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.

COMMONS DEED

Attribution-NonCommercial-NoDerivs 2.5

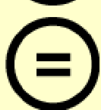
You are free:

- to copy, distribute, display, and perform the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor.

Noncommercial. You may not use this work for commercial purposes.



No Derivative Works. You may not alter, transform, or build upon this work.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

This is a human-readable summary of the Legal Code (the full license).

Disclaimer

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 protege 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

J.I. Rendo and I.W.Phillips are with the Research School of Informatics, Department of Computer Science, Loughborough University, Loughborough, Leicestershire, LE11 3TU, UK (e-mail: J.I.Rendo@lboro.ac.uk, I.W.Phillips@lboro.ac.uk)

[1] service is the namespace for http://www.daml.org/services/owl-s/1.1/Service.owl

Authorized licensed use limited to: LOUGHBOROUGH UNIVERSITY. Downloaded on January 30, 2009 at 09:58 from IEEE Xplore. Restrictions apply.

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*[2] , *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 a 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 plot part of the notation used in CSP[3].

TABLE I
CSP_M OPERATORS

Operator	Behaviour
Prefixing ($->$)	$a \rightarrow P$ is a process that behaves like P after doing event a
Sequence ($;$)	$P; Q$ represents a process that behaves like Q after behaving like P
Choice ($\|$)	$a \rightarrow P \| b \rightarrow Q$ is a process which can either engage in event a and then behave like P, or do event b and then behave like Q
Parallel ($\|\|$)	$P_A \|\|_B Q$ is a process that behaves as the concurrent composition of P and Q, but requires synchronisation between P and Q in the events belonging to the intersection of A and B

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 $\{c!t | t \in T\}$.

[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

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.

$$\begin{aligned} DVD_ACTION(play) &= doPlay \rightarrow SKIP \\ DVD_ACTION(pause) &= doPause \rightarrow SKIP \\ DVD_ACTION(stop) &= doStop \rightarrow SKIP \\ DVD_PLAYER &= c?x : T \rightarrow \\ &\quad DVD_ACTION(x); \\ &\quad DVD_PLAYER \end{aligned}$$

Typically, this functionality is accessed by a remote control.

$$\begin{aligned} DVD_REMOTE &= c!play \rightarrow DVD_REMOTE | \\ &\quad c!pause \rightarrow DVD_REMOTE | \\ &\quad c!stop \rightarrow DVD_REMOTE \end{aligned}$$

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:

$$DVD = DVD_REMOTE \;_c||_c\; DVD_PLAYER$$

The power of CSP as specification language comes from the refinement technique. In CSP it is said that a process P is refined by process Q when the relation $Q \sqsubseteq P$ is asserted. There are two types of refinement: traces and failures. On one hand, trace refinement ($Q \sqsubseteq_T P$) determines if process P at most, engages in the same sequence of events as process Q does. On the other hand, failure refinement ($Q \sqsubseteq_F P$) determines if process P, at least, engages in the same sequence events as process Q does.

Example 2: Process *Spec* represents the desired behaviour of the communication of the remote control and the DVD player, ensuring that actions invoked in the first process correspond to actions in the second process.

$$\begin{aligned} Spec &= c.play \rightarrow doPlay \rightarrow Spec \\ &\quad c.pause \rightarrow doPause \rightarrow Spec \\ &\quad c.stop \rightarrow doStop \rightarrow Spec \end{aligned}$$

Because both tests $Spec \sqsubseteq_T DVD$ and $Spec \sqsubseteq_F DVD$ 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,

Authorized licensed use limited to: LOUGHBOROUGH UNIVERSITY. Downloaded on January 30, 2009 at 09:58 from IEEE Xplore. Restrictions apply.

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 input 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* [4], *owlsx:LufProfile*, *owlsx:LufProccess* 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:GetStateResults* respectively. Both classes relate state variables, inputs, outputs and literal values by means of the bindings attached to their properties *owlsx:withStateVariableBinding* and *process:withOutputBinding*[5].

