Deliverable

WP3: Service Ontologies and Service Description

D3.9

Ontology for Web Services
Choreography and Orchestration

Barry Norton (OU)
Carlos Pedrinaci (OU)
Jens Lemcke (SAP)
Mathias Kleiner (ILOG)
Laurent Henocque (ILOG)
Gabriela Vulcu (NUIG)

March 20, 2007
SUMMARY

This document is the culmination of the work started in deliverables D3.4 ("An Orchestration and Business Process Ontology"), D3.5 ("An Ontology for Web Service Choreography") and their common appendix titled "DIP Interface Description Ontology" (DIO). There it was proposed that a ‘three-layer’ model for choreography and orchestration be introduced; while UML Activity Diagrams could provide a human readable representation of behaviour and manipulation in tools oriented towards the software industry, and Abstract State Machines (ASMs) allow compatibility with existing WSMO-based tools, the Cashew ontology would provide a means for translation between these, representation in an ontological form that is amenable to reasoning, and comprehensibility by adopting the standard vocabulary of ‘workflow patterns’.

This deliverable documents extensions to the WSMO meta-model and to the WSML grammar to allow the use of both the Activity Diagrams and Cashew formalisms for the expression of behavioural models for choreography and orchestration. It also documents the translation between these models, including two extensions to the current ASM-based model for interfaces, concretising proposals in the WSMO Working Group.

Disclaimer: The DIP Consortium is proprietary. There is no warranty for the accuracy or completeness of the information, text, graphics, links or other items contained within this material. This document represents the common view of the consortium and does not necessarily reflect the view of the individual partners.
DOCUMENT INFORMATION

<table>
<thead>
<tr>
<th>IST Project Number</th>
<th>FP6 – 507483</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full Title</td>
<td>Data, Information, and Process Integration with Semantic Web Services</td>
</tr>
<tr>
<td>Project URL</td>
<td><a href="http://dip.semanticweb.org/">http://dip.semanticweb.org/</a></td>
</tr>
<tr>
<td>Document URL</td>
<td></td>
</tr>
<tr>
<td>EU Project Officer</td>
<td>Kai Tullius</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Deliverable Number</th>
<th>3.9</th>
<th>Title</th>
<th>Ontology for Web Services Choreography and Orchestration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Work Package Number</td>
<td>3</td>
<td>Title</td>
<td>Service Ontologies and Service Description</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Date of Delivery</th>
<th>Contractual</th>
<th>Actual</th>
<th>31-Dec-06</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>version 0.1</td>
<td>final</td>
<td></td>
</tr>
<tr>
<td>Nature</td>
<td>prototype □</td>
<td>report □ dissemination □ ontology □</td>
<td></td>
</tr>
<tr>
<td>Dissemination Level</td>
<td>public □</td>
<td>consortium □</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Authors (Partner)</th>
<th>Barry Norton (OU), Carlos Pedrinaci (OU), Jens Lemcke (SAP), Mathias Kleiner (ILOG), Laurent Henocque (ILOG), Gabriela Vulcu (NUIG)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resp. Author</td>
<td>Barry Norton</td>
</tr>
<tr>
<td>E-mail</td>
<td><a href="mailto:b.j.norton@open.ac.uk">b.j.norton@open.ac.uk</a></td>
</tr>
<tr>
<td>Partner</td>
<td>OU</td>
</tr>
<tr>
<td>Phone</td>
<td>+44 1908 659399</td>
</tr>
</tbody>
</table>

| Abstract (for dissemination) | This document is the culmination of the work started in deliverables D3.4 (“An Orchestration and Business Process Ontology”), D3.5 (“An Ontology for Web Service Choreography”) and their common appendix titled “DIP Interface Description Ontology” (DIO). There it was proposed that a ‘three-layer’ model for choreography and orchestration be introduced; while UML Activity Diagrams could provide a human readable representation of behaviour and manipulation in tools oriented towards the software industry, and Abstract State Machines (ASMs) allow compatibility with existing WSMO-based tools, the Cashew ontology would provide a means for translation between these, representation in an ontological form that is amenable to reasoning, and comprehensibility by adopting the standard vocabulary of ‘workflow patterns’. This deliverable documents extensions to the WSMO meta-model and to the WSML grammar to allow the use of both the Activity Diagrams and Cashew formalisms for the expression of behavioural models for choreography and orchestration. It also documents the translation between these models, including two extensions to the current ASM-based model for interfaces, concretising proposals in the WSMO Working Group. |

| Keywords | Web Service Interface, Choreography, Orchestration Description, Languages |

<table>
<thead>
<tr>
<th>Version Log</th>
<th>Issue Date</th>
<th>Rev No.</th>
<th>Author</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>01-Dec-06</td>
<td>1</td>
<td>Barry Norton</td>
<td>initial document</td>
</tr>
<tr>
<td></td>
<td>01-Feb-06</td>
<td>2</td>
<td>Barry Norton</td>
<td>Final edits for review</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reviewers</th>
<th>James Scicluna</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partner</td>
<td>DERI Innsbruck</td>
</tr>
<tr>
<td>Phone</td>
<td>+43 512 507 6450</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E-mail</th>
<th><a href="mailto:james.scicluna@deri.org">james.scicluna@deri.org</a></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Jos de Brujin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partner</td>
</tr>
<tr>
<td>Phone</td>
</tr>
<tr>
<td>E-mail</td>
</tr>
</tbody>
</table>
# Project Consortium Information

<table>
<thead>
<tr>
<th>Partner</th>
<th>Acronym</th>
<th>Contact</th>
</tr>
</thead>
</table>
| National University of Galway | NUIG | Dr. Sigurd Harand  
Digital Enterprise Research Institute (DERI)  
National University of Ireland, Galway  
Galway  
Ireland  
E-mail: sigurd.harand@deri.org  
Tel: +353 91 495112 |
| Fundacion De La Innovacion.Bankinter | Bankinter | Monica Martinez Montes  
Fundacion de la Innovation. BankInter,  
Paseo Castellana, 29  
28046 Madrid, Spain  
Email: mmtnez@bankinter.es  
Tel: 916234238 |
| Berlecon Research GmbH | Berelcon | Dr. Thorsten Wichmann  
Berlecon Research GmbH,  
Oranienerburger Str. 32,  
10117 Berlin, Germany  
E-mail: tw@berlecon.de  
Tel: +49 30 2852960 |
| British Telecommunications Plc. | BT | Dr. John Davies  
BT Exact (Orion Floor 5 pp12)  
Adastral Park Martlesham  
Ipswich IP5 3RE, United Kingdom  
Email: john.nj.davies@bt.com  
Tel: +44 1473 609583 |
| Swiss Federal Institute of Technology, Lausanne | EPFL | Prof. Karl Aberer  
Distributed Information Systems Laboratory  
Ecole Polytechnique Federale de Lausanne  
Bât. PSE-A  
1015 Lausanne, Switzerland  
E-mail: Karl.Aberer@epfl.ch  
Tel: +41 21 693 4679 |
| Essex County Council | Essex | Mary Rowlatt,  
Essex County Council,  
PO Box 11, County Hall, Duke Street, Chelmsford, Essex, CM1 1LX, United Kingdom.  
E-mail: maryr@essexcc.gov.uk  
Tel: +44 (0)1245 436524 |
| Forschungszentrum Informatik | FZI | Andreas Abecker  
Forschungszentrum Informatik  
Haid-und-Neu Strasse 10-14  
76131 Karlsruhe, Germany  
E-mail: abecker@fzi.de  
Tel: +49 721 96540 |
<table>
<thead>
<tr>
<th>Company</th>
<th>Contact Information</th>
</tr>
</thead>
</table>
| Institut für Informatik,      | Prof. Dieter Fensel  
| Leopold-Franzens Universität Innsbruck | Institute of computer science  
|                               | University of Innsbruck  
|                               | Technikerstr. 25  
|                               | A-6020 Innsbruck, Austria  
|                               | Email: dieter.fensel@deri.org  
|                               | Tel: +43 512 5076485                                                             |
| ILOG SA                       | Christian de Sainte Marie  
|                               | 9 Rue de Verdun, 94253, Gentilly, France  
|                               | Email: csma@ilog.fr  
|                               | Tel: +33 1 49082981                                                             |
| Inubit AG                     | Torsten Schmale,  
|                               | inubit AG,  
|                               | Lützowstraße 105-106  
|                               | D-10785 Berlin, Germany  
|                               | Email: ts@inubit.com  
|                               | Tel: +49 30726112 0                                                             |
| Intelligent Software Components, S.A. | Dr. V. Richard Benjamins, Director R&D  
|                               | Intelligent Software Components, S.A.  
|                               | Pedro de Valdivia 10  
|                               | 28006 Madrid, Spain  
|                               | Email: rbenjamins@isoco.com  
|                               | Tel: +34 913 349 797                                                             |
| NIWA WEB Solutions            | Alexander Wahler  
|                               | NIWA WEB Solutions  
|                               | Niederacher & Wahler OEG,  
|                               | Kirchengasse 13/1a  
|                               | A-1070 Wien  
|                               | Email: wahler@niwa.at  
|                               | Tel.: +43 131 95843 11                                                            |
| The Open University           | Dr. John Domingue  
|                               | Knowledge Media Institute,  
|                               | The Open University, Walton Hall, Milton Keynes, MK7 6AA, UK  
|                               | Email: j.b.domingue@open.ac.uk  
|                               | Tel.: +44 1908 655014                                                            |
| SAP AG                        | Dr. Elmar Dorner  
|                               | SAP Research, CEC Karlsruhe  
|                               | SAP AG  
|                               | Vincenz-Priessnitz-Str. 1  
|                               | 76131 Karlsruhe, Germany  
|                               | Email: elmar.dorner@sap.com  
|                               | Tel: +49 721 6902 31                                                              |
| Sirma AI Ltd.                 | Atanas Kiryakov,  
|                               | Ontotext Lab, - Sirma AI EAD,  
|                               | Office Express IT Centre, 3rd Floor  
|                               | 135 Tzarigradsko Chaussee, Sofia 1784, Bulgaria  
|                               | Email: atanas.kiryakov@sirma.bg  
<p>|                               | Tel.: +359 2 9768 303                                                              |</p>
<table>
<thead>
<tr>
<th>Unicorn Solution Ltd.</th>
<th>Unicorn Solutions Ltd, Malcha Technology Park 1, Jerusalem 96951, Israel</th>
</tr>
</thead>
<tbody>
<tr>
<td>E-mail: <a href="mailto:Jeff.Eisenberg@unicorn.com">Jeff.Eisenberg@unicorn.com</a></td>
<td></td>
</tr>
<tr>
<td>Tel.: +972 2 6491111</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vrije Universiteit Brussel</th>
<th>Pieter De Leenheer, Starlab- VUB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vrije Universiteit Brussel Pleinlaan 2, G-10 1050 Brussel, Belgium</td>
<td></td>
</tr>
<tr>
<td>E-mail: <a href="mailto:Pieter.De.Leenheer@vub.ac.be">Pieter.De.Leenheer@vub.ac.be</a></td>
<td></td>
</tr>
<tr>
<td>Tel.: +32 (0) 2 629 3749</td>
<td></td>
</tr>
</tbody>
</table>
# Table of Contents

1 **Introduction** .................................................. 1

2 **WSMO Extensions for Behavioural Descriptions** .......... 3
   2.1 WSMO and Abstract State Machines .......................... 3
       2.1.1 Overview ................................................. 3
       2.1.2 Sketched ASM Execution Semantics ....................... 6
   2.2 Cashew .......................................................... 6
   2.3 Revised UML2 AD Subset ......................................... 8
       2.3.1 Metamodel .................................................... 8
       2.3.2 Restrictions & Difficulties .............................. 15

3 **WSML Extensions for Behavioural Descriptions** ........... 17
   3.1 Changes to Existing Core WSML Definitions .................. 17
   3.2 Additional Core WSML Definitions ............................ 17
   3.3 Extensions to Existing WSMO Working Group Proposals for Choreography and Orchestration .......................... 18
   3.4 Grammar for Cashew Workflows ................................ 20
   3.5 Grammar for UML Activity Diagrams ........................... 22

4 **Translations** .................................................. 25
   4.1 Activity Diagrams Subset to ASM .................. 25
       4.1.1 Overview .................................................... 25
       4.1.2 Translation Aspects for Basic Constructs .......... 25
       4.1.3 Translation Aspects for Complex Structures ........ 27
   4.2 Cashew and Activity Diagrams ................................. 29

5 **Example** ....................................................... 35

6 **Conclusion** .................................................... 38

**References** ...................................................... 41

A **MOF Formalisation of Extended WSMO Meta-model** .......... 42

B **OCML Formalisation of Cashew Meta-model** .................. 46

C **MOF Formalisation of Cashew Metamodel** ..................... 50

D **MOF Formalisation of Activity Diagrams Subset Meta-model** 51

E **Translation Rules: Activity Diagrams Subset to ASM** ........ 54
   E.1 Overview & Open Issues ....................................... 54
   E.2 Prerequisites .................................................... 54
       E.2.1 Ontology ..................................................... 54
       E.2.2 Definitions .................................................. 55
       E.2.3 Normal Form ................................................. 56
   E.3 Translation ...................................................... 58
E.3.1 Global Structure .............................................. 58
E.3.2 Concepts ....................................................... 59
E.3.3 Transition Rules ............................................. 59

F Translation between Cashew and Activity Diagrams Subset 64
F.1 Transform Sequential ........................................... 65
F.2 Transform Concurrent .......................................... 65
F.3 Transform XOR .................................................. 65
F.4 Transform While ................................................ 65
F.5 Transform Until .................................................. 66
F.6 Transform Interleave ........................................... 66
F.7 Transform DeferredChoice .................................... 66
F.8 Transform DeferredUntil ...................................... 67
F.9 Transform DeferredWhile .................................... 67
1 INTRODUCTION

This document specifies a meta-model and grammar for describing behavioural interfaces, consisting of choreographies and orchestrations, of semantic Web services (SWS) and goals in the DIP project. This document is the follow-up version of the DIP deliverables D3.5 “An Ontology for Web Service Choreography” that explains the approach for choreography related descriptions, D3.4 “An Orchestration and Business Process Ontology” that is concerned with orchestration, and their common appendix titled “DIP Interface Description Ontology”.

Choreography is concerned with the interactions of Web services with their users in order to consume a Web service, and orchestration is concerned with how a Web service uses and aggregates other Web services in order to achieve its functionality; in WSMO both choreography and orchestration are considered as interfaces that describing behavior of Web services via their interactions, for more details see [20]. Since both are behavioural interfaces concerned with communication of ontological instances, we can use a common formalism as the basis for both types of description. An ontologized form of abstract state machines (ASMs) has previously been provided for choreography [18] and proposed for orchestration [6].

