Semantic Web Services Framework (SWSF)

Version 1.0

This is the initial technical report of the Semantic Web Services Language (SWSL) Committee of the Semantic Web Services Initiative (SWSI). This report consists of the following four top-level documents, with four related appendices.

We welcome feedback and discussion from interested parties on the **public-sws-ig@w3.org** email list.

- Archives
- Subscription info

Semantic Web Services Framework (SWSF) Overview

Version 1.0

Authors:

Steve Battle (Hewlett Packard) Abraham Bernstein (University of Zurich) Harold Boley (National Research Council of Canada) Benjamin Grosof (Massachusetts Institute of Technology) Michael Gruninger (NIST) Richard Hull (Bell Labs Research, Lucent Technologies) Michael Kifer (State University of New York at Stony Brook) David Martin (SRI International) Sheila McIlraith (University of Toronto) Deborah McGuinness (Stanford University) Jianwen Su (University of California, Santa Barbara) Said Tabet (The RuleML Initiative)

Abstract

This document provides an overview of the Semantic Web Services Framework (SWSF), which includes the Semantic Web Services Language (SWSL) and the Semantic Web Services Ontology (SWSO). This is one of four documents that make up the initial report of the Semantic Web Services Language Committee of the Semantic Web Services Initiative.

Status of this document

History of publication at http://www.daml.org/services/swsl/report/overview/:

- v. 0.9: April 6, 2005
- v. 0.91: April 13, 2005
- v. 0.92: April 25, 2005
- v. 0.93: May 5, 2005

History of publication at http://www.daml.org/services/swsf/overview/:

- v. 1.0: May 9, 2005

Table of contents

1 Introduction to SWSL and SWSO

The promise of Web services and the need for widely accepted standards enabling them are widely recognized, and considerable efforts are underway to define and evolve such standards in the commercial realm. In particular, the Web Services Description Language (WSDL) [*WSDL 1.1*] is already well established as an essential building block in the evolving stack of Web service technologies, and is being standardized in the W3C's Web Services Description Working Group. WSDL, in essence, allows for the specification of the syntax of the input and output messages of a basic service, as well as other details needed for the invocation of the service. WSDL does not, however, support the specification of workflows composed of basic services. In this area, the Business Process Execution Language for Web Services (BPEL4WS) [*BPEL 1.1*], under development at OASIS, has the most prominent status. The Choreography Description Language under development by W3C's Web Services Choreography Working Group, serves to "define from a global viewpoint ... the information exchanges that occur and the jointly agreed ordering rules that need to be satisfied" in carrying out a Web service-based transaction [*WS-Choreography*]. With respect to registering Web services for purposes of advertising and discovery, Universal Description, Discovery and Integration (UDDI) [*UDDI v3.02*] has received the most attention to date. Standards are also being developed in connection with various other aspects of Web service provisioning, such as reliable messaging, security, and resource management.

At the same time, recognition is growing of the need for richer semantic specifications of Web services, based on a compressive representational framework that spans the full range of service-related concepts. Such a framework will enable fuller, more flexible automation of service provision and use, support the construction of more powerful tools and methodologies, and promote the use of semantically well-founded reasoning about services. Because an expressive representation framework permits the specification of many different aspects of services, it can provide a foundation for a broad range of activities, across the Web service lifecycle. For example, richer semantics can support greater automation of service selection and invocation; automated translation of message content between heterogeneous interoperating services; automated or semi-automated approaches to service composition; more comprehensive approaches to service monitoring and recovery from failure; and fuller automation of verification, simulation, configuration, supply chain management, contracting, and negotiation for services. The technologies presented in this document are designed to realize this *Semantic Web Service* vision [*McIlraith01*].