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:startsWithEvent* and *owlsx:startsWithValue* which relate a plug with an event, and hence with a state variable.

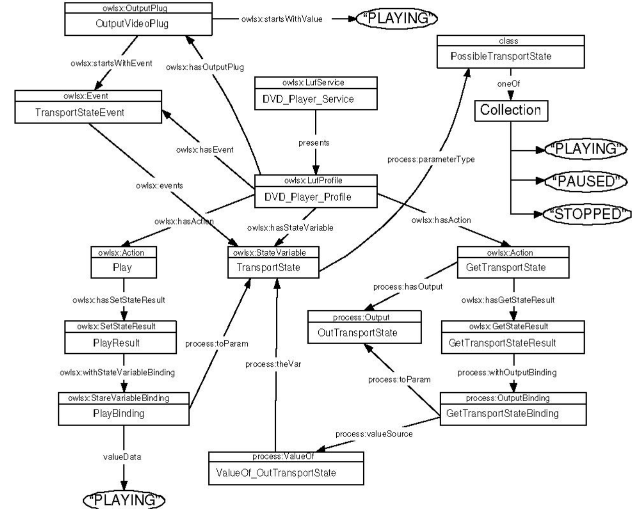


Fig. 1. *DVD_Player* Class Diagram

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.

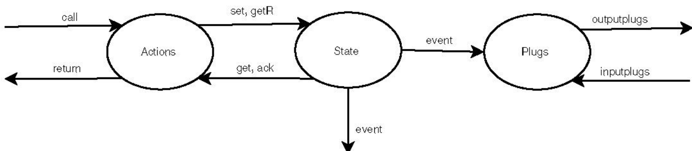


Fig. 2. CSP Model for a LufService

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 CSP_M [6] code.

```
datatype PossibleTransportState = PLAYING | PAUSED | STOPPED
```

[4] owlsx is the namespace for http://www.lboro.ac.uk/owlsextension.owl

[5] process is the namespace for http://www.daml.org/services/owl-s/1.1/Process.owl

[6] CSM_M stands for Machine Readable CSP, the usual input for the CSP tools such as FDR2)

Authorized licensed use limited to: LOUGHBOROUGH UNIVERSITY. Downloaded on January 30, 2009 at 09:58 from IEEE Xplore. Restrictions apply.

```
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 =
  eventtransportstate.PossibleTransportState

channel get : GET_PROTOCOL
channel set : SET_PROTOCOL
channel event : EVENT_PROTOCOL
channel call : CALL_PROTOCOL
channel return : RETURN_PROTOCOL
channel outputvideoplug

TransportState(value) =
   get.get_transportstate!value  -> TransportState(value)
[] set.set_transportstate?new ->
   if (new != value) then
     event.eventtransportstate!new -> TransportState(new)
   else TransportState(new)

State = TransportState(STOPPED)

OutputVideoPlug(PLAYING) =
   outputvideoplug -> OutputVideoPlug(PLAYING)
[] event.eventtransportstate?new -> OutputVideoPlug(new)

OutputVideoPlug(value) =
   event.eventtransportstate?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?v0 ->
   return.return_gettransportstate!v0 -> Actions

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

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 *ProssibleTransportState* 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 occam implementation of the DVD player presented before.

```
PROTOCOL GETR.TRANSPORTSTATE.PROTOCOL IS INT:
PROTOCOL GET.TRANSPORTSTATE.PROTOCOL IS INT:
PROTOCOL SET.TRANSPORTSTATE.PROTOCOL IS INT:

PROTOCOL EVENT.TRANSPORTSTATE.PROTOCOL
  CASE
    eventtransportstate; INT
:

PROTOCOL CALL.PROTOCOL
  CASE
    call.play
    call.pause
    call.stop
    call.gettransportstate
:

PROTOCOL RETURN.PROTOCOL
  CASE
    return.play
    return.pause
    return.stop
    return.gettransportstate; INT
:

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

  CHAN OF GETR.TRANSPORTSTATE.PROTOCOL
  getR.transportstate:
  CHAN OF GET.TRANSPORTSTATE.PROTOCOL
  get.transportstate:
  CHAN OF SET.TRANSPORTSTATE.PROTOCOL
  set.transportstate:
  CHAN OF EVENT.TRANSPORTSTATE.PROTOCOL
  event.outputplug:

  PROC TransportState()
    INT value:
    INT new:
    INT dummy:
    SEQ
      value:= 0
      WHILE TRUE
        ALT
          getR.transportstate ? dummy
            get.transportstate ! new
          set.transportstate ? value
            SEQ
              IF
                value = new
                  SEQ
                    event !
                      eventtransportstate; new
                    event.outputplug !
                      eventtransportstate; new
                TRUE
                  SKIP
```


Authorized licensed use limited to: LOUGHBOROUGH UNIVERSITY. Downloaded on January 30, 2009 at 09:58 from IEEE Xplore. Restrictions apply.

```
            value := new
  :

  PROC State()
    PAR
      TransportState()
  :

  PROC OutputPlug()
    INT dummy:
    INT start.value:
    INT current.value:
    SEQ
      start.value := 2
      current.value := 0
      WHILE TRUE
        SEQ
          ALT
            event.outputplug ? CASE
              eventtransportstate; current.value
                SKIP
          IF
            current.value = start.value
              outputplug ! 1
            TRUE
              SKIP
  :

  PROC Plugs()
    PAR
      OutputPlug()
  :

  PROC Actions()
    INT dummy:
    INT output:
    WHILE TRUE
      call ? CASE
        call.play
          SEQ
            set.transportstate ! 2
            return ! return.play
        call.pause
          SEQ
            set.transportstate ! 1
            return ! return.pause
        call.stop
          SEQ
            set.transportstate ! 0
            return ! return.stop
        call.gettransportstate
          SEQ
            getR.transportstate ! 0
            get.transportstate ? output
            return !
              return.gettransportstate; output
  :