In the previous choreography and orchestration deliverables we proposed that two major deficiencies of the ASM-based behavioural models is the lack of a graphical notation and of tool support. We chose to use UML2 Activity Diagrams for the following reasons:

- UML2 diagrams are a well-understood notation in the software engineering community;
- UML2 diagrams have a good deal of tool support, including by the open source community and in the Eclipse framework on which the WSMO Studio, the standard integration platform in DIP, is based;
- the DIP work on composition already depended on the use of UML2 Activity Diagrams due to its use of ILOG Configurator;
- web service compositions are usually expressed as workflows, and UML2 Activity Diagrams have been shown capable [25] of expressing a good proportion of the Workflow Patterns, which attempt to capture the common idioms of this approach [23].

Our approach in the common appendix to the previous choreography and orchestration deliverables was to build a user ontology to express UML Activity Diagrams and attach these to web service interfaces via WSML non-functional properties. Furthermore, we presented the basis of a translation from these diagrams into ASM-based choreographies for compatibility with WSMO/L.
This work, however, exposed a number of shortcomings. Although we presented the basis for a translation from Activity Diagrams to ASMs, the reverse translation is not possible since the flattened model of behaviour does not allow high-level features such as concurrency to be recovered, meaning that the Activity Diagrams needed to be stored with every service. At the same time, however, the Activity Diagrams ontology is oriented towards graphical display, not ontological reasoning.

The solution to this issue was proposed in the future work to the previous deliverables; the behavioural ontology Cashew, previously developed in the IRS-III [2], was to be introduced, forming a three-level model for behaviour. Cashew is based directly on the patterns by which workflow is characterised, hence meeting the ontological requirement for ‘shared conceptualisation’, and furthermore represents a hierarchical decomposition on these behavioural lines, rather than graph-like decomposition for visual representation, and so is more suited to semantic reasoning. The aim is that the Activity Diagrams and Cashew representations should be inter-translatable, and for execution, and inter-change outside DIP, could be translated to ASMs.

Other problems exposed by the previous work regarded the use of WSML and WSMO4J, its standard object model. First, the ontology itself was not expressed in syntactically valid WSML, specifically when it referred to WSMO entities, and although this could be syntactically corrected, this kind of meta-modelling is not supported by the standard object model for WSML, WSMO4J. Secondly, there was no means provided for the use of goals and services with choreographies described as activity diagrams in orchestrations described by abstract state machines and vice versa. The natural solution to this — to make state signatures a common interface between different modelling notations and not simply a property of abstract state machines — was not allowed by the grammar itself. As a result of these problems, and reviewers suggestions that we integrate the work extended with Cashew more closely with WSMO and WSML led us to propose first-class extensions to these overcoming these problems.

Consequently this document must contain a number of models to fully relay the work carried out:

- the existing WSMO meta-model [20], and its ASM-based extension to choreography, are developed in MOF [16] — Section 2.1 diagrams this and its extension, and leaves the definitions to Appendix A;

- the Cashew model is developed and most fully expressed, in OCML and then re-defined, as an extension to the WSMO meta-model, in MOF — Section 2.2 diagrams this and leaves the definitions to Appendices B and C respectively;

- the Activity Diagrams model continues to be developed, and most fully expressed in UML2 Class Diagrams, constrained by Z — shown in Section 2.3 — and is re-expressed in MOF in Appendix D;

- all extensions are expressed as extensions to the WSML grammar [5], in SableCC, in Section 3.

Finally, Section 4 covers the work to date on translation between the involved languages, with details in further appendices, Section 5 exemplifies the use of these models, and Section 6 concludes the document.
2 WSMO EXTENSIONS FOR BEHAVIOURAL DESCRIPTIONS

This section introduces the meta-models for the 3 levels of behavioural description.

2.1 WSMO and Abstract State Machines

2.1.1 Overview

The fragment of the current WSMO meta-model on which we concentrate is shown in the UML Class Diagram in Figure 2.1 (we elide certain standard attributes throughout; for the full formal definitions, defined in the OMG’s meta-object facilities, MOF [12], please see Appendix A).

We see that the two behavioural models, Orchestration and Choreography (though neither are defined in the current WSMO meta-model; they are merely attribute ranges), currently belong to the concept ‘Interface’, and both goals and services may be described behaviourally over multiple interfaces. WSMO Deliverable D14 [21] proposes to extend the meta-model so Choreography directly contains a behavioural description via ontologized abstract state machines (ASMs), as illustrated in Figure 2.2. Actually this deliverable does not contain a full meta-model, only one for state signatures, but we have reverse-engineered the rest from the WSML grammar, discussed in Section 3.3.

In order to accommodate the extensions we proposed in previous deliverables [8, 9], we propose to generalise on this model as shown in Figure 2.3. Note that it is possible, in this way, to express choreographies and orchestrations at any level within our three-level model. Furthermore, our previous proposal [8] for the need to express client choreographies is added to the model as an attribute of the wg-mediator linking goals to services. In this way, we clarify the role of choreographies attached to web services, in keeping with the main WSMO Use Case [7], which we call service choreographies.

These are attached to the wg-mediator since we see client choreographies as formalising another part of the mediation between goals and services, just as the ‘uses mediator’ attribute allows the attachment of an oo-mediator to define the mediation between the goal ontology and web service ontology, and the ‘mediation service’ attribute allows the attachment of a goal or web service to define the data mediation. In particular, this ‘client choreography’ attribute represents the protocol mediation between the goal choreography and web service choreography.

![Figure 2.1: Relevant Fragment of Current WSMO Meta-model](image-url)
Figure 2.2: WSMO D14 Meta-model
Figure 2.3: Extended WSMO Meta-model
2.1.2 Sketched ASM Execution Semantics

In the translation in Section 4.1, we use a subset of the ASM model whose syntax and execution semantics are as specified in [18]. We give a very brief and informal introduction here to clearly state the assumptions for the following of this document. The ASM rules we used are defined by guarded transition rules of the following form.

\[
\text{if } \text{Condition} \text{ then } \text{Update} \text{ endif}
\]

The description of \textit{Conditions} and \textit{Updates} bases on the explanations in Sect. 2.1.1 as well. The \textit{Conditions} and \textit{Updates} of ASM transition rules operate on states which are represented by the instances of an ontology. Each incoming message from the outside reflects on the ontology by the appearance of a respective instance. In the same way, instances appearing in the ontology are sent as messages when they are appropriately marked. The concepts whose instances are subject to sending and receiving respectively, are marked by being part of a special section in the interface description which denotes their \textit{mode}. For further detail, see Sect. 2.1.1 and its references.

For the handling of this extension to the basic ASM concept, ASM defines an aberrant semantics for rules whose conditions refer to the mentioned messaging ontology concepts as follows.

- Each time the \textit{Condition} of a transition rule \( r \) becomes true,\(^1 \) \( r \) becomes active (\( \text{active}(r) = \text{true} \)).\(^2 \)
- Each time the \textit{Condition} of a transition rule \( r \) becomes false, \( r \) becomes inactive (\( \text{active}(r) = \text{false} \)).
- A transition rule \( r \) can only be executed when it is \( \text{active}(r) \).
- Each time a transition rule \( r \) is executed, it becomes inactive (\( \text{active}(r) = \text{false} \)).

There are other possibilities to express transition rules, like \text{forall} and \text{choose}. These constructs are used to express parallelism and non-determinism, respectively (see [1, p. 31]).

2.2 Cashew

The meta-model of the Cashew workflow language was developed in OCML in the IRS, importing the WSMO model. This definition is reproduced in Appendix B; the model can be diagrammed as shown in Figure 2.4. The model is then expressed in MOF in Appendix C.

Workflow patterns are paired to form block-oriented workflow types, each instance of which compose performances, as in OWL-S [15]. Unlike OWL-S, however, as well as performing sub-workflows, Cashew allows us to perform explicit communications, matching the WSMO model of asynchronous communication, and goals. Rather than performing mediators, as in the ASM extension above, Cashew workflows contain

\(^1\)It suffices, if the execution of an assignment makes a \textit{Condition} true, that has been true before.

\(^2\)Clearly stated, it is necessary that all conjunctive blocks of a \textit{Condition} must become false and then again true in order to activate a rule. (This implies that the same must hold for each block of a disjunctive \textit{Condition}.)

---

\[\text{FP6 – 507483} \]
\[\text{Deliverable 3.9} \]

\[6\]
Figure 2.4: Cashew Workflow Meta-model
dataflow mediators *outside* the control flow. These mediators may also contain a second new type, pf-mediators. These mediate between performances and the parent workflow, avoiding the use of the implicitly defined ‘theParentPerform’, which OWL-S uses to connect to the inputs and outputs of the context.

The workflow types can be understood informally as follows:

- **Sequential** — a sequential workflow performs each of its component parts in turn, and is ready to execute when its first component is ready, and finishes when the last is complete;

- **Interleaved** — an interleaved workflow becomes ready when any component is, and performs its components in any order that they are ready, but strictly one at a time (and is based on the interleaved parallel routing workflow pattern) and finishing only when each has been executed once;

- **Concurrent** — a concurrent workflow becomes ready when all components are and performs its components together, finishing when they are all complete;

- **XOR** — an XOR choice workflow performs exactly one of its two components according to the truth value that results from evaluating a logical expression, and finishes when this is complete;

- **Deferred Choice** — a deferred choice performs exactly one of its components according to readiness, this allows:
  - a non-deterministic choice, when more than one component is ready;
  - an external choice by the client, since this can be based on receive performances;
  - an external choice between components, since this can be decided on goals which are input-ready according to dataflow from other performances;

- **While** and **Until** loop over performances according to the truth value of a logical expression, re-evaluated at each iteration, ‘until’’s iterating once before evaluation;

- **Deferred While** and **Deferred Until** loop over performances, breaking when an alternative branch becomes ready, ignored on the first iteration in the case of ‘until’.

### 2.3 Revised UML2 AD Subset

#### 2.3.1 Metamodel

We present the abstract model (we are defining a subset of UML 2 activity diagrams [10]) as a UML class diagram together with Z constraints allowing to specify well-formed choreographies and orchestrations. In order to produce an unquestionable specification, we have chosen not to use the UML constraint language OCL, but instead a fragment of the Z language. This has several advantages:
• the limitations brought by the exclusive use of the dotted notation in OCL are overcome using Z, a language with extremely rich expressiveness

• all workflow well formedness rules can be presented unambiguously

• Z is extensible: it allows the declaration of user defined operators that complement the syntax. We use this feature to introduce the largely accepted dotted notation. As often as required and possible, OCL like dotted statements will be used.

For clarity, we do not display the Z definitions of classes as they follow the class diagrams. The reader can refer to [14] for an extensive presentation of Z.

We begin with a short overview of activity diagrams, then we present the syntax, semantics and constraints of the constructs allowed in our subset. We also provide a WSML-extension for this UML2 AD subset in Section 3.5.

Overview of UML2 AD Diagrams.

Diagrams. The main constituents of UML2 AD diagrams are actions. These actions are combined by directed flow edges, which can respectively express control and object flow. These flow edges can themselves be interconnected via control nodes. When object flow edges enter or leave actions, there can be pins directly attached to the action that represent the specific data types that are to be exchanged via the respective object flow edge. Another way for expressing the data type of an object flow is to enrich it by object nodes. Their purpose is the same as of pins. However, each type defined by pins or object nodes reached when following an object flow towards its target must be more general than (or equivalent to) its preceding types. A detailed description of UML2 AD diagrams can be found in [10].

Token Flow. For the verbal explanation of the semantics of a UML2 AD diagram, the concept of “token flow” is used. The finishing of an action therefore causes tokens to be existent in every output pin and leaving flow edge. The so-called token flow semantics now is used to describe how control nodes influence the execution of the diagram. In general, an action can only be executed if all its incoming flow edges and input pins yield at least one token each.

Activity Groups.

Semantics.

• Groups: no special semantic, it just enables to group together a part of the activity. Web services can be represented as a group.

• InterruptibleRegions: used to model external choices. Whenever a token traverses an interrupting edge, all other tokens of the region are consumed.

Relations and roles.
Figure 2.5: Overview - Activity Groups

\[ \text{interrupts} : \text{ActivityEdge} \rightarrow \text{InterruptibleActivityRegion} \]
\[ \text{isInputOf} : \text{ActivityEdge} \rightarrow \text{ActivityNode} \]
\[ \text{isOutputOf} : \text{ActivityEdge} \rightarrow \text{ActivityNode} \]
\[ \text{incomingEdges} : \text{ActivityNode} \rightarrow \mathbb{P} \text{ActivityEdge} \]
\[ \text{outgoingEdges} : \text{ActivityNode} \rightarrow \mathbb{P} \text{ActivityEdge} \]
\[ \text{immediatelyContainedGroups} : \text{ActivityGroup} \rightarrow \mathbb{P} \text{ActivityGroup} \]
\[ \text{immediatelyContainedNodes} : \text{ActivityGroup} \rightarrow \mathbb{P} \text{ActivityNode} \]
\[ \text{nodeGroup} : \text{ActivityNode} \rightarrow \text{ActivityGroup} \]

\[ \forall n : \text{ActivityNode}; g : \text{ActivityGroup} \quad g = \text{nodeGroup}(n) \iff n \in \text{immediatelyContainedNodes}(g) \]
\[ \forall n : \text{ActivityNode} \quad \text{incomingEdges}(n) = \{ e : \text{ActivityEdge} | e\text{.isInputOf} = n \} \]
\[ \forall n : \text{ActivityNode} \quad \text{outgoingEdges}(n) = \{ e : \text{ActivityEdge} | e\text{.isOutputOf} = n \} \]

Constraints.

- InterruptibleRegions: Interrupting edges have source in the region and target outside the region

\[ \forall x : \text{ActivityEdge}; y : \text{InterruptibleActivityRegion} \mid y = x\text{.interrupts} \quad \]
\[ x\text{.isOutputOf}\text{.nodeGroup} = y \land \]
\[ x\text{.isInputOf}\text{.nodeGroup} \neq y \]

Activity Nodes and Edges.

Semantics. The operational semantics of object and control flows are described in the UML as ”traverse-to-completion” semantics. The aim of these semantics is to allow workflow not to enter undue self blocking states, that could be caused for instance by tokens mistakenly sent to an alternative outgoing path, and thus missing for a synchronization to occur via an other outgoing path. The currently presented subset
of UML2AD diagrams overcomes most difficulties by disallowing random alternative routes outgoing actions. In other words, when a token is produced by an action, it is presented to an output pin that has no more than one edge connected.

- Object Flows: carry data tokens
- Control Flows: carry control tokens.
- Guards: conditions expressing which decision node’s outgoing edge will receive a token.

