Composition and Semantic Enhancement of Web-Services: The CASheW-s Project

Simon Foster, Andrew Hughes, and Barry Norton

Department of Computer Science, University of Sheffield,
Regent Court, 211 Portobello Street, Sheffield S1 4DP, UK

1 Introduction

CASheW-s is an ongoing research project looking at the composition of semantic web-services via workflow-oriented orchestrations. We have chosen to look first at *OWL-S* [BHLM04] as it has sensibly taken out the core features needed in a workflow language for orchestration and given two existing forms of workflow semantics, as can be seen in [AHS02] and [NM02]. Our aim is to give another form of semantics, in a style which can be efficiently implemented, analysed and extended, courtesy of the principal of compositionality. We do this via an intermediate syntax in an extension to the process calculus *CaSE* [NLM03], which we call *CaSHew-NUtS*.

Formally speaking, we take compositionality to mean that — according to our semantic function $[\![]\!]$ mapping *OWL-S*, via our process calculus, to a labelled transition system semantics, and some notion of semantic equality, $=$ — $[\![t_1; t_2]\!] = [\![t1]\!] \mid [\![t2]\!]$ and furthermore that $[\![t_1]\!] = s \implies [\![t_1; t_2]\!] = s \mid [\![t2]\!]$. The reason this is such an important property of semantics is that this allows a semantic model to be built by composition alongside the definition of syntax, for instance in an interactive editor, as well as being the enabling property of semantics for modular analysis. We claim that no semantics currently defined have this property.

Furthermore we will define an extension to *OWL-S* called *CaSheW-s*, with part of our motivation being to introduce purely functional language extensions [Nor04]. The orchestration engine is being developed in the purely functional language *Haskell* [Jon03], and should be able to execute any workflow whose semantics can be described in the process calculus. We thus envisage in the future the creation of orchestration semantics for ontologies such as *WSMO* [FB02].

Along with the engine, an editor is being produced, which diagrammatically represents an *OWL-S* workflow using an extended fragment of the *UML* Activity Diagram notation. We believe it is important to use a graphical notation familiar to engineers, rather than invent a new one. This takes the form of a plug-in for *Eclipse*, a popular open-source development environment.

2 Representation

While the syntax represented in the *OWL-S* ontology is appropriate for the storage and communication of a workflow, it does not represent the way such a design is built in an editor. In particular, to compose the performance of a new process in an existing context, *i.e.* building a composite process, all previous performances have to have been defined along with their complete workflow, with no possibility to extend this, if the current syntax is used as a model. To increase the utility of compositionality within our semantic translation, we choose to abstract out the connections so that these can be interleaved with the performances they connect within those process types that allow such communication.

$$
\begin{array}{rcl}
\mathit{Process} & ::= & \textbf{AtomicProcess}\ m\ \mathit{AProcess}\ \mid \\
 & & \textbf{CompositeProcess}\ m\ \mathit{CProcess} \\
 & & \quad \mathit{ConsumeList}\ \mathit{ProduceList} \\
\mathit{CProcess} & ::= & \textbf{Sequence}\ \mathit{PerformanceList}\ \mid \\
 & & \textbf{Split}\ \mathit{PerformanceList}\ \mid \\
 & & \textbf{SplitJoin}\ \mathit{PerformanceList}\ \mid \\
 & & \textbf{Any-Order}\ \mathit{PerformanceList}\ \mid \\
 & & \textbf{ChooseOne}\ \mathit{PerformanceList}\ \mid \\
 & & \textbf{IfThenElse}\ \mathit{Performance}\ \mathit{Performance}\ \mid \\
 & & \textbf{RepeatWhile}\ \mathit{Performance}\ \mid \\
 & & \textbf{RepeatUntil}\ \mathit{Performance} \\
\mathit{Performance} & ::= & \textbf{Perform}\ n\ \mathit{Process}\ \mathit{DataAggregation} \\
\mathit{Connection} & ::= & \textbf{Connect}\ n_1, c, n_2, a, j \\
\mathit{PerformanceList} & ::= & \mathit{Performance}\ \mid \\
 & & \mathit{Performance}; (\mathit{PerformanceList})\ \mid \\
 & & \mathit{Connection}; (\mathit{PerformanceList}) \\
\mathit{DataAggregation} & ::= & \mathit{ValueDataList} \\
 & & \mathit{ValueCollectorList} \\
\mathit{ValueDataList} & ::= & ()\ \mid\ a; (\mathit{ValueDataList}) \\
\mathit{ValueCollectorList} & ::= & ()\ \mid\ a, k; (\mathit{ValueCollectorList}) \\
\mathit{ConsumeList} & ::= & ()\ \mid\ a, n, b, j; \mathit{ConsumeList} \\
\mathit{ProduceList} & ::= & ()\ \mid\ c, n, d; \mathit{ProduceList}
\end{array}
$$

Table 1. The *OWL-S* Process Type

We have therefore created, as shown in Table 1, a representation of *OWL-S* in which composite processes[1] depend on performances[2] whose declared dataflow has only two components: *value data* declarations, as before, and what we have called *value collectors*. A value data list just names which inputs[3] are provided by constants. A value collector allows the representation of a *value function*, but only declares how many components are needed, k, not where they come from. In the absence of an explicit value collector declaration, an input is assumed to have a singleton component.

