WSMO Deliverable

D15.1 v0.1

DATAFLOW FOR
ORCHESTRATION IN WSMO


Authors:
Barry Norton
Thomas Haselwanter

Editors:
Barry Norton

This version:
http://www.wsmo.org/TR/d15/d15.1/v0.1/20070105/

Latest version:
http://www.wsmo.org/2004/d15/d15.1/v0.1/

Previous version:
http://www.wsmo.org/TR/d15/d15.1/v0.1/20060616/
Abstract

One of the distinguishing characteristics between choreography — as currently used in WSMO/L — and orchestration is that orchestration involves communication not just with a client, but also between component parts.

In this deliverable we consider a number of challenges for specifying dataflow in a manner consistent with WSMO/L, and present a proposal which involves introducing the explicit concept of ‘performance’, and a new type of mediator to mediate between performances.

Briefly, the problems that this is intended to solve are:

- the need for mediation in any connection, including dataflow, between heterogenous components;
- the need for ‘extraction’ and ‘aggregation’ in the consumption and production of messages according to the WSML grounding approach;
- the need to reconcile the ‘ontologization’ of abstract state machines (ASMs), via state signatures, with concurrency;
- the need to reconcile the current interpretation of population of concepts in state signatures with explicit control flow;
- the need to provide for dataflow in alternative, more high-level, representations of behaviour than ASMs in various extensions.
Contents

1 Introduction 4
2 Meta-model 5
3 Grammar 6
4 Example 7
5 Extension to Workflow Definitions 10
1 Introduction

The current proposal for Choreography in WSMO proposes that abstract state machines are ontologized so that the state is represented by the instances of a set of ontological concepts and relations (hereafter the document will refer only to concepts, leaving implicit that this also applies to relations) contained in a state signature. The state signature, as well as collecting these concepts, constrains the operations that may be made over the instances by transition rules. These operations are divided into tests and updates. Tests are recursive rules, i.e. decompose into other rules, and may inspect the instances and bind variables; update rules may add, delete or change the instances of a concept. The state signature attaches modes to concepts, as follows:

- concepts of IN mode can be tested on, but neither populated, nor have their instances updated or deleted;
- concepts of OUT mode can be populated, but not the tested on;
- concepts of SHARED mode can be tested on and populated, and the instances updated;
- concepts of STATIC mode can be tested on, but the instances can be neither updated, created or deleted;
- concepts of CONTROLLED mode can be tested on and have their instances created, deleted and updated.

Modes are also the way in which (syntactic) grounding is assigned to concepts to be communicated. IN, OUT and SHARED mode concepts can be given groundings, for instance via WSDL to messages, via interfaces and operations. Since the entire message is grounded — not the separate parts; for example an entire request, not the inputs that make it up — we see that there is a mismatch with the way that dataflow is normally thought of in orchestration; i.e. connecting individual outputs to individual inputs, rather than the messages of which they form ‘parts’ (in WSDL 1.1 terminology).

Furthermore, it is also unclear from the discussion of ontologized ASMs how concurrency can be handled. Unless we prevent multiple use of any given service in the same orchestration, which is unreasonable, it is not clear how properly to reason about multiple instances of an OUT mode concept. Similarly disambiguating between instances in the IN mode concept in a web service’s state signature is a problem if we directly allow multiple achieves over the same service. A related problem concerns the proposal to use gg-mediators to define the dataflow between goals in orchestrations [Domingue et al., 2005], dataflow being potentially different for each use.

The approach of this proposal is to generalise on the mechanism currently proposed in the main D15 deliverable, wherein explicit ‘invoke’ rules are added to the ASM vocabulary. The primary difference here is that the ‘prototypical invocations in context’ that these rules represent are treated as first class members of the description — called performs, which give rise to performances — and given identifiers. Being explicitly identified in this way means that we can introduce mediators between performances and respect the principal, i.e. the use of mediators to express dataflow, from [Domingue et al., 2005] while disambiguating which goal is meant.