**Attributes.** We define *Guard* as an uninterpreted set

\[ \text{Guard} \]

and else a particular member of *Guard*:

\[ \text{else} : \text{Guard} \]

We now specify the *guard* attribute as a partial function from *ActivityEdge* to *Guard*:

\[ \text{guard} : \text{ActivityEdge} \rightarrow \text{Guard} \]

**Constraints.**

- **ActivityEdge:**
  - Only edges outgoing from a decision node can have a guard. Decision nodes are visually and formally presented with the other control nodes later in the document in Figure 2.9

\[ \forall e : \text{ActivityEdge}; g : \text{Guard} \mid g = \text{guard}(e) \land e.\text{isOutputOf} \in \text{DecisionNode} \]

- Only one edge outgoing from the same decision node can have an else condition as the guard.

\[ \forall n : \text{DecisionNode} \bullet \\
\#\{e : \text{ActivityEdge} \mid n = \text{isOutputOf}(e) \land \text{else} = \text{guard}(e)\} = 1 \]
• Control Flow:
  Control flows may not have object nodes at either end

\[ \forall e : \text{ControlFlow} \bullet \\
  e.\text{isInputOf} \notin \text{ObjectNode} \land e.\text{isOutputOf} \notin \text{ObjectNode} \]

Action and Object Nodes.

![Diagram of Action Nodes and Object Nodes]

Figure 2.8: Action Nodes and Object Nodes

Semantics.

• ActionNode:
  - Denotes that a local action is realized at this node
  - Pins are used to receive and send data tokens
  - The inputs are synchronized (all incoming edges and input pins have to carry a token for the action to start)

• OOMediation: This is an additional construct from UML2AD specification

• AbstractEvent: this is an additional construct from UML2AD specification. Not executable, i.e any AbstractEvent has to be specialized. source and target only apply allows to specify the corresponding sender of receiver in an orchestration.

• Operation: this is an additional construct from UML2AD specification. Specifies a local action as being an operation

• Adaptation: this is an additional construct from UML2AD specification. Specifies an aggregation or extraction of message(s)
Relations.

- concept : ObjectNode → Concept
- ontology : Concept → Ontology
- node : Pin → ActionNode
- inputPins : ActionNode → P InputPin
- outputPins : ActionNode → P OutputPin

∀ y : ActionNode • inputPins(y) ∩ outputPins(y) = ∅
∀ x : Pin; y : ActionNode • node(x) = y ⇔ x ∈ inputPins(y) ∪ outputPins(y)

Constraints.

- ObjectFlow:
  - Object Flow connects exclusively object nodes, decision nodes, merge nodes, fork nodes and join nodes.

∀ f : ObjectFlow •
  {f.isInputOf} ∩ ObjectNode ≠ ∅
  ∨ {f.isInputOf} ∩ DecisionNode ≠ ∅
  ∨ {f.isInputOf} ∩ MergeNode ≠ ∅
  ∨ {f.isInputOf} ∩ ForkNode ≠ ∅
  ∨ {f.isInputOf} ∩ JoinNode ≠ ∅

- The downstream object node type must be the same of the upstream object node type

∀ f : ObjectFlow; s, t : Pin |
  s = isOutputOf(f) ∧ t = isInputOf(f) •
  s.ontology = t.ontology

∀ f : ObjectFlow; s, t : Pin |
  s = isOutputOf(f) ∧ t = isInputOf(f) •
  s.concept = t.concept

- AcceptEvent:
  No incoming activity edge

  ∀ e : ActivityEdge • e.isInputOf ∉ AcceptEvent

- SendEvent:
  No outgoing activity edge

  ∀ e : ActivityEdge • e.isOutputOf ∉ SendEvent
Control Nodes.

Semantics.

- **AbstractSplit**: this is an additional construct from UML2AD specification. Not executable: any AbstractSplit has to be specialized.

- **AbstractJoin**: this is an additional construct from UML2AD specification. Not executable: any AbstractJoin has to be specialized.

- **MergeNode**: any token offered on any incoming edge is offered to the outgoing edge.

- **DecisionNode**: each token arriving can traverse to only one outgoing edge.

- **ForkNode**: incoming token duplicated to outgoing edges.

- **JoinNode**: when all incoming edges have tokens, one is created on outgoing edge. Only one incoming edge can be an object flow. Outgoing edge can be an object flow only if there is an object flow among the incoming edges (in this case, the incoming data token is sent to the outgoing edge).

- **Flow Final**: consumes one token.

- **Activity Final**: all tokens in the activity are consumed.

Constraints.

- **AbstractSplit**: 1 incoming edge only
  \[
  \forall x : \text{AbstractSplit} \quad \#(x.\text{incomingEdges}) = 1
  \]

- **AbstractJoin**: 1 outgoing edge only
  \[
  \forall x : \text{AbstractJoin} \quad \#(x.\text{outgoingEdges}) = 1
  \]
• JoinNode:
  Only one incoming edge is an object flow

  \[ \forall x : \text{JoinNode} \bullet \#((x.\text{incomingEdges}) \cap \text{ObjectFlow}) \leq 1 \]

• InitialNode:
  no incoming edge

  \[ \forall x : \text{InitialNode} \bullet x.\text{incomingEdges} = \emptyset \]

• FinalNode:
  no outgoing edge

  \[ \forall x : \text{FinalNode} \bullet x.\text{outgoingEdges} = \emptyset \]

• DecisionNode: the edges coming into and out of a decision node must be either all object flows or all control flows

• MergeNode: the edges coming into and out of a decision node must be either all object flows or all control flows

  \[ \forall x : \text{ActivityNode} \mid x \in \text{DecisionNode} \cup \text{MergeNode} \bullet \]
  \[ (x.\text{incomingEdges} \cup x.\text{outgoingEdges}) \subseteq \text{ObjectFlow} \lor \]
  \[ (x.\text{incomingEdges} \cup x.\text{outgoingEdges}) \subseteq \text{ControlFlow} \]

2.3.2 Restrictions & Difficulties

There remain a number of elements not addressed by the current status of the AD version of the choreography and orchestration languages. This is so because the UML2 fragment for activity diagrams is not entirely supported. The choice for these restrictions is mostly a pragmatic one. We can illustrate this by listing a few examples.

Exceptions.

The lack for handling exceptions matches a difficulty with composing web services in the presence of compensation requirements. This case could not be addressed in the project and remains a research target when considering the composition of Semantic Web Services.

Multiple outputs.

The UML notation prepares for actions where an output pin serves several edges. The semantics for this construct generate a token competition across outgoing edges, which raises potential deadlock situation issues. We have chosen to deal with workflows that do not allow such multiple branching points on output pins, because they did not occur as an obvious need in all the use cases we explored.
Pin cardinality.

UML again does offer the possibility to set cardinality constraints on pins. The idea is simple: the pin acts as a container with limited capacity. This possibility is non-covered by the subset of activity diagrams chosen here. Again, this seems totally unnecessary for choreographies, and although the inner implementation of the orchestration of a web service may involve such a semantics, this is beyond the scope of the composition prototype to generate workflows that would have this feature.
3 WSML EXTENSIONS FOR BEHAVIOURAL DESCRIPTIONS

The following sections specify the grammar for interfaces in dialect of Extended BNF (Backus-Naur Form) which can be used directly in the SableCC compiler compiler\(^1\). This grammar extends the grammar of the Web Service Modeling Language WSML [4] and extends the definitions in [21], as shown in the first sections.

3.1 Changes to Existing Core WSML Definitions

As stated in the meta-model, we add an optional client choreography as an attribute of wg-mediators. This means refining the definition from [4] and introducing the grammar keyword `client_choreography` and terminal `appliesChoreography`.

\[
\text{wgmediator} = \text{t_wgmediator id? source target} \\
\text{client_choreography? use_service?};
\]

\[
\text{client_choreography} = \text{t_applieschoreography id};
\]

//TERMINALS

\[
\text{t_wgmediator} = \text{‘wgMediator’}; \\
\text{t_applieschoreography} = \text{‘appliesChoreography’};
\]

3.2 Additional Core WSML Definitions

Following our proposal in the WSMO/L Working Group Deliverable D15.1, we introduce the following extensions to the core grammar. Note that although the different types of performance are defined as subclasses to ‘perform’ in the meta-model, since this makes it easier to define meta-models that refer to these, each perform is actually introduced with two keywords, first the generic ‘perform’ and then the type, following the identifier. This makes it clearer what can be connected via ppMediators.

//DATAFLOW MEDIATORS (new top-level elements)

\[
\text{ppmediator} = \text{t_ppmediator id? source? target? use_service?}; \\
\text{pfmediator} = \text{t_pfmediator id? source? target? use_service?};
\]

//Re-defining from WSML D16

\[
\text{mediator} = \text{oomediator |} \\
\text{ggmediator |} \\
\text{wgmediator |} \\
\text{wwmediator |} \\
\text{ppmediator |} \\
\text{pfmediator};
\]

//PERFORMANCES (for illustration only, not directly used in grammar -

\(^1\)http://www.sablecc.org/
different formalisms use subsets)

\[
\text{performance} = t_{\text{perform}} \text{id? perform-alt};
\]

\[
\text{perform-alt} = \{\text{perform\_achievegoal}\} t_{\text{achievegoal}} \text{id} | \\
\{\text{perform\_invokeservice}\} t_{\text{invokeservice}} \text{id} | \\
\{\text{perform\_applymediation}\} t_{\text{applymediation}} \text{id} | \\
\{\text{perform\_receive}\} t_{\text{receive}} \text{source} | \\
\{\text{perform\_send}\} t_{\text{send}} \text{target} | \\
\{\text{perform\_workflow}\} t_{\text{workflow}} \text{id};
\]

//TERMINALS

\[
t_{\text{perform}} = \text{'perform'};
\]

\[
t_{\text{invokeservice}} = \text{'invokeService'};
\]

\[
t_{\text{achievegoal}} = \text{'achieveGoal'};
\]

\[
t_{\text{applymediation}} = \text{'applyMediation'};
\]

\[
t_{\text{receive}} = \text{'receive'};
\]

\[
t_{\text{send}} = \text{'send'};
\]

\[
t_{\text{workflow}} = \text{'workflow'};
\]

\[
t_{\text{ppmediator}} = \text{'ppMediator'};
\]

\[
t_{\text{pfmediator}} = \text{'pfMediator'};
\]

3.3 Extensions to Existing WSMO Working Group Proposals for Choreography and Orchestration

We preserve the definition of interface from [4], but represent it here for clarity. Thereafter we adjust the grammar from [18] to allow ASMs, defined via the grammar keyword ‘abstractStateMachine’, which is optional to preserve backwards compatibility, to be just one possible definition of behaviour, all sharing the common notion of state signature, which is detached from ASMs.

The grammar keyword orchestration_transitions expands on the adaptation of the choreography ASM language to orchestration from WSMO Working Group Deliverable 15 [6]. The productions activity_diagram and cashew_workflow are defined in the sections that follow.

\[
\text{choreography} = t_{\text{choreography}} \text{id? header* state\_signature? choreography\_formalism};
\]

\[
\text{orchestration} = t_{\text{orchestration}} \text{id? header* state\_signature? orchestration\_formalism};
\]

\[
\text{choreography\_formalism} = \{\text{chor\_asm}\} t_{\text{asm}} \text{transitions} | \\
\{\text{chor\_ad}\} t_{\text{activity\_diagram}} \text{activity\_diagram} | \\
\{\text{chor\_cashew}\} \text{cashew\_workflow};
\]
orchestration_formalism = \{orch_asm\} t_asm? orch_transitions |
\{orch_ad\} t_activitydiagram activity_diagram |
\{orch_cashew\} cashew_workflow;

//EXTENSION OF D15

orch_transitions = t_transitionrules id? nfp? orch_rule*;

orch_rule = \{orch_if\} t_if condition t_then orch_rule+ t_endif |
\{orch_forall\} t_forall variablelist
    t_with condition t_do orch_rule+ t_endforall |
\{orch_choose\} t_choose variableslist
    t_with condition t_do orch_rule+ t_endchoose |
\{orch_piped\} piped_orch_rule |
\{orch_update\} update_rule |
\{orch_perform\} perform_rule;

condition = \{restricted_le\} expr;

piped_orch_rule = orch_rule t_pipe orch_rule;
//Note: this corrects an error in D14/15

perform_rule = t_perform id? asm_perform_alt;

orch_perform_alt = \{orch_perform_achievegoal\} t_achievegoal id |
\{orch_perform_invokeservice\} t_invokeservice id |
\{orch_perform_applymediation\} t_applymediation id;

//TERMINALS

t_choreography = 'choreography';
t_orchestration = 'orchestration';
t_asm = 'abstractStateMachine';
t_activitydiagram = 'activityDiagram';
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 = '|';
3.4 Grammar for Cashew Workflows

//CASHEW HEADER

cashew_workflow = cashew_workflow_def cashew_sub_defs?

cashew_sub_defs = t_defines lbrace cashew_workflow+ rbrace;

//PERFORMANCES

cashew_perform = t_perform id? cashew_perform_alt;

cashew_perform_alt = {cashew_perform_achievegoal} t_achievegoal id |
{cashew_perform_receive} t_receive source |
{cashew_perform_send} t_send target |
{cashew_perform_workflow} t_workflow id;

atom_perform = t_perform id? atom_perform_alt;

atom_perform_alt = {atom_perform_achievegoal} t_achievegoal id |
{atom_perform_receive} t_receive source |
{atom_perform_send} t_send target;

//WORKFLOWS

cashew_workflow_def = t_cashew_workflow id? state_signature? workflow_pattern dataflow_mediators?

workflow_pattern = {sequential} t_sequential min2perflist |
{concurrent} t_concurrent min2perflist |
{interleaved} t_interleaved min2perflist |
{xor} cashew_perform if? t_xor cashew_perform |
{while} t_while log_expr cashew_perform |
{until} t_until log_expr cashew_perform |
{deferredchoice} t_deferredchoice min2choicelist |
{deferredwhile} t_until atom_perform |
{deferreduntil} t_repeat cashew_perform |
{deferreduntil} t_until atom_perform;

if = t_if log_expr;

min2perflist = lbrace cashew_perform moreperfs+ rbrace;

moreperfs = comma cashew_perform;

min2choicelist = lbrace atom_perform enables? morechoices+ rbrace;

morechoices = comma id enables?;

enables = t_enables cashew_perform;

dataflow_mediators = t_dataflowmediators idlist;

//TERMINALS

t_cashew_workflow = ‘cashewWorkflow’;
t_defines = ‘defines’;
t_sequential = ‘sequential’;
t_concurrent = 'concurrent';
t_interleaved = 'interleaved';
t_xor = 'xor';
t_while = 'while';
t_until = 'until';
t_deferredchoice = 'deferredChoice';
t_deferredwhile = 'deferredWhile';
t_deferreduntil = 'deferredUntil';
t_unless = 'unless';
t_repeat = 'repeat';
t_dataflowmediators = 'dataflowMediators';
3.5 Grammar for UML Activity Diagrams

activity_diagram = startnode? group;