This report presents two major components:

- The **Semantic Web Services Language (SWSL)** is used to specify formal characterizations of Web service concepts and descriptions of individual services. It includes two sublanguages. ***SWSL-FOL*** is based on first-order logic (FOL) and is used primarily to express the formal characterization (ontology) of Web service concepts. ***SWSL-Rules*** is based on the logic-programming (or "rules") paradigm and is used to support the use of the service ontology in reasoning and execution environments based on that paradigm. SWSL is a general-purpose language (that is, its features are not service-specific), but it has been designed to address the needs of Semantic Web Services. Also associated with SWSL is a *simplified presentation syntax*, which reduces to SWSL-FOL.
- The **Semantic Web Services Ontology (SWSO)** presents a conceptual model by which Web services can be described, and an axiomatization, or formal characterization, of that model. The complete axiomatization is given in first-order logic, using SWSL-FOL, with a model-theoretic semantics that specifies the precise *meaning* of the concepts. We call this FOL form of the ontology ***FLOWS*** -- **F**irst-Order **L**ogic **O**ntology for **W**eb **S**ervices. In addition, the axioms from FLOWS have been systematically translated into the SWSL-Rules language (with an unavoidable weakening of some axioms). The resulting ontology, which relies on logic-programming semantics, is called ***ROWS*** -- **R**ules **O**ntology for **W**eb **S**ervices.

More specifically, SWSL is a general-purpose logical language, with certain features to make it usable with the basic languages and infrastructure of the Web. These features include URIs, integration of XML built-in types, and XML-compatible namespace and import mechanisms. SWSL includes two layers of expressiveness: SWSL-FOL and SWSL-Rules. SWSL-FOL is a first-order logic, extended with features from HiLog [*Chen93*] and the frame syntax of F-logic [*Kifer95*]. SWSL-Rules is a full-featured logic programming (LP) language, which includes a novel combination of features from Courteous logic programs [*Grosof99a*], HiLog, and F-logic.

Nearly all the elements of the syntax are common to both SWSL-FOL and SWSL-Rules, so as to promote the ability of developers to easily work with both layers, and to facilitate various kinds of interchange and interoperation between the layers. The presentation syntax is designed for readability, and incorporates a number of convenience features, such as an object-oriented style of presentation, which can be used to improve code organization and comprehensibility, but without changing the expressiveness and tractability of the underlying logical systems. An XML-based serialization syntax, based on RuleML, is also specified.

FLOWS is an axiomatized ontology of service concepts, which provides the conceptual framework for describing and reasoning about services. FLOWS draws many of its intuitions and lessons-learned from OWL-S, the OWL ontology for Web services [*OWL-S 1.1*]. A key contribution of the FLOWS ontology is the development of a rich behavioural process model, based on ISO 18629 Process Specification Language (PSL) [*Gruninger03a*], [*Gruninger03b*]. Originally designed to support interoperability among process modeling languages, PSL provides the ideal foundation for interoperability among emerging Web service process models, while supporting the realization of automation task associated with the Semantic Web Service vision. FLOWS goes beyond PSL in modeling many Web-service specific process concepts including messages, channels, inputs and outputs.

FLOWS is designed modularly. It comprises an ontology for service descriptors, somewhat akin to a domain-independent yellow-pages or OWL-S service profile, an extensive process model ontology, and a grounding that relates the process model message types to WSDL messages. The process model ontology is in turn comprised of a core ontology and a number of extensions.

The statements above about FLOWS are also true of ROWS, which is derived from FLOWS.

In addition to presenting SWSL and SWSO, this document also provides some guidance as to how SWSL-FOL and SWSL-Rules can be used together, examples of applications of SWSL and SWSO, and a discussion of how FLOWS-based service descriptions can be *grounded* in the concrete descriptions of messages and protocols provided by WSDL.

This proposal has been prepared by the Semantic Web Services Language Committee of the Semantic Web Services Initiative, a collaborative international research effort. In addition to providing further evolution of the SWSL language and ontology, SWSI will also be a forum for working towards convergence of SWSL with the products of the WSMO research effort [*Bruijn05*].