We later show how this mechanism can be extended to other representations of behaviour, based on the workflow model from which it originally arose within the DIP project.
2 Meta-model

Figure 1 illustrates the meta-model proposed for dataflow in orchestration. The formal definition in MOF is in Listing 4 in the appendix. The central extension is the notion of a perform action, which represents a prototypical instance in context. In other words, a prototype of the execution of its associated WSMO element (a goal, web service, mediator or communication) in the context of control flow and dataflow.

In order to represent dataflow we introduce a new mediator directly between perform actions, in order to mediate between the performances they give rise to at run-time. The two attributes of pp-mediators are useful for this purpose: ‘usesMediator’ allows the relevant ontologies used each side to be specified via an oo-mediator, if these are different; the ‘hasMediationService’ attribute, as in other mediators, allows the use of a goal or web service to specify the needed mediation.

Where the mediation can be achieved internally, for instance by applying the mapping engine, a goal without a capability is used to specify the transformation. The state signature in the goal choreography is used to specify the concepts that are used; a concept with an OUT mode will be passed to the mapping engine, and will correspond with (i.e. be the same concept as) an OUT mode in a service performed by the source, an IN mode in a goal performed by the source or a CONTROLLED mode of the orchestration. Similarly each IN mode of the mediation goal will correspond with either an IN mode of a service performed by the target, an OUT mode of a goal performed by the target, or with a CONTROLLED mode of the orchestration. A SHARED mode is used in the mediation goal where the respective modes in the source and target are already of the same concept.
3 Grammar

In order to include perform actions in the context of control flow the grammar for abstract state machines is extended to define them, as a generalisation on the proposal for D15 where these are extended with a syntax for invocations.

This grammar is shown in Listing 1.

```
//Grammar

orchrules = ttransitionrules id? nfp? orchrule *;

orchrule = {if} t_if condition t_then rule + t_endif |
{forall} t_forall variablelist |
{forall} t_with condition t_do rule + t_endforall |
{choose} t_choose variablelist |
{choose} t_with condition t_do rule + t_endchoose |
{uncond} piped_rule | 
{update} orchrupdate_rule;

condition = {restricted_le} expr;

piped_rule = rule t_pipe rule;

//Note: this is sufficient and simpler than the grammar in D14

orchrupdate_rule = {state_update} modifier lbrace fact rbrace | 
{perform} t_perform id? performalt;

performalt = {perform_receive} t_receive id | 
{perform_send} t_send id | 
{asm_perform_achievegoal} t_achievegoal id | 
{asm_invoke_service} t_invoke service id | 
{asm_perform_mediation} t_applymediation id;

ppmediator = t_ppmediator id? sources? target? use_service? ;

//TERMINALS

t_perform = 'perform';
t_receive = 'receive';
t_send = 'send';
t_achievegoal = 'achieveGoal';
t_invokeService = 'invokeService';
t_applymediation = 'applyMediation';
t_ppmediator = 'ppMediator';

//REPRODUCED FROM D16

t_source = 'source';
t_target = 'target';

//REPRODUCED FROM D14

t_transitionrules = 'transitionRules';
t_if = 'if';
t_then = 'then';
t_endif = 'endif';
t_forall = 'forall';
t_with = 'with';
t_do = 'do';
t_endforall = 'endForall';
t_choose = 'choose';
t_endchoose = 'endChoose';
t_pipe = '|';

Listing 1: Grammar in SableCC

The intention is that perform rules be performative, that is, when the rule is triggered a performance should be instantiated as a result. This performance — representing an instance of the choreography engine for a goal or service, and an interaction with the mapping engine, or a service or goal invocation, for a mediator — becomes an instance, with the identifier given to the perform rule, of the relevant class defined by the meta-model while this is in progress. For example, when a rule ‘perform pgExample achieveGoal goalExample’ is triggered, an instance of “http://www.wsmo.org/wsml/wsml-syntax#performGoal” is created, where the ‘achievesGoal’ attribute is ‘goalExample’. In this way we see that there can only be one performance at any one time for a given rule.
4 Example

To exemplify this proposal, we use a simplification of a use case from the DIP project. In the telecoms industry, a common business process concerns the ordering of a ‘bundle’: a set of inter-related products and services. In the simplified version we present this is a consumer network service and the modem device necessary to use this. For example, a consumer may want a DSL connection, and require a compatibility check on their exchange and the reservation of provision for this connection then, if successful, a new account, together with a DSL modem. A similar process applies to ISDN and to dial-up, except that different checks are made. This relies on the domain ontology shown in Listing 2.