// ACTIVITY GROUPS

group = {activitygroup} t_activitygroup id? activitygroupcontents |
{interruptibleregion} t_interruptibleregion id? activitygroupcontents;

activitygroupcontents = lbrace node+ edge* group* rbrace;
// activity groups group immediately contained nodes, edges and
// subgroups
// edges are attached to the group to which their source node belongs

// NODES

node = {generalaction} t_generalaction id? nfp* nodecontents |
{oomediator} t_oomediator id? nfp* [t_definedby id]? nodecontents |
{flowstart} t_flowstart id? nfp* |
{flowfinal} t_flowfinal id? nfp* |
{activityfinal} t_activityfinal id? nfp* |
{aggregation} t_aggregation id? nfp* [t_definedby id]? nodecontents |
{extraction} t_extraction id? nfp* [t_definedby id]? nodecontents |
{operation} t_operation id? nfp* [t_definedby id]? nodecontents |
// definedBy is the identifier for a goal
{fork} t_fork id? nfp* |
{join} t_join id? nfp* |
{decision} t_decision id? nfp* |
{merge} t_merge id? nfp* |
{sendeventaction} t_sendeventaction id? nfp* sslink? adlink? nodecontents |
{accepteventaction} t_accepteventaction id? nfp* sslink? adlink? nodecontents |
{objectnode} t_objectnode id? nfp* sslink? carriesconcept?;

startnode = t_startnode id;
// the start node of a choreography or orchestration
sslink = t_sslink id;
// In a choreography, this is the in/out mode of the state signature
// the event corresponds to.
// In an orchestration:
// - for events this is the in/out mode in the state
// signature of the orchestration, or in a composed service’s
// - for object nodes this is the controlled mode of the
// orchestration’s own state signature

adlink = carriesconcept partnerlink?;
carriesconcept = t_carriesconcept id;
partnerlink = t_partnerlink id;
// If by-passing state signatures we name the concept carried
// (which need not be unique).
// In the case that we are defining an orchestration, we also
// name the event in the partner activity diagram
// (since this necessarily defines a choreography)

nodecontents = pin*;
// a node basically lists its pins

pin = {inputpin} t_inputpin id
nfp* sslink? carriesconcept? |
{outputpin} t_outputpin id
nfp* sslink? carriesconcept?;
// a pin is input or output and has a link to identify the
// mode in the state signature to which it is mapped

// EDGES

edge = {controlflow} t_controlflow id?
   nfp* edgecontents |
   {dataflow} t_dataflow id?
   nfp* edgecontents;
// edge declarations bind edges to the previously declared nodes

edgecontents = source target interrupting? guard?;
interrupting = t_interrupts id;
// the id must be that of a valid activity group

guard = {else} t.guard t_else |
   {expression} t.guard log_expr;
//TERMINALS

t_definedby = ‘definedBy’;
t_adchoreography = ‘adChoreography’;
t_activitygroup = ‘activityGroup’;
t_interruptibleregion = ‘interruptibleRegion’;
t_generalaction = ‘generalAction’;
t_admediator = ‘mediator’;
t_aggregation = ‘aggregation’;
t_flowstart = ‘flowStart’;
t_flowfinal = ‘flowFinal’;
t_activityfinal = ‘activityFinal’;
t_aggregation = ‘aggregation’;
t_extraction = ‘extraction’;
t_fork = ‘fork’;
t_join = ‘join’;
t_decision = ‘decision’;
t_merge = ‘merge’;
t_operation = ‘operation’;
t_accepteventaction = ‘acceptEventAction’;
t_sendeventaction = ‘sendEventAction’;
t_inputpin = ‘inputPin’;
t_outputpin = ‘outputPin’;
t_controlflow = ‘controlFlow’;
t_dataflow = ‘dataFlow’;
t_guard = ‘guard’;
t_else = ‘else’;
t_startnode = ‘startNode’;
t_interruptions = ‘interrupts’;
t_objectnode = ‘objectNode’;
t_carriesconcept = ‘carriesConcept’;
t_partnerlink = ‘partnerLink’;
t_sslink = ‘ssLink’;
4 TRANSLATIONS

In the preceding sections, we gave high-level models (Sect. 2) and grammars (Sect. 3) for the different languages used on the three layers, i.e. ASM, Cashew, and UML 2 Activity Diagrams. This chapter provides translations between these languages. In Sect. 4.1, we give the translations from the UML 2 Activity Diagrams subset to ASM. Section 4.2 describes the translation between Cashew and activity diagrams. We thus provide a means to interlink the different notions of interface descriptions in DIP.

4.1 Activity Diagrams Subset to ASM

In this section, we outline the main idea of the translation from our activity diagrams subset to our extension to abstract state machines. A detailed version can be found in Appendix E.

4.1.1 Overview

Transition Rules.

The main idea for the translation of UML2 AD diagrams to ASM transition rules is to define a set of rules for each action that describes the circumstances under which the action is going to be executed. Due to the token flow semantics, this depends on the object flow that precedes its input pins. The result of an action relevant to the execution semantics is the appearance of tokens in its output pins and leaving flow edges. We reflect this in the effect of the rules.

Since in general, rules fire in parallel in ASM, we need to explicitly state that in each execution step only one rule should fire. Therefore, we use the choose construct which will contain all rules. Details in the appendix.

Concepts.

For the description of the token flow used in the definition of UML2 AD execution semantics, we use concepts and their instances in ASM. The intuitive counterpart of object nodes and pins in UML2 AD therefore are the concepts of an ontology in ASM. Tokens in UML2 AD will informally be reflected by instances in ASM. But, this mapping of aspects of UML2 AD and ASM is not of a natural homogeneity. Whereas tokens have a more dynamic nature by locally moving around between determined actions during the execution of a UML2 AD diagram, instances in a ASM transition rule system are of a more static nature in that they exist in the global state space of the attached ontology and are potentially accessible by every transition rule at every step of execution. This important difference needs to be addressed by the translation between both paradigms.

4.1.2 Translation Aspects for Basic Constructs

In this section, we start to describe our translation for basic constructs of UML2 AD. In Sect. 4.1.3, we will start to combine these basic constructs to handle more complex workflows.
Decision Node.

At a decision node (Fig. 4.1), an incoming object will be forwarded along one of its leaving edges $x = 1..n$. We therefore create a new instance of the corresponding concept in the updates. An optional condition $c_x$ will be directly reflected in our rules. Since the created set of rules must only fire once for one token, we need to delete the triggering instance from the ontology in the updates. Please note, that we create $n$ rules of the following form.

$$
\text{if } (_\# \text{memberOf } In) \text{ and } \text{asm}(c_x) \text{ then }
\begin{align*}
&\text{add } (_\# \text{memberOf } O_x) \\
&\text{delete } (_\# \text{memberOf } In)
\end{align*}
\text{endif}
$$

Merge Node.

A merge node (Fig. 4.2) directly forwards incoming objects to its leaving edge. The incoming objects are again represented as instances of the ontology concepts $In_1, In_2, \ldots, In_n$, the output object as instance of $O$. We use $n$ rules ($x = 1 \ldots n$) to model this behaviour in the following way.

$$
\text{if } (_\# \text{memberOf } In_x) \text{ then }
\begin{align*}
&\text{add } (_\# \text{memberOf } O) \\
&\text{delete } (_\# \text{memberOf } In_x)
\end{align*}
\text{endif}
$$

The delete statement hinders the rule from firing again if there is no unused token available. Important here is that we must not model the whole behaviour in one single rule by merging their conditions, like

$$
(_\# \text{memberOf } In_1) \text{ or } (_\# \text{memberOf } In_2) \text{ or } \ldots \text{ or } (_\# \text{memberOf } In_n)
$$

Such a rule would imply that its update is only executed once, even if multiple blocks of the disjunction hold. In contrast, the semantics of UML2 AD states that every token that is passed to a merge node is going to be forwarded, even if some appear at the same time.
Fork Node.

At a fork node (Fig. 4.3), incoming objects are forwarded to all leaving edges \( x = 1..n \). However, we cannot just delete the token that triggered this fork, since we must ensure that its other branches fire as well. Therefore, we introduce new state variables represented by instances of newly defined ontology concepts \( F_{O_1}, F_{O_2}, \ldots, F_{O_n} \).

Whenever a token passes a fork node, we implement its flow via one branch \( x \) by creating an instance of the corresponding concept \( O_x \). At the same time, we create instances for each branch the token did not pass (\( F_{O_i} \) where \( i = 1..n \) and \( i \neq x \)). Also, we remove the instance that triggered this fork node. We follow these instructions for each of the branches \( x = 1..n \).

\[
\text{if } (# \text{memberOf } In) \text{ then} \\
\quad \text{add } (# \text{memberOf } O_x) \text{ add } (# \text{memberOf } F_{O_1}) \\
\quad \text{add } (# \text{memberOf } F_{O_2}) \ldots \text{add } (# \text{memberOf } F_{O_{x-1}}) \\
\quad \text{add } (# \text{memberOf } F_{O_{x+1}}) \ldots \text{add } (# \text{memberOf } F_{O_n}) \\
\quad \text{delete } (# \text{memberOf } O_x) \\
\text{endif}
\]

Now, we still need to create rules for the branches that the token did not pass in the first place. In contrast to the former ones, these refer to the state variables (\( F_{O_1}, F_{O_2}, \ldots, F_{O_n} \)) instead of \( In \). For \( x = 1..n \), we create rules of the following form.

\[
\text{if } (# \text{memberOf } F_{O_x}) \text{ then} \\
\quad \text{add } (# \text{memberOf } O_x) \\
\quad \text{delete } (# \text{memberOf } F_{O_x}) \\
\text{endif}
\]

Join Node.

Tokens at a join node (Fig. 4.4) are passed along its leaving edge only if all incoming edges yield instances of their respective input concepts \( In_1, In_2, \ldots, In_n \). After firing, we need to delete the tokens consumed in the rule’s updates.

\[
\text{if } (# \text{memberOf } In_1) \text{ and } (# \text{memberOf } In_2) \\
\quad \text{and } \ldots \text{ and } (# \text{memberOf } In_n) \\
\quad \text{then add } (# \text{memberOf } O) \text{ delete } (# \text{memberOf } In_1) \\
\quad \text{delete } (# \text{memberOf } In_2) \ldots \text{delete } (# \text{memberOf } In_n) \\
\text{endif}
\]

4.1.3 Translation Aspects for Complex Structures

The descriptions given above for the translation of UML2 AD control nodes to ASM transition rules work in the basic case when an object flow consists of only the one control node described. However, the translation must be able to cope with an arbitrary combination of multiple control nodes in any object flow.
Traverse-to-Completion.

The chosen design of modeling rules out of the perspective of actions ensures capturing the “traverse-to-completion” semantics by collecting all conditions needed to be fulfilled for an action to execute in a single rule. This means, we are not modeling the single steps of the traversal of tokens, but allow a rule only to fire when all its required conditions are met. Thus, in a practical implementation for a sequential rule execution, a scheduler would choose one of the competing rules ready to fire, which would then consume all tokens at once using its update statements.

The semantics of the fork node as defined in [11, p. 363] yields a slight exception to this rule. An incoming token is copied at the fork node as long as an activity connected to one of its branches can fire. The copies are kept until the respective other connected activities become able to fire as well. This behavior is exactly resembled by the machines we create (cp. Sect. 4.1.2).

Preserving Semantics.

The outlined issues are covered by the formal translation given in Sects. E.2 and E.3. In this sense, the presented translation preserves the execution semantics of UML2 AD.

Example.

For illustration, we now give the ASM rules for the UML2 AD workflow containing the complex control node structure shown in Fig. 4.5. We assume that the objects A through H are only internally used and do not represent messages exchanged by the service.

![Figure 4.5: Complex Control Flow in UML2 AD](image-url)
4.2 Cashew and Activity Diagrams

Cashew is an ontological meta-model for workflow-oriented descriptions of semantic web service interfaces. Cashew describes these, using a set of core workflow patterns [24] as the unit of composition. UML Activity Diagrams provide, on the other hand, a higher-level process description language which is well-suited for workflow editing and representation as it benefits from the support of industrial-strength tools.

In this section we describe how cashew workflows can be transformed into their equivalent UML Activity Diagrams representation. The complete transformation is
presented in pseudo-code in Appendix F. These transformations are largely based on earlier work [25]. Still, the transformation presented here introduces some modifications in order to favour workflow composability as well as it makes the appropriate links to the WSMO meta-model. The reader is referred to Figure 2.4 for a complete list of the workflow operators provided by Cashew.

The **Sequential** operator has a direct and obvious transformation into UML Activity Diagrams by connecting sequenced performances with control flows. This operator does not present any difficulties with respect to its composability since it provides by definition two unique nexus, i.e. the first and the last performances, where other workflows can be connected.

Figure 4.6 illustrates how to represent the **Concurrent** operator in UML Activity Diagrams. This is in fact composed of a UML AND-split followed by an AND-join meaning that every thread must complete. The performances in between representing the different activities that have to be performed concurrently. Doing so paves the way for composing workflows by providing a central point where preceding and subsequent workflows can be connected as illustrated in the figure by the top and bottom grey activity boxes respectively. A complete definition of the transformation can be found in Section F.2.

![Figure 4.6: Concurrent pattern in UML](image)

**Interleaved** workflows are transformed in a similar fashion to that shown in [25] where signals are used as semaphores, see Figure 4.7 for an example with three interleaved performances $V$, $W$ and $X$. Before a performance can start, a signal has to be received. In UML Activity Diagrams if several receivers are ready to consume an event, only one action accepts the event. Therefore, once a performance is completed, a signal needs to be sent again so that other performances can be executed. As a consequence no order is established between the performances a priori, which leads to an arbitrary order of execution as dictated by the Interleaved Parallel Routing workflow pattern [24]. Finally, in order to support composability of workflows, signals are merged as shown in Figure 4.7 and in Section F.6.

A **XOR** workflow between performances $V$ and $W$ is depicted in Figure 4.8. A decision node, based on the condition specified by the axiom $a$, connects both performances and a final merge node ensures the final result is composable.
The **While** and **Until** constructs, depicted in Figures 4.9 and 4.10 respectively, are both very similar to the XOR-type workflow. The main difference is that they do not require a merge node as a central point for connecting to subsequent workflows, since there is only one outgoing path. This is illustrated by means of the grey box connected to the ‘else’ branch. Both constructs differ in that the performance $V$ is always executed at least once in the Until pattern, whereas it might never be executed in While-type workflows. In fact as shown in the figures, the preceding performance connects to the decision node in the case of the While construct as opposed to Until-type workflows where it directly links to the performance $V$.