[1] Note that the names of processes are ranged over by m.
[2] Note that the names of performances are ranged over by n.
[3] Note that the names of inputs are ranged over by a and b, and outputs c and d.

Connections are represented as first class members of the *performance lists* in the body of composite process declarations. Note that those composite processes without a performance list do not allow communication between their component parts. Each connection names the performance, n_1, providing the output, c, as well as which component, j, of which input, a, of which performance, n_2, expects the values. In this way new connections can be composed with existing performances and the compositionality of our semantic representation extends to this. Note further that a compositional translation from the existing OWL-S representation to this model is possible since a performance can be immediately followed by the complete set of connections according to its value sources and value functions.

In manipulating the syntax, we have also taken the opportunity to add an explicit *consume* operator, which acts as the complement of the *produce* operator. Implicitly *ConsumeLists* both name the inputs, a, of the composite process, as in the *OWL-S* syntax, and at the same time the components, j, of the inputs, b, of the component performance, n, they provide. The symmetrical nature of the *produce* operator avoids the need to overload the typing of the connections; value source and value function declarations are used in *OWL-S* with a dummy 'parent perform' variable, even though a *process* communication is what is actually being implied.

CaSE is a conservative extension of *CCS* [Mil89] dealing with multi-party synchronizations, which represents the flow of time as abstract clock ticks. These are governed by maximal progress, meaning that internal action is prioritized over, and preempts, the advance of time. *CaSE* has been used to provide semantics for dataflow-oriented software composition [NF04]. We believe that by extension it can be used to provide a compositional model for web-service orchestration. However, our approach needs both ticks respecting maximal progress and a looser version in which maximal can be bypassed. *CaSHEW-NUtS* is a further conservative extension to *CaSE* which expands the transition labels in the operational semantics with the notion of indexed clocks. This index is used to differentiate between clocks for which the transitions respect the properties of maximal progress and determinism, and those which do not. One motivation behind this extension was the provision of semantics for the split operator in *OWL-S*. The split operator completes when each subprocess is scheduled for execution, and, thus, the clock signaling completion of the process has to tick, despite the presence of the remaining internal work still to be achieved in the encapsulated performance.

3 Implementation

Using inductive data-types we have a direct representation for *OWL-S* and *CaSHew-NUtS* processes. In functional style we can then perform a direct implementation of the mathematical translation between the two, that will be described in this work, as well as the operational semantics for *CaSHew-NUtS*.

Inputs, outputs and explicit silent actions, at the implementation level, can be bound to IO actions, which allows the calculus to call atomic web-services and execute inline *Haskell* code. *OWL-S* already allows the insertion of expressions written in an external syntax to be bound to control points, for example in `valueFunction`s and *if-then-else* conditions. As a result, it is our aim to allow the insertion of arbitrary, well-typed *Haskell* expressions at these points. In order to facilitate this, we are implementing a *Haskell* evaluation web service, which takes serialized input, executes the expression and returns serialized output.

In conjunction with the *CASheW-s* project, we are also working on expanding the existing libraries provided by the *HAIFA*[4] project; a library from our previous work for implementing Web-Services in *Haskell*. Included in *HAIFA* is a Generic *XML* Serializer, which allows serialization of *Haskell* data with minimum intervention from the programmer, a *SOAP/1.1* implementation, a web-service publisher and a simple *HTTP* server, on which the orchestration engine will be based. Currently, the work in this area primarily focuses on creating an implementation of *XML Schema* [TBM$^+$01] in *Haskell*, so that *XSD* types used in an ontology can be directly imported for use in a workflow.

In providing this language to the end-user we allow the tree structured data typical of *XML* to be processed easily for the purpose of mediation and so on. *Haskell*, being a purely functional language, explicitly distinguishes between what are pure functions, that is computations which only depend on the inputs supplied, and side-effecting computations, which can be effected by the `IO` monad [Wad90]. This allows a greater degree of safety in executing workflow code, and when working with pure functions only, the behaviour of a piece of code can be guaranteed. All `IO` sections can also be explicitly guarded with `try`/`catch` blocks in order to improve run-time safety and make a programs behaviour fully predictable. Furthermore, potentially harmful behaviour exhibited by code which effects the world can be restricted and potential security holes reduced, a concept very important in the world of the semantic web. Together with the process calculus semantic rules, it should therefore be possible to provide formal semantics for how a given workflow should behave, right down to the grounding level, with the only uncertainty being the behaviour of the servant network and hardware.