```plaintext
ontology bundle
concept consumerAccount
concept telephoneAccount subConceptOf consumerAccount
concept networkAccount subConceptOf consumerAccount
hasTelephoneAccount ofType (1) telephoneAccount
concept dialupAccount subConceptOf networkAccount
concept dslAccount subConceptOf networkAccount
concept isdnAccount subConceptOf networkAccount
concept networkProvision
hasTelephoneAccount ofType (1) telephoneAccount
concept dialupProvision subConceptOf networkProvision
concept dslProvision subConceptOf networkProvision
concept isdnProvision subConceptOf networkProvision
concept networkRequest
hasTelephoneAccount ofType (1) telephoneAccount
concept dialupRequest subConceptOf networkRequest
concept dslRequest subConceptOf networkRequest
concept isdnRequest subConceptOf networkRequest
concept modem
concept dialupModem subConceptOf modem
concept dslModem subConceptOf modem
concept isdnModem subConceptOf modem
concept modemRequest
hasModem ofType modem
concept modemOrder
hasModem ofType modem
concept reqBundle
hasNetworkRequest ofType (1) networkRequest
hasModemRequest ofType (1) modemRequest
concept resBundle
hasNetworkAccount ofType (1) networkAccount
hasModemOrder ofType (1) modemOrder
concept error
concept provisionError subConceptOf error
concept stockError subConceptOf error
```

Listing 2: Telecoms Bundle Domain

For our purposes of our example, we encode a general goal for network account orders, which can be met by different services, and a single service for all modem orders. These artifacts are defined in Listing 3, followed by the definition of `wsBundle` which is defined as an orchestration over them.