The overall design of the SWSL Language and Ontology, and of the application scenarios using those, has been motivated by an overall longer-term vision and set of objectives for Semantic Web services. These are summarized in a requirements document, which is available on the SWSL Committee's Web site [*SWSL Requirements*]. That document expressed our thinking as we set out to design the SWSF Languge and Ontology. The current design of SWSF addresses, directly or indirectly, most (but not all) of the points in the requirements document.

Related Efforts

OWL-S: OWL-S [*OWL-S 1.1*] is an ontology of service concepts expressed in OWL-DL, a decidable description logic language. In contrast, FLOWS is an ontology of service concepts expressed in first-order logic. Since OWL-DL trades off expressiveness for decidability, there were aspects of the OWL-S process model ontology whose semantics could not be defined in the OWL-DL language. This is not a problem with the FLOWS ontology. FLOWS provides a more comprehensive ontology, building on the conceptual model and lessons-learned from OWL-S. As such it captures all concepts in OWL-S. In terms of coverage it is distinguished by its axiomatization of messages, something that was not addressed in OWL-S. An important final distinction between OWL-S and FLOWS is with respect to the role it plays. Whereas both endeavours attempt to provide an ontology for Web services, FLOWS had the additional objective of acting as a focal point for interoperability, enabling other business process modeling languages to be expressed or related to FLOWS.

Process Specification Language (PSL) and related specifications: There are a number of efforts whose aim is to develop a process specification ontology. These efforts include PSL [*Gruninger03a*], BPMI [*BPML 1.0*], and others. The process ontology of SWSL is based on PSL. From the perspective of PSL, FLOWS may be viewed as a collection of extensions that situate the description of processes within a larger context of message-based communications across networks.

BPEL4WS: There is much to be said about the relationship between this effort and BPEL4WS [*BPEL 1.1*]. BPEL4WS provides an executable business process modeling language that enables the specification (orchestration) of executable business processes as well as the description of non-executable processes. The most obvious point of comparison between BPEL4WS and this effort is with respect to the FLOWS ontology. The FLOWS ontology is a great deal more than process modeling, supporting the axiomatization of non-functional service descriptions (for service discovery) as well as the rich mapping of message types to WSDL. We believe that the FLOWS process model subsumes the BPEL4WS process model, enabling the encoding of both executable and abstract BPEL process models in FLOWS. (Note that we have not yet confirmed this through a systematic translation.) Whereas BPEL takes a message-centric view of services, FLOWS provides for a process-centric view of services, where message exchanges are conceived as additional processes that may be inferred automatically. A notable difference between the two formalisms is that FLOWS supports the encoding of service side-effects, i.e., the effects of services on the world, which enables automated composition of Web services, in additon to the manual composition supported by BPEL. Finally, BPEL is indeed designed for Web service orchestration, whereas FLOWS may also be used for choreography of services.

Web Services Modeling Ontology (WSMO): WSMO [*Bruijn05*] is a parallel effort to define an ontology and a language for Semantic Web services. Like SWSL-Rules, WSMO's rule language, WSML is largely based on F-logic and these languages share much of the logical expression syntax. Nevertheless, the two groups have pursued complimentary goals. WSMO has focused heavily on the language effort, and in particular on end user issues, associated with the language. In particular, they have developed a "conceptual syntax" for top-level descriptions of services, which might make the specifications easier to read for the end user. WSMO has also paid special attention to the issue of OWL compatibility. To this end, it defined WSML-Core as a subset of both OWL and WSML, which serves as a common ground for ontology interoperability. In contrast, SWSL's focus was on extending the functionality of the rule language. In particular, SWSL-Rules supports meta-reasoning with its HiLog and reification extensions. It also supports prioritized defaults and classical negation by incorporating Courteous Logic Programming.