The **Deferred Choice** pattern is shown in Figure 4.11 and a transformation algorithm is described in Section F.7. Deferred Choice starts concurrently as the control node is split between various Atomic Performances. Atomic Performances can be either a Perform Goal, a Perform Send or a Perform Receive. These Atomic Performances will determine at runtime the branch to be executed. Once a branch is selected, the subsequent performance will be executed preempting the other possible branches. To support this, Atomic Performances are within an **interruptible region** and interrupting edges connect them to their respective subsequent performances, i.e. performances $V$ and $X$ in Figure 4.11. It is worth noting that these performances are optional, e.g. the action to be performed could simply be sending a message. Finally, for the sake of composability all performances connect to a ‘merge node’ which represents the unique nexus where following performances are to be connected.
Figure 4.9: While pattern in UML

Figure 4.10: Until pattern in UML
In a similar fashion to the Deferred Choice pattern, **Deferred While** (see Figure 4.12) and **Deferred Until** (see Figure 4.13) are based on the use of interruptible activity regions. It is however worth noting that although the Deferred Choice pattern can include as many branches as desired, both Deferred While and Deferred Until only include two branches, one being the iteration body and the other representing the concluding Atomic Performance. Again, the difference between Deferred While and Deferred Until lies in that the iteration branch, i.e. the Performance V, is executed at least once in the Deferred Until pattern. Therefore, and to cater for composability, the Deferred While pattern includes a merge node where both the iterating performance and preceding workflows connect to. Conversely, in the Deferred Choice the incoming control flow is directly connected to the Performance V ensuring that it is at least executed once. Finally both patterns include a concluding branch with an Atomic Performance which also represents the outgoing nexus. The reader is referred to Section F.9 and Section F.8 for their corresponding transformation algorithms.

Finally, in order to define the dataflow each send and receive performance is associated with a UML action with pins that represent the atomic parts of the message, see Figures 4.11(a) and 4.11(b). In this way dataflow can be represented by connecting pins together with UML object flow edges.
Figure 4.11: Deferred Choice pattern in UML

Figure 4.12: Deferred While pattern in UML

Figure 4.13: Deferred Until pattern in UML
5 Example

As an example of the use of the three-level model, we adapt a scenario from the telecoms industry use case. This concerns a composite business process to order three related products as a ‘bundle’; a modem, the network connection and a PC. These orders are checked in a strict order due to the likelihood of failure, starting with the network, where many consumer lines may be incapable of supporting certain types of connection — in particular DSL-based ones.

Each of the three services to check the respective product has a choreography as diagrammed in Figure 5.1(a), and each service to confirm the order has a choreography as diagrammed in Figure 5.1(b). We note that while each has a receive event followed by some response event, the ModemRequest service has an internal (XOR) choice between a subsequent failure or success event.

The representation of this choreography is shown at each of the three-levels in Figure 5.2. We note that the Activity Diagrams version directly encodes these representations, that the Cashew workflow directly represents the receive sequentially followed by this xor-choice between two sends, and that the ASM uses all three recursive transition rule types and direct manipulations on the state signature (since this is a choreography). These three representations are shown in an extension of the standard DIP tool, WSMO Studio ¹, supporting the extended WSML grammar via an extension to the standard object model for WSMO/L, WSMO4J².

A section of the orchestration between the six services in shown as an Activity Diagram in Figure 5.3. We note that the choreography’s internal choice has been turned into a deferred choice in the orchestration since, as a client to these services, the orchestration engine does not resolve this choice directly, but waits for the response message in order to choose which branch to follow.

A full treatment of this example, including an orchestration on all three levels, has been produced and tested but space prevents us from reproducing this here.

¹http://www.wsmostudio.org
²http://wsmo4j.sourceforge.net
Figure 5.1: Example Choreographies as Activity Diagrams

Figure 5.2: Example Choreography on 3 levels in WSML
Figure 5.3: Example Orchestration as an Activity Diagram
6 CONCLUSION

In this deliverable, we have finalised the work started in the deliverables D3.5 “An Ontology for Web Service Choreography”, D3.4 “An Orchestration and Business Process Ontology”, and their common appendix titled “DIP Interface Description Ontology” (DIO). We present our full model for the ‘three-level’ semantic descriptions of behaviour for semantic Web service specifications. The main contributions of this document can be summarised as follow.

New Description Layer. The workflow description language Cashew was introduced as a new layer into the stack of languages used. It is often argued that UML2 Activity Diagrams (AD) are ambiguous due to their lack of any formal semantics. Cashew provides such a well-defined way of expressing workflows. With this new layer introduced, we provide support for multiple parties encountering semantic Web service descriptions: The business-oriented users who need a widely accepted and well-known language to describe their business processes; more technical-oriented integration experts who demand a well-founded and unambiguous high-level workflow language to perform reasoning and integration tasks; and execution infrastructures that require a technical representation which enforces the least commitment possible to any specific higher-level constructs, but can rather be executed directly. Therefore, for the first class of users, we provide AD, for the second Cashew, and for the last we foresee to use ASM.

Transformations. In this deliverable, we provide two transformations (see Sect. 4): the transformation between Cashew and activity diagrams, and the revised transformation from activity diagrams to ASM. These transformations are to be understood as a basic set giving a proof-of-concept. They show the interrelation of the different languages and also already allow to perform basic tasks with choreographies in DIP. That is, the proposed languages in combination with the transformations given allow a system integration user to define and reason about collaborative processes in Cashew. And either execute the behaviours in the IRS-III using Cashew, or transform to abstract state machines for execution in the WSMX environment.

New Intent. Whereas the deliverables D3.4, D3.5 and the DIO aimed at showing the application of integrated choreography and orchestration descriptions in DIP, this deliverable targets at proposing a concrete extension to the WSMO specification. Therefore, the Sects. 2.1 and 3 contain our suggestions for adaption of WSMO in order to interoperable with the DIP understanding of choreography and orchestration.

Metamodels. We defined the languages in this deliverable by consistently giving their metamodels in MOF (see Sect. 2). Thus, the ontology-based metamodel of AD, which was called “AD-O” in the DIO, was replaced in this deliverable by a MOF-based specification. The newly added Cashew also was defined based on MOF. For completeness, Sect. 2.1 also contains a MOF-based representation of ASM including our proposal for the extension of WSMO.
Grammars. In addition to the conceptual metamodels, this deliverable also defines concrete syntaxes for the expression of choreography and orchestration languages (see Sect. 3). The proposed WSML extension is well aligned with the corresponding proposed WSMO extension. For the grammar definition, we use the same BNF notation as used in the WSML specifications. Using the same BNF notation should make incorporation into existing definitions as easy and clear as possible.

During this work we have also carried a great deal of implementation work. In particular: WSMO4J, the object model for WSML, was extended with a parser for our new grammar; WSMO Studio, the standard platform for the creation of WSML descriptions, had an experimental version created with this parser, as shown in Figure 5.3; WSMX, the open-source implementation of WSMO partially developed during DIP, based its orchestration component on our extended ASM grammar; and IRS-III, the second WSMO implementation used during DIP, was extended with an implementation of Cashew-based orchestration.

At the same time we have proposed notions from our approach to the WSMO Working Group, in particular the notion of mediator-based dataflow has been the subject of a new WSMO deliverable [17]. We have also applied our three-level approach to the description of choreography and orchestration in the OASIS Working Group on Semantic Execution Environment, wherein tools like the IRS-III and WSMX are themselves described, at the architectural level, in WSMO models.
REFERENCES


[18] D. Roman and Scicluna J. (eds.). Ontology-based Choreography of WSMO Services. WSMO Deliverable D14, 2006. final version 0.3; available from http://www.wsmo.org/TR/d14/v0.3/. 1, 6, 18, 42


A MOF Formalisation of Extended WSMO Meta-model

We reproduce here the relevant parts of the MOF definitions of the WSMO meta-model from WSMO Deliverable 2 [20], together with our extensions and changes to WSMO Deliverable 14 [18][23], and the extensions from WSMO Deliverable 15.1 [17].

//Reproduced from WSMO D2:

```plaintext
Class wsmoTopLevelElement
  hasNonFunctionalProperties type nonFunctionalProperties

Class webService sub Class wsmoTopLevelElement
  importsOntology type ontology
  usesMediator type {ooMediator, wwMediator}
  hasCapability type capability multiplicity = single–valued
  hasInterface type interface

Class goal sub Class wsmoTopLevelElement
  importsOntology type ontology
  usesMediator type {ooMediator, ggMediator}
  requestsCapability type capability multiplicity = single–valued
  requestsInterface type interface

Class interface
  hasNonFunctionalProperties type nonFunctionalProperties
  importsOntology type ontology
  usesMediator type ooMediator
  hasChoreography type choreography
  hasOrchestration type orchestration

Class mediator sub Class wsmoTopLevelElement
  importsOntology type ontology
  hasSource type {ontology, goal, webService, mediator}
  hasTarget type {ontology, goal, webService, mediator}
  hasMediationService type {goal, webService, wwMediator}

//Adapted from WSMO D2:

Class wgMediator sub Class mediator
  usesMediator type ooMediator
  hasSource type {webService, goal, wgMediator, ggMediator}
  hasTarget type {webService, goal, ggMediator, wgMediator}
  appliesChoreography type choreography
```

¹Note that we drop ‘hasState’ from ‘stateSignature’, since this is not a meta-model for the grammar but a meta-model to carry the behavioural semantics of ASMs.

²Note also that logicalExpression itself isn’t defined in the WSMO meta-model and has only a grammatical WSML definition. There are definitions for nonFunctionalProperties, ontology, concept and relation, but we elide this since they are unchanged.

³Finally, note that transition rules are also not given a meta-model, only a grammar, in WSMO D14, but we reverse engineer a meta-model, correcting for the over-generality of ‘pipedRules’, in order to extend this.
//Reproduced from WSMO D16:

Class stateSignature
  hasNonFunctionalProperties type nonFunctionalProperties
  importsOntology type ontology
  usesMediator type ooMediator
  hasStatic type mode
  hasIn type mode
  hasOut type mode
  hasShared type mode
  hasControlled type mode

Class mode sub−Class {concept, relation}
  hasGrounding type grounding

//Adapted from WSMO D16:

Class choreography
  hasNonFunctionalProperties type nonFunctionalProperties
  hasStateSignature type stateSignature
  hasChoreographyFormalism type choreographyFormalism

//Introduced here

Class orchestration
  hasNonFunctionalProperties type nonFunctionalProperties
  hasStateSignature type stateSignature
  hasOrchestrationFormalism type orchestrationFormalism

Class choreographyFormalism

Class orchestrationFormalism

Class asmChoreography sub−Class choreographyFormalism
  hasTransitionRules type transitionRules

Class cashewChoreography sub−Class choreographyFormalism
  hasCashewWorkflow type cashewWorkflow
  //Defined in Section 3.4

Class adChoreography sub−Class choreographyFormalism
  hasActivityDiagram type activityDiagram
  //Defined in Section 3.5

Class asmOrchestration sub−Class orchestrationFormalism
  hasTransitionRules type orchestrationTransitionRules

Class cashewOrchestration sub−Class orchestrationFormalism
  hasCashewWorkflow type cashewWorkflow

Class adOrchestration sub−Class orchestrationFormalism
  hasActivityDiagram type activityDiagram
//The following are unstated in WSMO D14 and we allow ourselves to
//make transitionRule a sub-class to allow choreography transition
//rules to be a subset of those allowed for orchestration

Class transitionRules
  hasTransitionRule type transitionRule

Class transitionRule sub Class of orchestrationTransitionRule

Class if sub Class transitionRule
  hasCondition type logicalExpression multiplicity = single-valued
  then type transitionRule

Class forAll sub Class transitionRule
  hasVariableList type VariableList
  with type logicalExpression multiplicity = single-valued
  do type transitionRule

Class choose sub Class transitionRule
  hasVariableList type VariableList
  with type logicalExpression multiplicity = single-valued
  do type transitionRule

Class pipedRules
  hasTransitionRule type transitionRule

//Reproduced from WSMO D15.1

Class perform

Class performAchieveGoal sub Class of perform
  hasGoal type goal

Class performInvokeService sub Class of perform
  hasWebService type webService

Class performApplyMediation sub Class of perform
  hasMediator type mediator

Class performReceive sub Class of perform
  hasSource type mode

Class performSend sub Class of perform
  hasTarget type mode

Class performWorkflow sub Class of perform
  hasWorkflow type workflow

Class workflow

Class ppMediator sub Class of mediator
  hasSource type {perform, ppMediator}
  hasTarget type {perform, ppMediator}
  usesMediator type ooMediator

Class pfMediator sub Class of mediator
  hasSource type {workflow, perform, ppMediator}
hasTarget type \{\text{workflow, perform, ppMediator}\}
usesMediator type ooMediator

\textbf{Class} orchestrationTransitionRules
hasOrchestrationTransitionRule type orchestrationTransitionRule

\textbf{Class} orchestrationTransitionRule

\textbf{Class} performRule sub–\textbf{Class} orchestrationTransitionRule
hasPerformance type \{\text{performAchieveGoal, performInvokeService, performApplyMediation}\}
B OCML FORMALISATION OF CASHEW META-MODEL

;;; Mode: Lisp; Package: ocml

;;; Author: Barry Norton, KMi, Open University
;;; Contributors: Simon Foster, DCS, University of Sheffield
;;; Carlos Pedrinaci, KMi, Open University

(in-package "OCML")

(in-ontology cashew-ontology)

(def-class workflow (problem-solving-pattern invokable-entity)
  "The superclass to all Cashew workflow types. cf. 'Process' in the original Cashew-S paper."
  ((has-dataflow-mediator :type dataflow-mediator)))

(def-class dataflow-mediator (mediator)
  "The superclass to all dataflow connection types in Cashew workflows")

(def-class pp-mediator (dataflow-mediator)
  "A dataflow connection between two peer performances (i.e. composed in same workflow). cf. 'Connect' in original Cashew-S paper."
  ((has-source-component :type perform :cardinality 1)
   (has-target-component :type perform :cardinality 1)))

(def-class pf-mediator (dataflow-mediator)
  "A dataflow connection from a performance to its parent workflow. cf. 'Producer' in original Cashew-S paper."
  ((has-source-component :type (or perform workflow) :cardinality 1)
   (has-target-component :type (or perform workflow) :cardinality 1)))

(def-class perform ()
  "The superclass to all types of performance in Cashew, being the unit of composition in a workflow")

(def-class atomic-perform (perform)
  "Those performs which can be performed atomically")

(def-class receive (atomic-perform)
  "Accepts a message"
  ((has-output-role :type role))) ;; :cardinality 1)))

(def-class send (atomic-perform)
"Sends a message"
((has-input-role :type role))) ;; :cardinality 1)))

(def-class perform-goal (atomic-perform)
  "A performance of a goal"
  ((has-goal :type goal :cardinality 1))))

(def-class perform-workflow (perform)
  "A performance of a (sub-)workflow"
  ((has-workflow :type workflow :cardinality 1))))

(def-class sequential (workflow)
  "A workflow performing its (list of) components in a sequence"
  ((has-perform-list :type perform-list :cardinality 1))))

(def-class perform-list (list)
  ((has-element-type :value perform)))