  PAR
    State()
    Actions()
    Plugs()
:
```

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 [7].

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

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

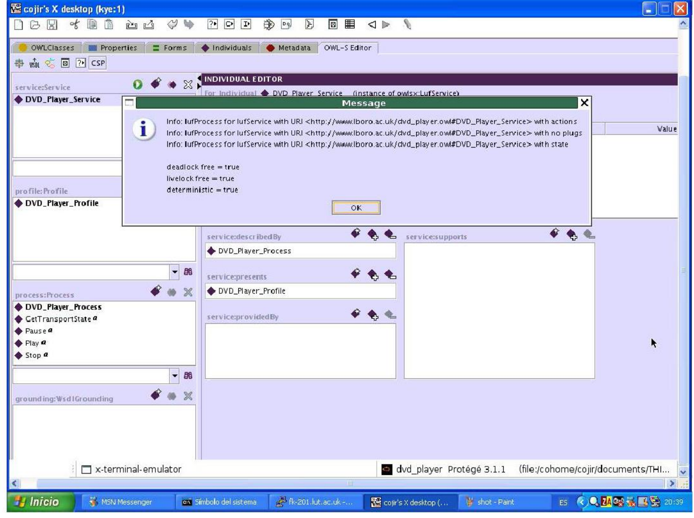


Fig. 3. Developer tool

the translation of an OWL-S representation to CSP_M. 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 business 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

Authorized licensed use limited to: LOUGHBOROUGH UNIVERSITY. Downloaded on January 30, 2009 at 09:58 from IEEE Xplore. Restrictions apply.

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 Java Spaces (Jini implements Java Spaces) [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

[1] S. Helal, "Programming Pervasive Spaces," *IEEE Pervasive Computing*, vol. 4, no. 1, pp. 84–87, Jan. 2005.

[2] R. Masuoka, B. Parsia, Y. Labrou, and E. Sirin, "Ontology-Enabled Pervasive Computing Applications," *IEEE Intelligent Systems*, vol. 18, no. 5, pp. 68–72, Feb. 2003.

[3] Y. Lee, S. A. Chun, and J. Geller, "Web-Based Semantic Pervasive Computing Services," *IEEE Computational Intelligent Bulletin*, vol. 4, no. 2, pp. 4–15, Dec. 2003.

[4] D Chakraborty, F Perich, S Avancha, and A Joshi, "DReggie: Semantic Service Discovery for M-Commerce Applications," in *Workshop on Reliable and Secure Applications in Mobile Environment, In Conjunction with 20th Sym posium on Reliable Distributed Systems (SRDS)*, October 2001.

[5] "OWL-based Web Service Ontology, OWL-S," http://www.daml.org/services/owl-s/.

[6] Hoare, C.A.R, *Communicating Sequential Processes*, Prentice Hall International.

[7] P. H. Welch, "Java Threads in the Light of occam/CSP," in *Architectures, Languages and Patterns for Parallel and Distributed Applications*, P.H.Welch and A.W.P.Bakkers, Eds., Amsterdam, April 1998, WoTUG, vol. 52 of *Concurrent Systems Engineering Series*, pp. 259–284, IOS Press.

[8] G.H Hilderink, A.W.P Bakkers, and J.F Broenink, "A Distributed Real-Time Java System Based on CSP," in *Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, March 2000, p. 400.

[9] N.C Brown and P.H. Welch, "An Introduction to the Kent c++csp Library," in *Communicating Process Architectures 2003*, Jan F. Broenink and Gerald H. Hilderink, Eds., 2003, pp. 1–16.

[10] J.I. Rendo and I.W. Phillips, "Pervasive Computing with OWL-S and a Formal Method," Conference Supplement of the UBICOMP 05 hold in Tokyo, Japan, Sep 2005.

[11] "Web Ontology Language, OWL," http://www.w3.org/2004/OWL/.

[12] J. Davies and C. Crichton, "Concurrency and Refinement in the Unified Modeling Language.," *Formal Aspects of Computing*, vol. 15, no. 2-3, pp. 118–145, Nov. 2003.

[13] D. Elenius, G. Denker, D. Martin, F. Gilham, J. Khouri, S. Sadaati, and R. Senanayake, "The OWL-S Editor - A Development Tool for Semantic Web Services.," in *ESWC*, 2005, pp. 78–92.

[14] S. McIlraith and D. Mandell, "Comparison of DAML-S and BPEL4WS," Tech. Rep., Knowledge Systems Lab, Stanford University, 2002.

[15] S. Narayanan and S. A. McIlraith, "Simulation, Verification and Automated Composition of Web Services," in *WWW '02: Proceedings of the 11th international conference on World Wide Web*, New York, NY, USA, 2002, pp. 77–88, ACM Press.

[16] G. Sala, L. Bordeaux, and M. Schaerf, "Describing and reasoning on web services using process algebra," in *ICWS '04: Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, Washington, DC, USA, 2004, p. 43, IEEE Computer Society.

[17] E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau, "HTN Planning for Web Service Composition Using SHOP2," *Journal of Web Semantics*, vol. 1, no. 4, pp. 377–396, 2004.

[18] N.C. Schaller, S.W. Marshall, and Y. Cho, "A Comparison of High Performance, Parallel Computing Java Packages," in *Communicating Process Architectures 2003*, Jan F. Broenink and Gerald H. Hilderink, Eds., 2003, pp. 1–16.

[19] A. Ripke, A.R Allen, and Y. Feng, "Distributed Computing using Channel Communications in Java," in *Communicating Process Architectures 2000*, Andr W. P. Bakkers Peter H. Welch, Ed., 2003, pp. 1–16.

[20] V. Raju, L. Rong, and G. S. Stiles, "Automatic Conversion of CSP to CTJ, JCSP, and CCSP," in *CPArchitectures 2003*, Jan F. Broenink and Gerald H. Hilderink, Eds., 2003, pp. 63–81.

Authorized licensed use limited to: LOUGHBOROUGH UNIVERSITY. Downloaded on January 30, 2009 at 09:58 from IEEE Xplore. Restrictions apply.