile presents what function the service computes. This information is expressed in terms of the

1Service is the namespace for http://www.daml.org/services/owl-s/1.1/Service.owl
transformation that the service produces. Particularly, the profile specifies the inputs required by the service and the outputs generated. Moreover, the profile describes the required preconditions by the service and the expected effects that result from the service execution. All this functionality is exposed through properties profile:hasInput, profile:hasOutput, profile:hasPrecondition and profile:hasResult.

A service profile is a simplified view of a service, since it only gives information about what a service does. To give a more detailed perspective, the service:ServiceModel class describes services as processes. There exist atomic processes that only transform inputs to outputs, and composite processes that are composed by other processes (atomic or composite) using control constructs such as Sequence, If - Then - Else or Choice.

The execution of an OWL-S service can be compared with a combination of remote procedure calls. The OWL-S grounding specifies all the semantics of the parameters to be provided when executing these calls, and the semantics that is returned in messages when the services succeed or fail. A software service user should be able to interpret the grounding class to understand what input is necessary to invoke the service, and what information will be returned.

B. Communicating Sequential Processes (CSP)

In CSP, the behaviour of a process is described by the sequence of events or actions that it may perform. Table I plots part of the notation used in CSP.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prefixing (-&gt;)</td>
<td>( a \rightarrow P ) is a process that behaves like ( P ) after doing event ( a )</td>
</tr>
<tr>
<td>Sequence (;)</td>
<td>( P ; Q ) represents a process that behaves like ( Q ) after behaving like ( P )</td>
</tr>
<tr>
<td>Choice (</td>
<td></td>
</tr>
<tr>
<td>Parallel (</td>
<td></td>
</tr>
</tbody>
</table>

Important in CSP is the concept of channel. Events of the form \( c\!v \) stand for the transmission of message \( v \) on channel \( c \). Each channel has a type which declare the set of values which can be passed on it. If \( T \) is the type of channel \( c \), then the set of events related with \( c \) is \( \{ c\!v \mid v \in T \} \).

Based on this abstraction, process \( c\!v \rightarrow P \) communicates the message \( v \) on channel \( c \) and then behaves like \( P \). Its symmetrical is process \( c\!x : T \rightarrow P(x) \), which is ready to communicate any value of \( x \in T \) and then behave like \( P(x) \).

**Example 1:** Every DVD player offers several commands to the user, such as play, pause and stop.

- \( DVD\_ACTION(play) = \) doPlay \( \rightarrow \) SKIP
- \( DVD\_ACTION(pause) = \) doPause \( \rightarrow \) SKIP
- \( DVD\_ACTION(stop) = \) doStop \( \rightarrow \) SKIP
- \( DVD\_PLAYER = c!x : T \rightarrow \) SKIP

Typically, this functionality is accessed by a remote control.

- \( DVD\_REMOTE = c!play \rightarrow DVD\_REMOTE\[c\!play \rightarrow DVD\_REMOTE]\) c!pause \rightarrow DVD\_REMOTE\[c\!pause \rightarrow DVD\_REMOTE]\) c!stop \rightarrow DVD\_REMOTE\[c\!stop \rightarrow DVD\_REMOTE]\)

Both processes, the DVD player and the remote communicate on a channel called \( c \) of type \( T = \{ play, pause, stop \} \).

Hence, the communication between both processes is expressed by:

- \( DVD = DVD\_REMOTE\[c\!\{c\!play \rightarrow do\!Play \rightarrow Spec\} c\!pause \rightarrow do\!Pause \rightarrow Spec\} c\!stop \rightarrow do\!Stop \rightarrow Spec\} \)

Because both tests Spec \( \subseteq T \) DVD and Spec \( \subseteq F \) DVD are asserted as valid, it is possible to ensure that process DVD satisfies the specification Spec, that is, both processes behave in the same way.

III. PROPOSAL

Our aim is to find a procedure that, given a service description in the OWL-S service model, a correct implementation for the service is obtained. It seems that this goal can be achieved by exploiting the similarities between OWL-S and CSP. Both of them specify systems in terms of processes and their interfaces (inputs and outputs in OWL-S and channels in CSP). However, not all OWL-S structures have their symmetrical in CSP and vice-versa. For this reason, we propose an upper ontology for developing services,

---

\(^2\)profile is the namespace for http://www.daml.org/services/owl-s/1.1/Profile.owl

\(^3\)In this table, it is assumed that \( a \) and \( b \) are elements of the sets \( A \) and \( B \) respectively, which are the alphabets of processes \( P \) and \( Q \) respectively. A process alphabet is nothing more than the set of events that it is allowed to perform.
mainly based on OWL-S which instances can be automatically translated to CSP and subsequently, to a CSP based language in order to obtain an implementation.

In our model, a Service is responsible for offering the functionality of a device. It has a State, which can be accessed through Actions. Changes in the state may be communicated to other services through special outputs called Events. Additionally, a service has an input and an output ports, called Plugs for data communication.