(def-class concurrent (workflow)
  "A workflow performing its components concurrently.
  cf. SplitJoin in original Cashew-S paper."
  ((has-perform-list :type perform-list :cardinality 1))))

(def-class interleaved (workflow)
  "A workflow performing its components interleavedly.
  cf. AnyOrder in original Cashew-S paper."
  ((has-perform-list :type perform-list :cardinality 1))))

(def-class xor-choice (workflow)
  "A workflow that resolves a choice between alternatives, using an
  explicit condition, and performs exactly one"
  ((decision :type unary-kappa-expression :cardinality 1)
   (if-true :type perform :cardinality 1)
   (if-false :type perform :cardinality 1))))

(def-class deferred-choice (workflow)
  "A workflow that resolves a choice between alternatives, using their
  readiness, and performs exactly one"
  ((has-choice-list :type choice :cardinality 1))))

(def-class choice ()
  "One choice within a deferred-choice"
  ((has-atomic-perform :type atomic-perform :cardinality 1))
  ((enables :type perform :min-cardinality 0 :max-cardinality 1))))

(def-class xor-while (workflow)
  "A workflow that performs some number of iterations of the body
workflow, according to the value of an explicit logical decision
  ((decision :type unary-kappa-expression :cardinality 1)
   (body :type perform :cardinality 1)))

(def-class deferred-while (workflow)
  "A workflow that performs some number of iterations of its body,
   according to the readiness of an alternative"
  ((break :type perform :cardinality 1)
   (body :type perform :cardinality 1)))

(def-class xor-until (workflow)
  "A workflow that performs some positive number of iterations of its
   body, subsequent iterations according to the value of an explicit
   logical decision"
  ((body :type perform :cardinality 1)
   (decision :type unary-kappa-expression :cardinality 1)))

(def-class deferred-until (workflow)
  "A workflow that performs some positive number of iterations of its
   body, subsequent iterations according to the readiness of some
   alternative"
  ((body :type perform :cardinality 1)
   (break :type perform :cardinality 1)))

(def-relation has-perform-list (?thing ?perform-list)
  "This definition generalises the notion of
   'having a perform list' to classes as well
   as workflow instances."
  :sufficient (or (and (instance ?thing)
   (has-perform-list (the-parent ?thing)
    ?perform-list))
   (and (class ?thing)
    (member ?perform-list
     (all-class-slot-values
      ?thing has-perform-list)))))

(def-relation has-perform (?thing ?perform)
  "This definition generalises the notion of
   'having a component perform' to all workflow
   classes, not just those with this as a direct attribute,
   as well as to their instances"
  :sufficient (or (and (instance-of ?thing 'workflow)
   (has-perform (the-parent ?thing) ?perform))
   (or (and (subclass-of ?thing 'sequential)
    (member ?perform (the-class-slot-value
     ?thing has-perform-list)))
    (or (subclass-of ?thing 'int-choice)
(or (= ?perform (the-class-slot-value ?thing if-true))
   (= ?perform (the-class-slot-value ?thing if-false))))
(or (and (or (subclass-of ?thing 'int-while)
              (subclass-of ?thing 'int-until))
         (= ?perform (the-class-slot-value ?thing body)))
 (or (and (or (subclass-of ?thing 'ext-while)
              (subclass-of ?thing 'ext-until))
         (or (= ?perform (the-class-slot-value ?thing break))
             (= ?perform (the-class-slot-value ?thing body))))
 (or (and (subclass-of ?thing 'workflow)
          (member ?perform (all-class-slot-values ?thing has-perform)))))))
C MOF Formalisation of Cashew Metamodel

Class cashewWorkflow
  // hasNonFunctionalProperties type nonFunctionalProperties
  hasStateSignature type stateSignature
  hasWorkflowPattern type workflowPattern multiplicity = single-valued
  hasDataflowMediators type dataflowMediators

Class cashewPerformanceList
  hasPerformanceList type cashewPerformanceList
  hasPerformance type {performReceive, performSend, performAchieveGoal, performWorkflow}
  multiplicity = single-valued

Class workflowPattern
  // hasNonFunctionalProperties type nonFunctionalProperties

Class sequential sub-Class workflowPattern
  hasPerformanceList type cashewPerformanceList multiplicity = single-valued

Class unordered sub-Class workflowPattern
  hasPerformance type {performReceive, performSend, performAchieveGoal, performWorkflow}

Class interleaved sub-Class unordered

Class concurrent sub-Class unordered

Class choice sub-Class workflowPattern

Class internalChoice sub-Class choice
  hasCondition type logicalExpression

Class xor sub-Class internalChoice
  ifBranch type {performReceive, performSend, performAchieveGoal, performWorkflow}
  thenBranch type {performReceive, performSend, performAchieveGoal, performWorkflow}

Class while sub-Class internalChoice
  hasPerformance type {performReceive, performSend, performAchieveGoal, performWorkflow}

Class until sub-Class internalChoice
  hasPerformance type {performReceive, performSend, performAchieveGoal, performWorkflow}

Class externalChoice sub-Class choice

Class deferredChoice sub-Class externalChoice
  hasDeferredChoiceBranch type deferredChoiceBranch

Class deferredChoiceBranch
  enabler type {performReceive, performSend, performAchieveGoal}
  multiplicity = single-valued
  enabled type {performReceive, performSend, performAchieveGoal, performWorkflow}
  multiplicity = single-valued

Class deferredWhile sub-Class externalChoice
  unlessBranch type {performReceive, performSend, performAchieveGoal}
  repeatBranch type {performReceive, performSend, performAchieveGoal, performWorkflow}
D MOF Formalisation of Activity Diagram Subset Meta-model

The MOF version of the activity diagram language can be diagrammed in UML as shown in Figure D.1, and is formalised as follows.

Class activityDiagram
  hasNonFunctionalProperties type nonFunctionalProperties
  hasActivityGroup type activityGroup multiplicity = single-valued
  startNode type activityNode multiplicity = single-valued

Class activityGroup
  hasNonFunctionalProperties type nonFunctionalProperties
  hasActivityNode type activityNode
  hasActivityEdge type activityEdge
  hasActivityGroup type activityGroup

Class interruptibleRegion sub Class activityGroup

Class activityNode
  hasNonFunctionalProperties type nonFunctionalProperties

Class controlNode sub Class activityNode

Class decision sub Class controlNode

Class merge sub Class controlNode

Class flowStart sub Class controlNode

Class flowFinal sub Class controlNode

Class activityFinal sub Class controlNode

Class join sub Class controlNode

Class fork sub Class controlNode

Class objectNode sub Class activityNode
  ssLink type mode
  carriesConcept type concept

Class pin sub Class objectNode

Class receiveEventAction sub Class objectNode
  partnerLink type sendEventAction

Class sendEventAction sub Class objectNode
  partnerLink type receiveEventAction

Class actionNode sub Class activityNode
inputPin type pin
outputPin type pin

Class generalAction sub-Class actionNode

Class extraction sub-Class actionNode
  inputPin type pin multiplicity = single-valued
definition type goal

Class aggregation sub-Class actionNode
  outputPin type pin multiplicity = single-valued
definition type goal

Class operation sub-Class actionNode
definition type goal

Class mediation sub-Class actionNode
definition type ooMediator

Class activityEdge
  hasNonFunctionalProperties type nonFunctionalProperties
  source type activityNode
target type activityNode
  hasGuard type {else, logicalExpression }
  interrupts type interruptibleRegion

Class else

Class dataFlow sub-Class activityEdge

Class controlFlow sub-Class activityEdge
Figure D.1: MOF Activity Diagrams Meta-model
E Translation Rules: Activity Diagrams Subset to ASM

E.1 Overview & Open Issues

This section gives the full set of rules that define the translation from the activity diagrams subset to ASM as announced in Sect. 4.1 plus explanations for understanding. Since the grammar given in Sect. 3.5 is in a preliminary state, this translation is not yet defined in a strict way in terms of its expressions.

However, the definitions given below are concise as they are grounded to mathematics. A computer-based implementation should therefore be easily deductible from the grounding and the rules given. During the explanations we use wherever possible the nomenclature from Sect. 3.5.

As connotated in Table E.1, the current version of the described translation algorithm covers all basic constructs of ADs. However, some enhanced constructs listed and those AD constructs being not reflected in the table are not yet being addressed. Also, flows which are forked and joined without entering any ActionNode inbetween, can not be handled correctly (“Freak flow” in Table E.1). These issues are open for future research.

<table>
<thead>
<tr>
<th>Construct</th>
<th>In current version</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Signal (in, out)</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Fork</td>
<td>yes</td>
<td>Set instead of FIFO semantics</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(cp. [11, p. 363, bottom])</td>
</tr>
<tr>
<td>Join</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Decision</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Merge</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Complex flow</td>
<td>yes</td>
<td>Preserves token flow semantics</td>
</tr>
<tr>
<td>“Freak” flow</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Object node</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Pin (in, out)</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Edge parameter</td>
<td>yes</td>
<td>For DecisionNodes only</td>
</tr>
<tr>
<td>Interruptible region</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Structured node</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>

Table E.1: Features of activity diagrams to ASM Translation

E.2 Prerequisites

E.2.1 Ontology

We define a WSMO ontology in WSML that will be used later on for expressing the dynamic aspects of the activity diagrams in the ASM choreography representation.

Rule 1 (Ontology). For every activity diagram, a local ontology localOntology will be created. Initial concepts are as follows. It will become updated during the description
of the algorithm.

\[\text{concept } \text{InternalComm}\]
\[\text{concept } \text{State} \text{subConceptOf} \text{InternalComm}\]
\[\text{concept } \text{ControlFlow} \text{subConceptOf} \text{State}\]
\[\text{concept } \text{Branch} \text{subConceptOf} \text{InternalComm}\]

E.2.2 Definitions

Throughout the following sections, we will use some functions to precisely express the translation of activity diagrams to ASM. The signatures of the single functions are given below.

Rule 2 (Element). We define the set “Element” as the union of all ControlNodes, ObjectNodes, and ActionNodes.

Rule 3 (Complex flow). We define the term “complex flow”. It means a set of all ControlNodes and ObjectNodes that can be reached from any of its members using only directly connecting ActivityEdges.

Rule 4 (Structure). For every Element \(e\) of a complex flow, we define a set of direct successors \(\text{Suc}(e)\) and a set of direct predecessors \(\text{Pre}(e)\) which contain the appropriate set of Elements with respect to their order in the activity diagram complex flow.

\[
\text{Suc} : \text{Element} \to 2^{\text{Element}} \\
\text{Pre} : \text{Element} \to 2^{\text{Element}}
\]

\[
\text{Suc}(e_{\text{pre}}) := \{ e : \text{edge} \in \text{ActivityEdge}, e_{\text{pre}} = \text{isInputOf}(\text{edge}), e = \text{isOutputOf}(\text{edge}) \}
\]

\[
\text{Pre}(e_{\text{suc}}) := \{ e : \text{edge} \in \text{ActivityEdge}, e = \text{isInputOf}(\text{edge}), e_{\text{suc}} = \text{isOutputOf}(\text{edge}) \}
\]

Rule 5 (Navigation). For every Element \(e_u, e_v\) of a complex flow, we define direct successor \(\text{suc}_1, \text{suc}_2, \ldots, \text{suc}_m\) and predecessor functions \(\text{pre}_1, \text{pre}_2, \ldots, \text{pre}_n\) which return an appropriate Element \((e_x, e_y)\) with respect to their order in the activity diagram complex flow. The functions \(\text{suc}\) and \(\text{pre}\) are abbreviations for the case of sets of cardinality one.

\[
\text{suc} : \text{Element} \to \text{Element} \\
\text{pre} : \text{Element} \to \text{Element}
\]

\[
e_x = \text{suc}_x(e_u) \iff e_x \in \text{Suc}(e_u), \bigcup e_x = \text{Suc}(e_u), 1 \leq x \leq m
\]

\[
e_y = \text{pre}_y(e_v) \iff e_y \in \text{Pre}(e_v), \bigcup e_y = \text{Pre}(e_v), 1 \leq y \leq n
\]

\[
\text{suc} : \text{Element} \to \text{Element}, \quad e_v = \text{suc}(e_u) \iff e_v \in \text{Suc}(e_u), \{ e_v \} = \text{Suc}(e_u)
\]

\[
\text{pre} : \text{Element} \to \text{Element}, \quad e_w = \text{pre}(e_t) \iff e_w \in \text{Suc}(e_t), \{ e_w \} = \text{Suc}(e_t)
\]

Rule 6 (Concepts). For every ObjectNode, there is a mapping to a Concept of the ontology.

\[\text{concept} : \text{ObjectNode} \to \text{Concept}\]
For every ActionNode, we define a sequence for their sets of InputPins and OutputPins.

\[
\begin{align*}
\text{inputPins}_x : \text{ActionNode} & \rightarrow \text{InputPin}, \quad p_x = \text{inputPins}_x(a) \iff p_x \in \text{inputPins}(a), \\
\bigcup p_x = \text{inputPins}(a), & \quad x = 1.. | \text{inputPins}(a) |, \quad a \in \text{ActionNode} \\
\text{outputPins}_y : \text{ActionNode} & \rightarrow \text{InputPin}, \quad p_y = \text{outputPins}_y(b) \iff p_y \in \text{outputPins}(b), \\
\bigcup p_y = \text{outputPins}(b), & \quad y = 1.. | \text{outputPins}(b) |, \quad b \in \text{ActionNode}
\end{align*}
\]

For the translation of activity diagrams \textit{GeneralActions} to ASM rules, we will make use of the following definitions.

\textbf{Rule 8 (Rule Functions).} For every \textit{GeneralAction}, \textit{AcceptEvent}, and \textit{SendEvent}, we will define a set of rules \(R\). For its definition, the \textit{actionFlow} and \textit{objectFlow} function will be used. \(C \times U\) denotes a set of pairs of conditions \(\in C\) and updates \(\in U\) of ASM, \(n\) is an arbitrary number \(\in \mathbb{N}\). We shortly type \(E\) for \textit{Element}, here.

\[
\begin{align*}
\text{rules} : \quad & \text{GeneralAction} \cup \text{AcceptEvent} \cup \text{SendEvent} \rightarrow 2^R \\
\text{actionFlow} : \quad & \text{GeneralAction} \cup \text{AcceptEvent} \cup \text{SendEvent} \rightarrow 2((2^{E \times E})^n) \\
\text{objectFlow} : \quad & \text{InputPin} \times 2^{E \times E} \times 2^C \times 2^{\text{Concept}} \times 2^{\text{Concept}} \rightarrow 2^{C \times U}
\end{align*}
\]

\textbf{E.2.3 Normal Form}

In order to keep the translation algorithm of Sect.E.3 simple, we reduce some of the redundancies entailed in activity diagrams. The resulting normal form of activity diagram will be used in the following of this description. The creation of this normal form from the original activity diagram modeling is defined below.