A major distinction between the WSMO effort and the effort presented here is with respect to the ontology. The two efforts are divergent, but complementary. WSMO has focussed on describing Web service choreography through guarded transition rules, Event-Condition-Action (ECA) rules which are viewed as abstract state machines. In contrast, FLOWS provides an extensive first-order process ontology, which enables description of process orchestration as well as message exchange among processes. The FLOWS ontology also provides the foundation for enabling automated simulation, verification and composition of Web services, something that cannot be done with the guarded transition rules, without further definition of the semantics.

---

2 Document Roadmap

This report comprises four top-level documents:

1. **Semantic Web Services Framework Overview**, this document, includes introductory material, comments about selected other work that is most directly related to this work, and acknowledgements.
2. The Semantic Web Services Language (SWSL) describes the syntactic elements of SWSL and, informally, its semantic underpinnings. It also explains how SWSL-FOL and SWSL-Rules can be used together, and presents an XML serialization syntax for SWSL, based on RuleML.
3. The Semantic Web Services Ontology (SWSO) presents FLOWS, a first-order ontology for Web services, expressed in SWSL-FOL, and its partial translation into ROWS, expressed in SWSL-Rules. SWSO also includes material about grounding FLOWS/ROWS process models with WSDL.

The SWSO document includes four appendices:

A PSL in SWSL-FOL and SWSL-Rules B Axiomatization of the FLOWS Process Model C Axiomatization of the Process Model in SWSL-Rules D Reference Grammars

4. SWSF Application Scenarios gives examples that illustrate possible uses of the ontology and the language.

So as to be more-or-less self-contained, each of these four documents includes a glossary and references. For simplicity, these are the same in each document.

For pedagogical reasons, it makes sense to introduce SWSL before SWSO, since SWSL is used in expressing SWSO. However, the SWSO document relies primarily on simple uses of SWSL-FOL. Hence, those readers who are already familiar with first-order logic may want to begin with the SWSO document, which contains the Web services-specific aspects of this work. A suggested order of reading for such readers is: the Overview document (this document); the Ontology document; the first two sections of the Applications document; the Language document; the three remaining sections of the Applications document.

3 Acknowledgements

We are grateful for the vision and support of **Murray Burke** and **Dieter Fensel**, who first envisioned the collaboration that became SWSI, and were essential in getting it started. **Katia Sycara**, in her role as SWSI co-executive (with Dieter) has been active and supportive throughout the efforts of the committee, and has given valuable guidance in many areas.

The participation of some members of SWSL was funded by the DARPA DAML Program. We very much appreciate the ongoing support, interest, and patience of DAML program managers **Jim Hendler, Murray Burke**, and **Mark Greaves**.

We are also grateful to **Carine Bournez** and **Sandro Hawke**, of the World Wide Web Consortium, for guidance in the preparation of this submission. We thank the W3C for permitting and supporting the use of the public-sws-ig mailing list for discussion of SWSL issues, and Carine Bournez for helping to make this possible.

We would like to give special thanks to the following individuals who have made significant contributions to this work:

- Karl Aberer
- Mark Burstein
- Bijan Parsia
- Daniela Berardi
- Richard Goodwin
- Drew McDermott
- Terry Payne
- Axel Polleres
- Austin Tate

We are also greatly appreciative of the efforts of the following individuals, who, as reviewers, provided valuable feedback on short notice:

- Tom Hite
- Ruben Lara
- Massimo Paolucci

This document has benefited from input from members of the Semantic Web Services Architecture (SWSA) committee.

4 Glossary

Activity
: *Activity*. In the formal PSL ontology, the notion of activity is a basic construct, which corresponds intuitively to a kind of (manufacturing or processing) activity. In PSL, an activity may have associated *occurrences*, which correspond intuitively to individual instances or executions of the activity. (We note that in PSL an activity is not a class or type with occurrences as members; rather, an activity is an object, and occurrences are related to this object by the binary predicate `occurrence_of`.) The occurrences of an activity may impact fluents (which provide an abstract representation of the "real world"). In FLOWS, with each service there is an associated activity (called the "service activity" of that service). The service activity may specify aspects of the internal process flow of the service, and also aspects of the messaging interface of that service to other services.