Example 3: The DVD player functionality can be accessed through a service called DVD:Player:Service with three actions, Play, Pause and Stop. These actions are responsible for setting the value of a state variable called TransportState, which ranges with values PLAYING, PAUSED and STOPPED.

A. OWL-S Service Modelling

The first step to develop our model involves the creation of classes owlsx:LufService, owlsx:LufProfile, owlsx:LufProcess and owlsx:LufGrounding which are subclasses of service:Service, service:ServiceProfile, service:ServiceModel and service:ServiceGrounding respectively.

The principal elements of an instance of owlsx:LufService, such as the state, plugs, actions and events are exposed through class owlsx:LufProfile. The state, events, plugs and actions are attached to the profile with properties owlsx:hasStateVariable, owlsx:hasEvent, owlsx:hasPlug and owlsx:hasAction, which range to instances of classes owlsx:State, owlsx:Event, owlsx:Plug and owlsx:Action respectively.

Events and state variables are related through the property owlsx:events. Actions might declare that as a consequence of its invocation, a state variable will change or publish its value. These action consequences are represented as instances of classes owlsx:SetStateResult and owlsx:GetStateResult respectively. Both classes relate state variables, inputs, outputs and literal values by means of the bindings attached to their properties owlsx:withStateVariable and process:withOutputBinding.

Example 4: In Figure 1, the DVD:Player:Service has an action, GetTransportState for retrieving the value of the state variable TransportState. This relation is indicated by the instance of class owlsx:GetStateResult. Action Play is responsible for setting to PLAYING the value of the state variable TransportState. The same schema used for action Play can be applied for actions Stop and Pause. For the purpose of sending video frames, the service has an instance of class owlsx:OutputPlug. The service will not output any piece of data on this plug unless the state variable TransportState changes its value to PLAYING. This situation is indicated by properties owlsx:startsWithError and owlsx:startsWithValue which relate a plug with an event, and hence with a state variable.

B. CSP Modelling

Once the service is declared in OWL-S following the previous layout, it is ready for being translated to a set of CSP equations. The rules for doing such translation are based on previous works of how to express Universal Modelling Language (UML) specifications in CSP [12]. Since a service is composed of actions, state variables, events and plugs, we propose to develop a process for each of these components, except for events. The service is the composition of all these processes as it is plotted in Figure 2.

Users can interact with the service through channels call and return. The first one serves for invoking the desired action on the service with the required input parameters. The return channel is used to retrieve the output results. Actions can interact with the state sending a retrieving messages from channels set and get respectively. Channels getR and ack are used to request and confirm a get or set operation respectively. State changes are communicated on channel event, on which plugs might be listening in order to send or read data.

Example 5: The translation of the DVD player service in OWL-S results in the following piece of CSPM 6 code.

datatype PossibleTransportState = PLAYING | PAUSED | STOPPED

6CSPM stands for Machine Readable CSP, the usual input for the CSP tools such as FDR2)
datatype CALL_PROTOCOL =
call_play | call_pause | call_stop | call_gettransportstate

datatype RETURN_PROTOCOL =
return_play | return_pause | return_stop | return_gettransportstate.PossibleTransportState

datatype GET_PROTOCOL =
get_transportstate.PossibleTransportState

datatype SET_PROTOCOL =
set_transportstate.PossibleTransportState

datatype EVENT_PROTOCOL =
event_transportstate.PossibleTransportState

channel get : GET_PROTOCOL
channel set : SET_PROTOCOL
channel event : EVENT_PROTOCOL
channel call : CALL_PROTOCOL
channel outputvideoplugin : event_transportstate.PossibleTransportState.

TransportState(value) =
get.get_transportstate(value) -> TransportState(value)
set.set_transportstate(new) ->
  if (new != value) then
    event.event_transportstate(new) -> TransportState(new)
  else
    TransportState(new)

State = TransportState(STOPPED)

OutputVideoPlug(PLAYING) =
outputvideoplugin -> OutputVideoPlug(PLAYING)

OutputVideoPlug(value) =
  event.event_transportstate(new) -> OutputVideoPlug(new)

Plugs = OutputVideoPlug(STOPPED)

Actions =
call.call_play -> set.set_transportstate(PLAYING) ->
  return.return_play -> Actions

call.call_pause -> set.set_transportstate(PAUSED) ->
  return.return_pause -> Actions

call.call_stop -> set.set_transportstate(STOPPED) ->
  return.return_stop -> Actions

call.call_gettransportstate ->
  get.get_transportstate() ->
    return.return_gettransportstate() -> Actions

DVD_Player =
(Actions || Plugs)[i:{{set, get, event}}]State
  \{{set, get, event}}

For each state variable, a set of channels get, getR, set and ack are generated. The protocol used over these channels is derived from the process:parameterType property of each state variable. In this case, the enumerated class PossibleTransportState is mapped to a datatype structure in CSP, whose members are the components of the enumerated class. The protocol of channel call is the desired action to invoke followed by its parameters, those are, the OWL-S inputs of the action. The same reasoning is applied for the return protocol, with the outputs of the action. Again, the CSP types are derived from the process:parameterType property of inputs and outputs. Every time that the value hold by process TransportState changes, a message with the new value will be output on channel event. Other processes interested on this event should be listening on this channel. This is the case of process OutputPlug, which is waiting for reading the value PLAYING in order to communicate video frames on channel outputplug. An analysis with FDR2 (a tool for checking CSP specifications) shows that the service is deterministic and free of deadlock and livelock.