\textbf{Remove Implications.}

“An action can only begin execution when all incoming control edges have tokens \([\ldots]\)” [11, p. 301] Thus, the interpretation of two \textit{ControlFlows} leading to an \textit{ActionNode} is defined as an implicit join.

\textbf{Rule 9.} All control leading to the same \textit{ActionNode} \(a\) will be joined in a \textit{JoinNode} whose output directly leads to \textit{ActionNode} \(a\).

“Tokens offered by the source node are all offered to the target node” of an \textit{ObjectFlow} [11, p.376]. Also, “an action can only begin execution when \([\ldots]\) all input pins have object tokens.” [11, p.376] We interpret these statements in the way that multiple \textit{ObjectFlows} entering a \textit{Pin} are being implicitly merged. We assume the same behavior at \textit{ObjectNodes}.

\textbf{Rule 10.} All \textit{ObjectFlows} leading to the same \textit{InputPin} \(p\) will be joined in a \textit{MergeNode} whose output directly leads to \textit{InputPin} \(p\).

\textbf{Rule 11.} All \textit{ObjectFlows} leading to the same \textit{ObjectNode} \(o\) will be joined in a \textit{MergeNode} whose output directly leads to \textit{ObjectNode} \(o\).
“When the execution of an action is complete, it offers tokens in its outgoing control edges […]” [11, p. 301] Multiple ControlFlows leaving an ActionNode thus have implicit fork semantics.

**Rule 12.** All ControlFlows leaving the same ActionNode \(a\) will leave from a ForkNode whose input directly stems from ActionNode \(a\).

“A token in an object node can traverse only one of the outgoing edges.” [11, p. 381] “Two object flows may have the same object node as source. In this case the edges will compete for objects. Once an edge takes an object from an object node, the other edges do not have access to it.” [11, p. 376] Therefore, multiple ControlFlows leaving a Pin or ObjectNode have an implicit decision semantics. We also make this semantics explicit.

**Rule 13.** All ObjectFlows leaving the same Pin \(p\) will leave from a DecisionNode whose input directly stems from Pin \(p\).

**Rule 14.** All ObjectFlows leaving the same ObjectNode \(o\) will leave from a DecisionNode whose input directly stems from ObjectNode \(o\).

**Divide Combined Nodes.**

In complex flows, so-called control nodes can be used to route these flows in certain ways. For convenience, some abbreviations were defined in our activity diagrams language for specific pairs of those nodes. We will transform these syntactic abbreviations back into their basic control node representations.

**Rule 15.** A combined decision and merge control node will be split to a DecisionNode and a MergeNode.

**Rule 16.** A combined fork and join control node will be split to a ForkNode and a JoinNode.

**Express ControlFlows as ObjectFlows.**

ControlFlow in our activity diagrams language can be seen as a specific kind of ObjectFlow where the object of a transformed ControlFlow is an (artificial and materialised) state value. We call the type of the state values ‘State’.

**Rule 17.** Every ControlFlow is converted to an ObjectFlow by introducing an ObjectNode \(o\) of type State in this ControlFlow. Thus, each ControlFlow edge \(e_{\text{orig}}\) in a complex flow will be substituted by two new ActivityEdges \((e_1, e_2)\) and \(o\) as follows.

\[
\text{isOutputOf}(e_1) := \text{isOutputOf}(e_{\text{orig}}), \quad \text{isInputOf}(e_1) := o \\
\text{isOutputOf}(e_2) := o, \quad \text{isInputOf}(e_2) := \text{isInputOf}(e_{\text{orig}}) \\
\text{concept}(o) := \text{localNamespace}\#\text{ControlFlow}
\]
Transform *ObjectNodes*\(^1\) to *Pins*.

*ObjectFlows* in activity diagrams can be defined in two ways. One is to use *ObjectNodes* anywhere in a set of successive *ObjectFlows*. The second is to use *Pins* in each very source and target of a set of successive *ObjectFlows*. Since the *Pin* notation is more expressive than the *ObjectNode* notation, we transform the *ObjectNode* to the *Pin* notation.

**Rule 18.** For each successive *ObjectFlow* of maximum length, we add *Pins* to its very source and target *ActionNodes* and remove the *ObjectNodes*. The type (*concept*) of a new *Pin* \(p_{\text{new}}\) is defined as the union of the types of all former *ObjectNodes* \(o_1, o_2, \ldots, o_m\) that are (directly) reachable through *ActivityEdges* and *ControlNodes* from the *Pin* \(p_{\text{new}}\).

\[
\text{concept} \text{NewConcept}_x \text{subConceptOf} \{ \text{concept}(o_1), \text{concept}(o_2), \ldots, \text{concept}(o_m), \\
\text{concept}(p_1), \text{concept}(p_2), \ldots, \text{concept}(p_n) \} \quad \text{where} \ x \in \mathbb{N}
\]

\[
\begin{align*}
\text{concept}(p_{\text{new}}) & := \text{localNamespace}\#\text{NewConcept}_x \\
\text{such that} \quad & \forall i, j \in \mathbb{N} : \ \text{concept}(p_i) = \text{concept}(p_j) \iff i = j \\
\text{and} \quad & \forall k, l \in \mathbb{N} : \ \text{NewConcept}_k = \text{NewConcept}_l \iff k = l
\end{align*}
\]

**E.3 Translation**

Here, we give a translation of the activity diagram model to the ASM language. We therefore assume the prerequisite steps from Sect.E.2 have been executed. Thus, simply spoken, the activity diagram model we base on only consists of *ObjectFlows* with only basic *ControlNodes*. When we say ‘activity diagram (AD)’ from here on, we refer to the adapted model as described above, unless otherwise stated.

**E.3.1 Global Structure**

**Rule 19.** Each choreography or orchestration specified in AD will be converted to ASM according to the following scheme.

\[
\begin{align*}
\text{webService} & \text{ serviceName} \\
\text{interface} & \text{ interfaceName} \\
\text{type} & \text{ chorOrchName} \\
\text{importsOntology} & \text{ localOntology} \\
\text{stateSignature} & \begin{align*}
\text{in} & \text{ inConcept}_1, \text{ inConcept}_2, \ldots, \text{ inConcept}_m \\
\text{out} & \text{ outConcept}_1, \text{ outConcept}_2, \ldots, \text{ outConcept}_n
\end{align*} \\
\text{abstractStateMachine} & \begin{align*}
\text{choose} & X \in \text{allRules} \\
& X
\end{align*}
\]

\(^1\)We understand *ObjectNodes* only in this section as the set *ObjectNodes* \(\setminus\) *Pins*. 

Rule 20. We define the type of the described interface as either choreography or orchestration. This is needed as an input to the translation algorithm.

\[
\text{type} \in \{ \text{choreography, orchestration} \}
\]

Rule 21. All names are needed as input to the translation algorithm. Those are serviceName, interfaceName, chorOrchName.

Rule 22. All \((m, n \in \mathbb{N})\) inputs and outputs of the AD given as input \((in_1, in_2, \ldots, in_m)\) and SendEvent \(out_1, out_2, \ldots, out_n\) are converted as follows. The grounding of each input and output must be given.

\[
in\text{Concept}_x := \text{concept}(in_x) \text{ withGrounding } \text{concept(grounding(in_x))}
\]

\[
out\text{Concept}_y := \text{concept}(out_y) \text{ withGrounding } \text{concept(grounding(out_y))}
\]

\[
\forall x, y \in \mathbb{N}, \ 1 \leq x \leq m, \ 1 \leq y \leq n
\]

Rule 23. The transition rules section consists of rule sets for every GeneralAction, AcceptEvent, and every SendEvent of an AD.

\[
\text{allRules} := \bigcup_{a \in \text{GeneralAction}} \text{rules(a)} \cup \bigcup_{i \in \text{AcceptEvent}} \text{rules(i)} \cup \bigcup_{o \in \text{SendEvent}} \text{rules(o)}
\]

E.3.2 Concepts

State Signature.

Rule 24. For every OutputPin \(p\) of a complex flow, we define a new concept statement in the ontology. This updates the ontology initially defined in Rule 1. For later reference, the function state is defined as follows.

\[
\text{concept newVar}_x \text{ subConceptOf State}
\]

\[
\text{state}(p) := \text{newVar}_x
\]

Branch Variables.

Rule 25. For every outgoingEdge \(e_1, e_2, \ldots, e_n\) of each ForkNode, we create a new concept in the ontology. This updates the ontology initially defined in Rule 1. For later reference, the function branch is defined as follows.

\[
\text{concept newVar}_x \text{ subConceptOf Branch}
\]

\[
\text{branch}(e_y) := \text{newVar}_x
\]

E.3.3 Transition Rules

Skeleton.

We model the complex flows by defining a set of rules for every GeneralAction, AcceptEvent, and SendEvent in the AD. One part of such a rule’s updates creates instances for the concepts that relate to the ActionNode’s OutputPins. Its other parts and the rule’s conditions depend on the complex flows preceding the InputPins of the respective ActionNode. They will be defined by the actionFlow function in a later rule.
Rule 26. For every GeneralAction \( a \), we define a set of ASM rules as follows.

\[
\text{rules}(a) := \{ \text{if } c_1 \text{ and } c_2 \text{ and } \ldots \text{ and } c_m \\
\text{ then } u_1 \ u_2 \ldots \ u_m \\
\quad \text{add}(_\# \text{memberOf} \{ \text{concept}(p_1), \text{state}(p_1) \}) \\
\quad \text{add}(_\# \text{memberOf} \{ \text{concept}(p_2), \text{state}(p_2) \}) \\
\quad \ldots \\
\quad \text{add}(_\# \text{memberOf} \{ \text{concept}(p_n), \text{state}(p_n) \}) \} \text{ endif} : \\
(\langle c_1, u_1 \rangle, \langle c_2, u_2 \rangle, \ldots, \langle c_m, u_m \rangle) \in \text{actionFlow}(a), \\
p_x = \text{outputPins}_x(a), \ x \in \mathbb{N}, \ 1 \leq x \leq n \}
\]

Rule 27. For every AcceptEvent \( i \), we define its translation as follows.

\[
\text{rules}(i) := \{ \text{if } ?\text{var}_1 \text{memberOf} \text{concept}(i) \\
\quad \text{and not } ?\text{var}_1 \text{memberOf} \text{InternalComm} \\
\quad \text{and } c_1 \text{ and } c_2 \text{ and } \ldots \text{ and } c_m \\
\quad \text{then } u_1 \ u_2 \ldots \ u_m \\
\quad \text{add}(_\# \text{memberOf} \{ \text{concept}(p_1), \text{state}(p_1) \}) \\
\quad \text{add}(_\# \text{memberOf} \{ \text{concept}(p_2), \text{state}(p_2) \}) \\
\quad \ldots \\
\quad \text{add}(_\# \text{memberOf} \{ \text{concept}(p_n), \text{state}(p_n) \}) \} \text{ endif} : \\
(\langle c_1, u_1 \rangle, \langle c_2, u_2 \rangle, \ldots, \langle c_m, u_m \rangle) \in \text{actionFlow}(i), \\
p_x = \text{outputPins}_x(i), \ x \in \mathbb{N}, \ 1 \leq x \leq n \}
\]

Rule 28. For every SendEvent \( o \), we define its translation as follows.

\[
\text{rules}(o) := \{ \text{if } c_1 \text{ and } c_2 \text{ and } \ldots \text{ and } c_m \\
\text{ then } u_1 \ u_2 \ldots \ u_m \\
\quad \text{add}(_\# \text{memberOf} \text{concept}(o)) \\
\quad \text{add}(_\# \text{memberOf} \{ \text{concept}(p_1), \text{state}(p_1) \}) \\
\quad \text{add}(_\# \text{memberOf} \{ \text{concept}(p_2), \text{state}(p_2) \}) \\
\quad \ldots \\
\quad \text{add}(_\# \text{memberOf} \{ \text{concept}(p_n), \text{state}(p_n) \}) \} \text{ endif} : \\
(\langle c_1, u_1 \rangle, \langle c_2, u_2 \rangle, \ldots, \langle c_m, u_m \rangle) \in \text{actionFlow}(o), \\
p_x = \text{outputPins}_x(o), \ x \in \mathbb{N}, \ 1 \leq x \leq n \}
\]

The actionFlow function determines all possible combinations of conditions and corresponding updates for a GeneralAction, AcceptEvent, and SendEvent that result from the combination of execution paths that might arise from all its InputPins \( p_1, p_2, \ldots, p_n \).

Rule 29. We define the actionFlow function for every GeneralAction, AcceptEvent, and every SendEvent \( a \) as follows. The cross operator “\( \times \)” of some sets builds the cross
product of their members. The function actionFlow thus contains a set of n-tuples. $\epsilon$ is some undetermined Element.

$$\text{actionFlow}(a) := \text{objectFlow}(p_1, \{ \langle \epsilon, p_1 \rangle \}, \emptyset, \emptyset, \emptyset) \times \text{objectFlow}(p_2, \{ \langle \epsilon, p_2 \rangle \}, \emptyset, \emptyset, \emptyset) \times \ldots \times \text{objectFlow}(p_n, \{ \langle \epsilon, p_n \rangle \}, \emptyset, \emptyset, \emptyset)$$

where $p_x = \text{inputPins}_x(a), \quad x \in \mathbb{N}, \quad 1 \leq x \leq n$

Complex Flow.

For building the rules concerning one InputPin, we use the data structure objectFlow whose interpretation yields a set. Each element objectFlow$_x$ of the set represents one possible condition under which tokens can arrive in the target InputPin with number $x$. Different elements of the set objectFlow$_x$ will in the end result in different ASM rules (see Rule 29). These rules will be created from the contents of the components $t$, Cond, $C_{\text{out}}$ and $C_{\text{del}}$. The component $t$ carries the InputPin which triggered the rule generation. Cond contains additional conditions for the current rule to fire. The components $C_{\text{out}}$ and $C_{\text{del}}$ comprise concepts for which instances need to be created and deleted in the current rule’s updates.

The construction of the set objectFlow$_x$ will be defined iteratively. Its component $E_{\text{next}}$ contains a set of tuples of Elements $(\langle e_1, e_2 \rangle)$. Therein, the Element $e_2$ needs to be expanded in a following step. The Element $e_1$ will be the successor of $e_2$ in the AD. In the beginning, $E_{\text{next}}$ will only consist of the target InputPin $x$ (Rule 29). If $E_{\text{next}}$ only contains OutputPins, the construction of objectFlow$_x$ will be complete (Rule 35).

If $E_{\text{next}}$ contains its first value InputPin $x$, we just copy its preceeding ControlNode to $E_{\text{next}}$. Please note that, by the way we construct the normal form in Sect. E.2.3, there will be only one predecessor of every InputPin.