We use *Haskell* as an implementation and mediation language for the reasons of safety already mentioned, but also for many of its unique qualities. Being a lazy language, it natively provides an ideal platform on which to construct code incorporating the complex recursive data-structures presented by *RDF* and *XLink*, rather than placing reliance on the termination of a link sequence. This allows *Haskell* to intrinsically deal with infinitely expanding *XML* trees. Higher-order polymorphic functions will also enable a more abstracted and modular way of combining mediators. We also envisage that *Haskell* will lend itself to a straightforward implementation of the partition refinement algorithm, capturing the calculus' behavioural equivalence theory and thereby allowing diagnosis of deadlocks *etc.*

[4] The HAIFA project page can be found at http://savannah.nongnu.org/projects/haifa

4 Editing

Our editor, as an *Eclipse* plug-in, allows for the parsing, serialization and visual manipulation of *OWL-S* constructs. Eventually, the editor will also allow the serialized *OWL-S* to be passed directly to the engine for execution, via the use of an appropriate web-service. In addition, the notion of extension points, common within the *Eclipse* environment [GB04], will be utilised to mirror the extension points seen within the *OWL-S* syntax. Notably, the editor will allow the handling of value functions via this mechanism. There is current work in restricted extensible settings like *Protegé* [Ele04] and other early work in *Eclipse* from CMU, but it is not clear that either is working with the aim of being extensible, as we plan. When extended as a *CASheW-s* editor these functions will be written in *Haskell*. Our implementation of this functionality will center around the editor's communication with a *Haskell* evaluator web-service. The user will be able to enter *Haskell* functions and have them validated by this service. This process also ensures that the function is well-typed, and that this property extends to the surrounding workflow. Once the feasibility of this approach is validated we should like to establish a formal type-theory including a suitable *Haskell* fragment; that [AHS02] is already based in *Haskell* semantics suggests that this would be a promising approach.

As well as being able to integrate *Haskell* functions into an *OWL-S* workflow, we will also build a *CASheW-s* ontology of mediators, which will contain semantically annotated *Haskell* functions, enabling mediation between *XSD* data-types grounding concepts within a workflow. The editor will be capable of looking up functions in this ontology, and inserting them at appropriate points in the workflow.

In this presentation, we will review the *CaSHew-NUtS* process calculus, formally describe the translation of *OWL-S* into this language, describe its implementation and sketch the architecture by which the resulting engine interacts with our editor.

Acknowledgments

This work is being conducted within the Darwin project programme for Masters students and Finalists in the Department of Computer Science at the University of Sheffield; we should like to acknowledge the efforts of the other *CASheW-s* members Atheesh Sanka, Ravish Baghdev and Xian Liu. The work is also partly supported by the *LIRICS* project, sponsored by the eCONTENT program of the European Union (grant EDC-22236), and previously by *Dot.Kom* project, sponsored by the IST part of EU Framework V (grant IST-2001-34038).

References

AHS02. Anupriya Ankolekar, Frank Huch, and Katia Sycara. Concurrent execution semantics of DAML-S with subtypes. In *Proc. 1st Intl. Semantic Web Conference (ISWC2002)*, volume 2342 of *LNCS*, pages 308–332. Springer Verlag, May 2002.

BHLM04. Mark Burstein, Jerry Hobbs, Ora Lassila, and Drew McDermott. OWL-S: Semantic markup for web services. http://www.daml.org/services/owl-s/1.1/overview/, 2004.

Ele04. Daniel Elenius. The OWL-S editor – a development tool for semantic web services. In *Proc. of the 2nd European Semantic Web Conference (ESWC05)*, 2004.

FB02. D. Fensel and C. Bussler. Web service modeling framework (WSMF), 2002.

GB04. Erich Gamma and Kent Beck. *Contributing To Eclipse: Principles, Patterns and Plug-Ins*, chapter 8, page 65. The Eclipse Series. Addison-Wesley, 1st edition, 2004. ISBN: 0-321-20575-8.

Jon03. Simon Peyton Jones. *Haskell 98 Language and Libraries.* Cambridge University Press, 2003.

Mil89. Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

NF04. Barry Norton and Matt Fairtlough. Reactive Types for Dataflow-Oriented Software Architecture. In *Proc. 4th Working IEEE/IFIP Conference on Software Architecture (WICSA04)*, IEEE Computer Society Press, page 2172, 2004.

NLM03. Barry Norton, Gerald Luettgen, and Michael Mendler. A Compositional Semantic Theory for Synchronous Component-Based Design. In *Proc. 14th Intl. Conference on Concurrency Theory (CONCUR'03)*, volume 2761 of *LNCS*, pages 461–476, 2003.

NM02. Srini Narayanan and Sheila A. McIlraith. Simulation, verification and automated composition of web services. In *Proc. 11th Intl. World Wide Web Conference (WWW2002)*, May 7-10 2002.

Nor04. Barry Norton. Proposed functional-style extensions for semantic web composition. In *Proc. of the 1st AKT Workshop on Semantic Web Services (AKT-SWS04)*, volume 122 of *CEUR Workshop Proceedings*, Dec 2004.

TBM$^+$01. Henry S. Thompson, David Beech, Murray Maloney, Noah Mendelsohn, Paul V. Biron, and Ashok̃ Malhotra. XML Schema. http://www.w3.org/TR/2001/REC-xmlschema-1-20010502 , May 2001.

Wad90. Philip Wadler. Comprehending monads. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 61–78. ACM Press, 1990.