C. Implementation

The next step in the service creation routine implies the realization of the service.

Example 6: We consider an ocean implementation of the DVD player presented before.

PROC DVD.Player(
  CHAN OF CALL.PROTOCOL call,
  CHAN OF RETURN.PROTOCOL return,
  CHAN OF CALL.PROTOCOL set,
  CHAN OF RETURN.PROTOCOL event,
  CHAN OF EVENT.PROTOCOL outputplug)

PROC FUN getR.gettransportstate()
  INT value:
  INT dummy:
  SEQ
    value := 0
    WHILE TRUE
      ALT
        getR.gettransportstate ? dummy
        ALT
          get.transportstate: new
          set.transportstate: new
          seq
            IF
              value = new
              SEQ
                event!
                event.transportstate: new
            event!
            event.transportstate: new
  SEQ
  SKIP
The translation from CSP to occam is straightforward. Each process in CSP has its correspondence in occam. In this implementation, it has decided to use integers to codify the different values of the TransportState state variable. In this case, '0', '1' and '2' stand for the values STOPPED, PAUSED and PLAYING respectively. The reason for doing that is because of the easier treatment of integer than strings in occam. The grounding instance of the service must indicate this mapping.  

The reader should be advised that the former piece of code is not correct in terms of indentation, a strong requirement in occam.

**Fig. 3. Developer tool**

**D. Supporting tool**

To automate the routine of service development, that is, creating the service in OWL-S, checking it in CSP and finally obtaining its implementation, authors have developed a protege plugin as an extension of the OWL-S editor [13]. Basically, the tool integrates the OWL-S extension proposed with FDR2. Developers can check the correctness of the services that they are designing just clicking in the CSP button that appears in the OWL-S Editor tab, as it is shown in Figure 3. At this moment, the tool only supports the translation of an OWL-S representation to CSPM. Further work will incorporate automatic code generation from CSP specifications.

**IV. RELATED WORK**

Web service composition is a trendy area of research because of the increasing impact of web services in today’s world. In light of this, several projects attempt to facilitate web service composition. Business Process Execution Language for Web Services (BPEL) and XLANG, together with OWL-S, are examples of these projects. The main difference of BPEL, XLANG and OWL-S is that OWL-S is an ontology while BPEL and XLANG are not. As a consequence, BPEL and XLANG lack of the needed semantics for achieving automatic service composition [14].

The formal verification of web service orchestration is not new and was tackled applying formal analysis to OWL-S [15] and BPEL models [16]. These projects translate composed services to Petri nets and process algebra respectively, mainly, for verification purposes.

One hand, it seems seamless the translation of OWL-S to Petri nets than to a process algebra such as CSP. On the other hand, process algebras provide better composability features than Petri nets. Because of this reason, and because the main purpose of this paper is the achievement of automatic home device composition, we point to the use of CSP as the mechanism for formal verification and service
implementation.

V. CONCLUSIONS AND FUTURE WORK

The proposed schema is a methodology for describing rich semantic services, which are correctly implemented and capable of being deployed in heterogeneous devices.

Firstly, due to the semantics provided by OWL-S, intelligent agents can extract all the needed information about a service in order to compose more complex ones. Several projects have tried to apply artificial intelligence techniques to achieve this goal [17]. The use of OWL-S in this proposal allows the reuse of this vast knowledge. Moreover, because of the standardization process of OWL-S these services are capable of being accessed outside the home environment.

Secondly, the introduction of CSP gives a level of formality enough to predict the behaviour of a service before their deployment. Not only developers can detect and fix common problems of concurrent programming, such as deadlocks or livelock, but also, due to the refinement technique, ensure that an implementation satisfies its specification.

Thirdly, the number of possibilities for implementing CSP specifications is broad enough for being supported by a wide range of devices. The alternatives range from lightweight and efficient implementations in hardware to heavier ones in Java. In fact, the CSP based Java implementations are more efficient than their rivals, such as JavaSpaces (Jini implements JavaSpaces) [18].

Future work is mainly addressed to fix the limitations of the proposal. Firstly, a rich grounding ontology is needed to support a wide range of implementations. Secondly, it is needed a protocol for ensuring interoperation between services implemented with different technologies. Several methods such as raw sockets, RMI and CORBA can be studied to overcome this limitation [19]. Finally, to facilitate the employment of this technology, the presented tool should be improved by integrating automatic code generation from a CSP specification [20].

REFERENCES