Rule 30. We define the objectFlow function for every InputPin $p$ as follows.

$$\text{objectFlow}(t, E_{\text{next}} \cup \{ \langle e_{\text{last}}, p \rangle \}, \text{Cond}, \text{C}_{\text{out}}, \text{C}_{\text{del}}) := \text{objectFlow}(t, E_{\text{next}} \cup \{ \langle p, \text{pre}(p) \rangle \}, \text{Cond} \cup \text{asm}(\text{cond}), \text{C}_{\text{out}}, \text{C}_{\text{del}})$$

When constructing the ASM rule of a token flow path at the spot of a DecisionNode $d$, we just follow the path by assigning $d$ to $e_{\text{last}}$ and extending $E_{\text{next}}$ by $d$’s predecessor. We add the ASM interpretation of the condition of the current edge to Cond. Please note, that for exploring one token flow path, we only explore one branch of the DecisionNode. The multiple rules generated for each DecisionNode as connotated in Sect. 4.1.2 result from the multiple application of Rule 31 to the different token paths initiated by different InputPins.

Rule 31. We define the objectFlow function for every DecisionNode $d$ as follows.

$$\text{objectFlow}(t, E_{\text{next}} \cup \{ \langle e_{\text{last}}, d \rangle \}, \text{Cond}, \text{C}_{\text{out}}, \text{C}_{\text{del}}) := \text{objectFlow}(t, E_{\text{next}} \cup \{ \langle d, \text{pre}(d) \rangle \}, \text{Cond} \cup \text{asm}(\text{cond}), \text{C}_{\text{out}}, \text{C}_{\text{del}})$$

where $\text{cond} = \text{guard}(\text{edge}) : \text{edge} \in \text{ActivityEdge}$,

$$d = \text{isInputOf}(\text{edge}), \quad e_{\text{last}} = \text{isOutputOf}(\text{edge})$$
For a MergeNode consisting of \( n \) incoming branches, we construct \( n \) rules which each represents one of the branches.

**Rule 32.** We define the objectFlow function for every MergeNode \( m \) as follows.

\[
\text{objectFlow}(t, E_{\text{next}} \cup \{ \langle e_{\text{last}}, m \rangle \}, \text{Cond}, C_{\text{out}}, C_{\text{del}}) := \text{objectFlow}(t, E_{\text{next}} \cup \{ \langle m, \text{pre}_1(m) \rangle \}, \text{Cond}, C_{\text{out}}, C_{\text{del}}) \\
\cup \text{objectFlow}(t, E_{\text{next}} \cup \{ \langle m, \text{pre}_2(m) \rangle \}, \text{Cond}, C_{\text{out}}, C_{\text{del}}) \\
\cup \ldots \\
\cup \text{objectFlow}(t, E_{\text{next}} \cup \{ \langle m, \text{pre}_n(m) \rangle \}, \text{Cond}, C_{\text{out}}, C_{\text{del}})
\]

“Tokens arriving at a fork are duplicated across the outgoing edges. If at least one outgoing edge accepts the token, duplicates of the token are made and one copy traverses each edge that accepts the token. The outgoing edges that did not accept the token due to failure of their targets to accept it, keep their copy in an implicit FIFO queue until it can be accepted by the target.” [11, p. 363] We follow this specification in that we continue expanding rules for the branch of \( e_{\text{last}} \) reached by the computation and create pending tokens for all other branches leaving the ForkNode (2nd line). In consequence, rules generated for InputPins different from \( t \) will create branch variables for \( e_{\text{last}} \). Therefore, we now add rules to consume these instances (3rd line).²

**Rule 33.** We define the objectFlow function for every ForkNode \( f \) as follows.

\[
\text{objectFlow}(t, E_{\text{next}} \cup \{ \langle e_{\text{last}}, f \rangle \}, \text{Cond}, C_{\text{out}}, C_{\text{del}}) := \text{objectFlow}(t, E_{\text{next}} \cup \{ \langle f, \text{pre}(f) \rangle \}, \text{Cond}, C_{\text{out}} \cup \text{branches}, C_{\text{del}}) \\
\cup \text{objectFlow}(t, E_{\text{next}}, \text{Cond}, C_{\text{out}}, C_{\text{del}} \cup \{ \text{branch}(\text{edge}_i) \})
\]

where \( \text{edge}_x \in \text{ActivityEdge}, \ f = \text{isInputOf}(\text{edge}_x), \ \forall x \in \{1..n\} \)

\( e_{\text{last}} = \text{isOutputOf}(\text{edge}_i) \)

\( \text{branches} = \{ \text{branch}(\text{edge}_y) : y \in \{1..n..\} \} \)

For a JoinNode \( j \), we track all its incomingEdges in the same rule by adding the Predecessors of \( j \) to \( E_{\text{next}} \).

**Rule 34.** We define the objectFlow function for every JoinNode \( j \) as follows.

\[
\text{objectFlow}(t, E_{\text{next}} \cup \{ \langle e_{\text{last}}, j \rangle \}, \text{Cond}, C_{\text{out}}, C_{\text{del}}) := \text{objectFlow}(t, E_{\text{next}} \cup \text{Predecessors}, \text{Cond}, C_{\text{out}}, C_{\text{del}})
\]

where \( \text{Predecessors} = \{ \langle j, p \rangle : p \in \text{Pre}(j) \} \)

**Complex Flow Termination.**

The computation terminates when \( E_{\text{next}} \) only contains OutputPins. In this case, the appearance of one objectFlow function corresponds to exactly one execution path a

²Please note that instead of adopting the FIFO semantics described above, we store all incoming tokens as instances of a concept branch. We thus deviate from the definition in [11] using a “set semantics”, here.
token may take reaching the InputPin \( t \). Along this path, tokens would leave the OutputPins \( p_1, p_2, \ldots, p_n \), the conditions \( \text{Cond} \) would have to hold, branch variable instances for \( C_{\text{out}} \) were created, branch variable instances for each member of \( C_{\text{del}} \) would be removed, and the InputPin \( t \) would finally receive a fresh token. The statement that “tokens would leave the OutputPins \( p_1, p_2, \ldots, p_n \)” is reflected by adding conditions requiring instances for the corresponding concepts, and by removing these original tokens from the OutputPins \( p_1, p_2, \ldots, p_n \) in the updates \( \text{upd} \) of Rule 35.

**Rule 35.** We define the objectFlow function for every set \( E_{\text{next}} \) containing only OutputPins as follows.

\[
\text{objectFlow}(t, E_{\text{next}}, \text{Cond}, C_{\text{out}}, C_{\text{del}}) := \{ (\text{cond}, \text{upd}) \}
\]

where \( E_{\text{next}} = \{ (\_\_\_, p_1), (\_\_\_, p_2), \ldots, (\_\_\_, p_n) \} \), \( p_x \in \text{OutputPin}, \ 1 \leq x \leq n \)

\( \text{Cond} = \{ c_1, c_2, \ldots, c_k \} \)

\( \text{cond} = \text{?} \text{var}_1 \text{memberOf} \{ \text{concept}(t), \text{state}(p_1) \} \)

\( \text{and} \text{?} \text{var}_2 \text{memberOf} \{ \text{concept}(t), \text{state}(p_2) \} \)

\( \text{and} \ldots \)

\( \text{and} \text{?} \text{var}_n \text{memberOf} \{ \text{concept}(t), \text{state}(p_n) \} \)

\( \text{and} \text{disjoint} (\{ \text{?} \text{var}_1, \text{?} \text{var}_2, \ldots, \text{?} \text{var}_n \}) \)

\( \text{and} c_1 \text{and} c_2 \text{and} \ldots \text{and} c_k \)

\( C_{\text{out}} = \{ \text{Out}_1, \text{Out}_2, \ldots, \text{Out}_g \} \)

\( C_{\text{del}} = \{ \text{Del}_1, \text{Del}_2, \ldots, \text{Del}_h \} \)

\( \text{upd} = \text{create}(\_\_\# \text{memberOf} \text{Out}_1) \)

\( \text{create}(\_\_\# \text{memberOf} \text{Out}_2) \ldots \text{create}(\_\_\# \text{memberOf} \text{Out}_g) \)

\( \text{delete}(\text{?} \text{var}_1 \text{memberOf} \_\#) \)

\( \text{delete}(\text{?} \text{var}_2 \text{memberOf} \_\#) \ldots \text{delete}(\text{?} \text{var}_n \text{memberOf} \_\#) \)

\( \text{delete}(\_\# \text{memberOf} \text{Del}_1) \)

\( \text{delete}(\_\# \text{memberOf} \text{Del}_2) \ldots \text{delete}(\_\# \text{memberOf} \text{Del}_h) \)

**Rule 36.** We define the function disjoint for a set of variables as follows.

\[
\text{disjoint}(\{ \text{?} \text{var}_1, \text{?} \text{var}_2, \ldots, \text{?} \text{var}_l \}) = \text{?} \text{var}_1 \neq \text{?} \text{var}_2 \text{and} \text{?} \text{var}_1 \neq \text{?} \text{var}_3
\]

\( \text{and} \ldots \text{and} \text{?} \text{var}_1 \neq \text{?} \text{var}_l \)

\( \text{and} \text{?} \text{var}_2 \neq \text{?} \text{var}_3 \text{and} \text{?} \text{var}_2 \neq \text{?} \text{var}_4 \)

\( \text{and} \ldots \text{and} \text{?} \text{var}_2 \neq \text{?} \text{var}_l \)

\( \text{and} \ldots \)

\( \text{and} \text{?} \text{var}_{l-1} \neq \text{?} \text{var}_l \)
F Translation between Cashew and Activity Diagrams Subset

This section presents in pseudo-code an algorithm that implements the transformation between Cashew workflows and UML Activity Diagrams as described in Section 4.1. The transformation algorithm relies on a simple data-structure for storing intermediate transformation results. This data-structure stores the complete transformation result as an interconnected network of objects where the initial node for the activity diagram is referred to as initialNode, and the outgoing edge as outgoingEdge.

Init Result
Switch Cashew-Workflow

Case Sequential:
    Result = Transform_Sequential(Cashew-Workflow)

Case Concurrent:
    Result = Transform_Concurrent(Cashew-Workflow)

Case Interleaved:
    Result = Transform_Interleaved(Cashew-Workflow)

Case Xor:
    Result = Transform_Xor(Cashew-Workflow)

Case While:
    Result = Transform_While(Cashew-Workflow)

Case Until:
    Result = Transform_Until(Cashew-Workflow)

Case DeferredChoice:
    Result = Transform_DeferredChoice(Cashew-Workflow)

Case DeferredWhile:
    Result = Transform_DeferredWhile(Cashew-Workflow)

Case DeferredUntil:
    Result = Transform_DeferredUntil(Cashew-Workflow)

The following sections describe the transformation for each of these workflow patterns. The pseudo-code makes use of the keyword new to indicate the creation of a new object like in traditional object-oriented programming languages. Additionally it refers to the construct Append that operates over two Cashew-Workflow instances and returns a new Cashew-Workflow which corresponds to the concatenation of the first Cashew-Workflow and the second one, i.e. the outgoing edge of the first is connected to the initial edge of the second. The result of the transformation is an interconnected network of objects that represents the Cashew Workflow as a UML Activity Diagram.
F.1 Transform Sequential

Result = new Sequence

For each Performance
    TransResult = Translate (Performance)
    If Result is empty
        Result.initialNode = TransResult.initialNode
        Result = Append(Result, TransResult)
        Result.outgoingEdge = TransResult.outgoingEdge

F.2 Transform Concurrent

Result = new Concurrent
Result.initialNode = new ForkNode
join = new JoinNode

For each Performance
    TransResult = Translate(Performance)
    Append(Result.initialNode, TransResult)
    Append(TransResult, join)

Result.outgoingEdge = join.outgoingEdge

F.3 Transform XOR

Result = new XOR
Result.initialNode = new DecisionNode

TransIf = Translate(XOR.IfBranch)
TransElse = Translate(XOR.ElseBranch)

DecisionNode.createBranch(XOR.IfCondition, TransIf)
DecisionNode.createElseBranch(TransIf)

merge = new MergeNode
Result = Append(Result, merge)
Result.outgoingEdge = mergeNode.outgoingEdge

F.4 Transform While

Result = new While
Result.initialNode = new DecisionNode

TransWhile = Translate(While.IfBranch)
DecisionNode.createBranch(While.IfCondition, TransWhile)

Result.outgoingEdge = DecisionNode.elseBranch

**F.5 Transform Until**

Result new Until

TransIf = Translate(Until.IfBranch)
Result.initialNode = TransIf.initialNode

decision = new DecisionNode

DecisionNode.createBranch(Until.IfCondition, Result.initialNode)
Result.outgoingEdge = DecisionNode.ElseBranch

**F.6 Transform Interleave**

Result = new Interleave
send = new Send(Interleave.getSignal)
fork = new ForkNode
Result.initialNode = ForkNode
Append(send, fork)

merge = new MergeNode

Foreach branch in Interleave
    transBranch = Transform(branch)
    transReceive = new Receive(Interleave.getSignal)
    Append(fork, receive)
    Append(receive, transBranch)
    transSend = new Send(Interleave.getSignal)
    Append(transBranch, transSend)
    Append(transSend, merge)

Result.outgoingEdge = merge.outgoingEdge

**F.7 Transform DeferredChoice**

Result new DeferredChoice
Result.initialNode = new ForkNode
activityGroup = new ActivityGroup
merge = new MergeNode

Foreach branch
    ActivityGroup.add(branch.AtomicPerform)
Append(Result.initialNode, branch.AtomicPerform)
TransEnabled = Transform(branch.enabledFragment)
Create InterruptingEdge between AtomicActivity and EnabledFragment
Append(merge, TransEnabled)

Result.outgoingEdge = merge.outgoingEdge

F.8 Transform DeferredUntil

Result = new DeferredUntil
fork = new ForkNode
activityGroup = new ActivityGroup
activityGroup.add(enabledTrans)

untilBranch = DeferredUntil.untilBranch
enabledTrans = Transform(untilBranch)

Append(fork, enabledTrans)
Create InterruptingEdge between enabledTrans and fork

atomicPerform = DeferredUntil.restBranch
activityGroup.add(atomicPerform)
Result.initialNode = atomicPerform
Append(fork, atomicPerform)
Result.outgoingEdge = atomicPerform.outgoingEdge

F.9 Transform DeferredWhile

Result = new DeferredWhile
fork = ForkNode
merge = new MergeNode
Result.initialNode = merge

activityGroup = new ActivityGroup

whileBranch = DeferredWhile.whileBranch
atomicPerform = whileBranch.atomicPerform
enabledTrans = Transform(enabledFragment)
activityGroup.add(enabledTrans)

Append(fork, atomicPerform)
Create InterruptingEdge between enabledTrans and merge
activityGroup.add(atomicPerform)

Append(fork, enabledTrans)
Create InterruptingEdge between enabledTrans and merge
Result.outgoingEdge = atomicPerform.outgoingEdge