Channel
: *Channel*. In FLOWS, a channel is a formal conceptual object, which corresponds intuitively to a repository and conduit for messages. The FLOWS notion of channel is quite primitive, and under various restrictions can be used to model the form of channel or message-passing as found in web services standards, including WSDL, BPEL, WS-Choreography, WSMO, and also as found in several research investigations, including process algrebras.

FLOWS
: ***F**irst-order **L**ogic **O**ntology for **W**eb **S**ervices*. FLOWS, also known as SWSO-FOL, is the first-order logic version of the Semantic Web Services Ontology. FLOWS is an extension of the PSL-OuterCore ontology, to incorporate the fundamental aspects of (web and other electronic) services, including service descriptors, the service activity, and the service grounding.

Fluent
: *Fluent*. In FLOWS, following PSL and the situation calculii, a fluent is a first-order logic term or predicate whose value may vary over time. In a first-order model of a FLOWS theory, this being a model of PSL-OuterCore, time is represented as a discrete linear sequence of *timees*, and fluents has a value for each time in this sequence.

Grounding
: *Grounding*. The SWSO concepts for describing service activities, and the instantiations of these concepts that describe a particular service activity, are *abstract* specifications, in the sense that they do not specify the details of particular message formats, transport protocols, and network addresses by which a Web service is accessed. The role of the *grounding* is to provide these more concrete details. A substantial portion of the grounding can be acheived by mapping SWSO concepts into corresponding WSDL constructs. (Additional grounding, e.g., of some process-related aspects of SWSO, might be acheived using other standards, such as BPEL.)

Message
: *Message*. In FLOWS, a message is a formal conceptual object, which corresponds intuitively to a single message that is created by a service occurrence, and read by zero or more service occurrences. The FLOWS notion of message is quite primitive, and under various restrictions can be used to model the form of messages as found in web services standards, including WSDL (1.0 and 2.0), BPEL, WS-Choreography, WSMO, and also as found in several research investigations. A message has a *payload*, which corresponds intuitively to the body or contents of the message. In FLOWS emphasis is placed on the knowledge that is gained by a service occurrence when reading a message with a given payload (and the knowledge needed to create that message.

Occurrence
: *Occurence (of a service)*. In FLOWS, a service *S* has an associated FLOWS activity *A* (which generalizes the notion of PSL activity). An *occurrence* of *S* is formally a PSL occurrence of the activity *A*. Intuitively, this occurrence corresponds to an instance or execution (from start to finish) of the activity *A*, i.e., of the process associated with service *S*. As in PSL, an occurrence has a starting time time and an ending time.

PSL
: *Process Specification Language*. The Process Specification Language (PSL) is a formally axiomatized ontology [*Gruninger03a*, *Gruninger03b*] that has been standardized as ISO 18629. PSL provides a layered, extensible ontology for specifying properties of processes. The most basic PSL constructs are

embodied in PSL-Core; and PSL-OuterCore incorporates several extensions of PSL-Core that includes several useful constructs. (An overview of concepts in PSL that are relevant to FLOWS is given in Section 6 of the Semantic Web Services Ontology document.)

QName

Qualified name. A pair (*URI*, *local-name*). The *URI* represents a namespace and *local-name* represents a name used in an XML document, such as a tag name or an attribute name. In XML, QNames are syntactically represented as *prefix*:*local-name*, where *prefix* is a macro that expands into a concrete URI. See Namespaces in XML for more details.

ROWS

***R**ules **O**ntology for **W**eb **S**ervices*. ROWS, also known as SWSO-Rules, is the rules-based version of the Semantic Web Services Ontology. ROWS is created by a relatively straight-forward, almost faithful, transformation of FLOWS, the First-order Logic Ontology for Web Services. As with FLOWS, ROWS incorporates fundamental aspects of (web and other electronic) services, including service descriptors, the service activity, and the service grounding. ROWS enables a rules-based specification of a family of services, including both the underlying ontology and the domain-specific aspects.