```plaintext
goal goalProvisionNetwork
capability
sharedVariables {?req, ?tel}
precondition preProvisionNetwork definedBy
?req hasTelephoneAccount hasValue ?tel memberOf networkRequest.
postcondition postProvisionNetwork definedBy
?req hasTelephoneAccount hasValue ?tel memberOf networkProvision.
interface
choreography
stateSignature
```
importsOntology bundle
out networkRequest
in networkProvision, provisionError
transitionRules trProvisionNetwork
  if (exists ?req memberOf networkRequest) then
    choose {?suc} with {?suc = true or ?suc = false} do
      if (?suc = true) then
        add (?res memberOf networkProvision)
      endIf
      if (?suc = false) then
        add (?res memberOf provisionError)
      endIf
    endChoose
  endIf

webService wsProvisionDialUp
capability
  sharedVariables {?req, ?tel}
  precondition preProvisionNetwork
  precondition preProvisionDialUp definedBy
    ?req memberOf dialupRequest.
  postcondition postProvisionDialUp definedBy
    ?res memberOf dialupProvision.
interface choreography
  stateSignature ssProvisionNetwork
  importsOntology bundle
  in networkRequest
  out networkProvision, provisionError
  transitionRules trProvisionNetwork

webService wsProvisionDSL
capability
  sharedVariables {?req, ?tel}
  precondition preProvisionNetwork
  precondition preProvisionDSL definedBy
    ?req memberOf dslRequest.
  postcondition postProvisionDSL definedBy
    ?res memberOf dslProvision.
interface choreography
  stateSignature ssProvisionNetwork
  transitionRules trProvisionNetwork

webService wsProvisionISDN
capability
  sharedVariables {?req, ?tel}
  precondition preProvisionNetwork
  precondition preProvisionDialUp definedBy
    ?req memberOf isdnRequest.
  postcondition postProvisionISDN definedBy
    ?res memberOf isdnProvision.
interface choreography
  stateSignature ssProvisionNetwork
  transitionRules trProvisionNetwork

goal goalCreateNetworkAccount
capability ...
interface choreography
  stateSignature ssCreateNetworkAccount
  importsOntology bundle
  out networkProvision
  in networkAccount
  transitionRules trCreateNetworkAccount ...

webService wsCreateDialUpAccount
capability ...
interface choreography
  stateSignature ssCreateDialUpAccount
  importsOntology bundle
  out networkProvision
  in networkAccount
  transitionRules trCreateNetworkAccount ...
webService wsCreateDSLAccount ...
webService wsCreateISDNAccount ...
webService wsOrderModem
   capability
      sharedVariables {modem}
      precondition definedBy
         req[hasModem hasValue modem] memberOf modemRequest.
      postcondition definedBy
         res[hasModem hasValue modem] memberOf modemOrder.
   interface
      choreography
         stateSignature
            importsOntology bundle
            in modemRequest
            out modemOrder, stockError
         transitionRules
            if (exists req memberOf modemRequest) then
               choose {inStock} with (inStock = true or inStock = false) do
                  if (inStock = true) then
                     add (res memberOf modemOrder)
                  endif
                  if (inStock = false) then
                     add (res memberOf stockError)
                  endif
            endChoose
            endIf
   
webService wsBundle
   capability
      sharedVariables {net, modem, tel}
      precondition definedBy
         req[hasNetworkRequest hasValue net, hasModemRequest hasValue modemReq memberOf reqBundle and
            modemReq[hasModem hasValue modem] and
            net[hasTelephoneAccount hasValue tel].
      postcondition definedBy
         res[hasNetworkAccount hasValue account, hasModemOrder hasValue modemOrd memberOf resBundle and
            modemOrd[hasModem hasValue modem] and
            net memberOf dialupRequest implies
               account memberOf dialupAccount and
            net memberOf dialRequest implies
               account memberOf dialAccount and
            net memberOf isdnRequest implies
               account memberOf isdnAccount.
   interface
      orchestration
         stateSignature
            importsOntology bundle
            in reqBundle
            out resBundle, provisionError, stockError
         transitionRules
            if (exists req memberOf reqBundle) then
               perform prReqBundle receive reqBundle
               perform applyMediation ppPGProvisionNetwork
               perform applyMediation ppPWOrderModem
            endif
            if (exists req memberOf modemRequest) then
               perform pgProvisionNetwork achieveGoal goalProvisionNetwork
            endif
            if (exists prov memberOf networkProvision and
                exists req memberOf modemRequest) then
               perform applyMediation ppPGCreateNetworkAccount
               perform pwOrderModem invokeService wsOrderModem
            endif
            if (exists order memberOf modemOrder and
                exists prov memberOf networkProvision) then
               perform pgCreateNetworkAccount achieveGoal goalCreateNetworkAccount