Service

(Formal) Service. In FLOWS, a service is a conceptual object, that corresponds intuitively to a web service (or other electronically accessible service). Through binary predicates a service is associated with various service descriptors (a.k.a. non-functional properties) such as Service Name, Service Author, Service URL, etc.; an *activity* (in the sense of PSL) which specifies intuitively the process model associated with the service; and a *grounding*.

Service contract

Describes an agreement between the service requester and service provider, detailing requirements on a service occurrence or family of service occurrences.

Service descriptor

Service Descriptor. This is one of several non-functional properties associated with services. The Service Descriptors include Service Name, Service Author, Service Contract Information, Service Contributor, Service Description, Service URL, Service Identifier, Service Version, Service Release Date, Service Language, Service Trust, Service Subject, Service Reliability, and Service Cost.

Service offer description

Describes an abstract service (i.e. not a concrete instance of the service) provided by a service provider agent.

Service requirement description

Describes an abstract service required by a service requester agent, in the context of service discovery, service brokering, or negotiation.

sQName

Serialized QName. A serialized QName is a shorthand representation of a URI. It is a macro that expands into a full-blown URI. sQNames are *not* QNames: the former are URIs, while the latter are pairs (*URI*, *local-name*). Serialized QNames were originally introduced in RDF as a notation for shortening URI representation. Unfortunately, RDF introduced confusion by adopting the term QName for something that is different from QNames used in XML. To add to the confusion, RDF uses the syntax for sQNames that is identical to XML's syntax for QNames. SWSL distinguishes between QNames and sQNames, and uses the syntax *prefix#local-name* for the latter. Such an sQName expands into a full URI by concatenating the value of *prefix* with *local-name*.

URI

Universal Resource Identifier. A symbol used to locate resources on the Web. URIs are defined by IETF. See Uniform Resource Identifiers (URI): Generic Syntax for more details. Within the IETF standards the notion of URI is an extension and refinement of the notions of Uniform Resource Locator (URL) and Relative Uniform Resource Locators.

5 References

[Berardi03]

Automatic composition of e-services that export their behavior. D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. In *Proc. 1st Int. Conf. on Service Oriented Computing (ICSOC)*, volume 2910 of *LNCS*, pages 43--58, 2003.

[Bernstein2000]

How can cooperative work tools support dynamic group processes? Bridging the specificity frontier. A. Bernstein. In *Proc. Computer Supported Cooperative Work (CSCW'2000)*, 2000.

[Bernstein2002]
Towards High-Precision Service Retrieval. A. Bernstein, and M. Klein. In *Proc. of the first International Semantic Web Conference (ISWC'2002)*, 2002.

[Bernstein2003]
Beyond Monotonic Inheritance: Towards Semantic Web Process Ontologies. A. Bernstein and B.N. Grosof (NB: authorship sequence is alphabetic). Working Paper, Aug. 2003. Available at: http://ebusiness.mit.edu/bgrosof/#beyond-mon-inh-basic.

[Bonner93]
Database Programming in Transaction Logic. A.J. Bonner, M. Kifer, M. Consens. *Proceedings of the 4-th Intl.~Workshop on Database Programming Languages*, C. Beeri, A. Ohori and D.E. Shasha (eds.), 1993. In Springer-Verlag Workshops in Computing Series, Feb. 1994: 309-337.

[Bonner98]
A Logic for Programming Database Transactions. A.J. Bonner, M. Kifer. Logics for Databases and Information Systems, J. Chomicki and G. Saake (eds.). Kluwer Academic Publishers, 1998: 117-166.