            endif
            if (exists acc memberOf networkAccount and
                exists order memberOf modemOrder) then
               perform applyMediation ppPSResBundle
            endif
            if (exists resBundle memberOf resBundle) then
               perform psResBundle send resBundle
            endif
            if (exists err memberOf provisionError) then
               perform send provisionError
            endif
            if (exists err memberOf stockError) then
               perform send stockError
            endif
ppMediator ppPGProvisionNetwork
  source prReqBundle
  target pgProvisionNetwork
  usesService goalPPPGProvisionNetwork

  goal goalProvisionNetwork
  interface
    choreography
    stateSignature
    out reqBundle
    in networkRequest

ppMediator ppPWOrderModem
  source prReqBundle
  target pwOrderModem
  usesService goalPPPWOrderModem

  goal goalPPPWOrderModem
  interface
    choreography
    stateSignature
    out reqBundle
    in modemRequest

ppMediator ppPGCreateNetworkAccount
  source pgProvisionNetwork
  target pgCreateNetworkAccount
  usesService goalPPPGCreateNetworkAccount

  goal goalPPPGCreateNetworkAccount
  interface
    choreography
    stateSignature
    shared networkProvision

ppMediator ppPSResBundle
  source {pgCreateNetworkAccount, pwOrderModem}
  target ppResBundle
  usesService goalPPPSResBundle

  goal goalPPPSResBundle
  interface
    choreography
    in networkAccount, modemOrder
    out resBundle

Listing 3: Telecoms Component Services

5 Extension to Workflow Definitions

In order to support the 3-layer approach to behavioural descriptions [Norton et al., 2007] developed during the DIP project — where ASMs form executable descriptions at the bottom layer, the Cashew workflow language forms a workflow patterns-oriented description and UML2 Activity Diagrams form the visual description — there are two possible approaches. The first is to make a WSMO meta-model and WSML grammar extension to explicitly support each different formalism. This was the approach taken due to meta-modelling issues in the object model, WSMO4J \(^1\), that prevented user ontologies being used for these alternative behavioural models over WSML artifacts.

The second approach is to extend the meta-model presented in Section 2 with one further type of perform rule, represented in the meta-model by ‘perform-Workflow’ associated with a concept ‘workflow’ of which we allow subconcepts in user ontologies. There are two advantages of this approach. First it allows a uniform approach to dataflow definition in WSMO, since pp-mediators are then extended to connect externally defined artifacts.

\(^1\)http://wsmo4j.sourceforge.net/
Secondly it allows still further behavioural formalisms to be included. For instance, it can be seen that the SUPER project also defines a stack of ontologies that fit the 3-level model, where BPMN is used as the visual definition of behaviour, the BPMO ontology as a patterns-oriented description, and Semantic BPEL as an executable model.

Further to this, it has been proposed that behavioural fragments should be well-defined, exposing parameters that allow dataflow to be defined in each context where they are performed. To this extend a second extension is proposed where dataflow between performances within a workflow and the workflow’s parameters can be defined using a new kind of mediator between these artifacts, i.e. pf-mediators. It is notable that this approach avoids a rather arbitrary definition of a variable with special treatment ‘theParentPerform’, designed to solve this issue in the OWL-S specification [Martin et al., 2004].

References


Acknowledgement

The work is funded by the European Commission under the projects DIP and SUPER. The editor would like to thank all DIP partners involved in choreography and orchestration, in particular those from ILOG and DERI, as well as all the members of the WSMO working group for their advice and input into this document.
MOF Definitions

```plaintext
Class perform
Class receive sub--Class perform
    target type mode
Class send sub--Class perform
    source type mode
Class performAchieveGoal sub--Class perform
    achievesGoal type goal multiplicity = single-valued
Class performInvokeService sub--Class perform
    invokeService type webService multiplicity = single-valued
Class performApplyMediation sub--Class perform
    hasMediator type dataflowMediator
Class dataflowMediator sub--Class mediator
Class ppMediator sub--Class dataflowMediator
    usesMediator type ooMediator
    hasSource type {perform, ppMediator}
    hasTarget type {perform, ppMediator}
//Extension to workflow descriptions
Class workflow
Class performWorkflow sub--Class perform
    hasWorkflow type workflow multiplicity = single-valued
Class pfMediator sub--Class dataflowMediator
    usesMediator type ooMediator
    hasSource type {perform, workflow}
    hasTarget type {perform, workflow}
```

Listing 4: MOF Meta-model Definition