[Bruijn05]
Web Service Modeling Ontology (WSMO). J. de Bruijn, C. Bussler, J. Domingue, D. Fensel, M. Hepp, M. Kifer, B. König-Ries, J. Kopecky, R. Lara, E. Oren, A. Polleres, J. Scicluna, M. Stollberg. *DERI Technical Report.*

[BPML 1.0]
A. Arkin. Business Process Modeling Language. BPMI.org, 2002

[BPEL 1.1]
Business Process Execution Language for Web Services, Version 1.1 . S. Thatte, editor. OASIS Standards Specification, May 5, 2003.

[Bultan03]
Conversation specification: A new approach to design and analysis of e-service composition. T. Bultan, X. Fu, R. Hull, and J. Su. In *Proc. Int. World Wide Web Conf. (WWW)*, May 2003.

[Chang73]
Symbolic Logic and Mechanical Theorem Proving. C.L. Chang and R.C.T. Lee. Academic Press, 1973.

[Chen93]
HiLog: A Foundation for Higher-Order Logic Programming. W. Chen, M. Kifer, D.S. Warren. Journal of Logic Programming, 15:3, February 1993, 187-230.

[Chimenti89]
Towards an Open Architecture for LDL. D. Chimeti, R. Gamboa, R. Krishnamurthy, VLDB Conference, 1989: 195-203.

[deGiacomo00]
ConGolog, A Concurrent Programming Language Based on the Situation Calculus. G. de Giacomo, Y. Lesperance, and H. Levesque. Artificial Intelligence, 121(1--2):109--169, 2000.

[Fu04]
WSAT: A Tool for Formal Analysis of Web Services. X. Fu, T. Bultan, and J. Su. *16th International Conference on Computer Aided Verification (CAV)*, July 2004.

[Frohn94]
Access to Objects by Path Expressions and Rules. J. Frohn, G. Lausen, H. Uphoff. Intl. Conference on Very Large Databases, 1994, pp. 273-284.

[Gosling96]
*The Java language specification.*Gosling, James, Bill Joy, and Guy L. Steele. 1996. Reading, Mass.: Addison-Wesley.

[Grosof99a]
A Courteous Compiler From Generalized Courteous Logic Programs To Ordinary Logic Programs. B.N. Grosof. IBM Report included as part of documentation in the IBM CommonRules 1.0 software toolkit and documentation, released on http://alphaworks.ibm.com. July 1999. Also available at: http://ebusiness.mit.edu/bgrosof/#gclp-rr-99k.

[Grosof99b]
A Declarative Approach to Business Rules in Contracts. B.N. Grosof, J.K. Labrou, and H.Y. Chan. Proceedings of the 1st ACM Conference on Electronic Commerce (EC-99). Also available at: http://ebusiness.mit.edu/bgrosof/#econtracts+rules-ec99.

[Grosof99c]
DIPLOMAT: Compiling Prioritized Default Rules Into Ordinary Logic Programs (Extended Abstract of Intelligent Systems Demonstration). B.N. Grosof. IBM Research Report RC 21473, May 1999. Extended version of 2-page refereed conference paper appearing in Proceedings of the National Conference on Artificial Intelligence (AAAI-99), 1999. Also available at: http://ebusiness.mit.edu/bgrosof/#cr-ec-demo-rr-99b.

[Grosof2003a]
Description Logic Programs: Combining Logic Programs with Description Logic. B.N. Grosof, I. Horrocks, R. Volz, and S. Decker. Proceedings of the 12th International Conference on the World Wide Web (WWW-2003). Also available at: http://ebusiness.mit.edu/bgrosof/#dlp-www2003.

[Grosof2004a]
Representing E-Commerce Rules Via Situated Courteous Logic Programs in RuleML. B.N. Grosof. Electronic Commerce Research and Applications, 3:1, 2004, 2-20. Preprint version is also available at: http://ebusiness.mit.edu/bgrosof/#.

[Grosof2004b]
SweetRules: Tools for Semantic Web Rules and Ontologies, including Translation, Inferencing, Analysis, and Authoring. B.N. Grosof, M. Dean, S. Ganjugunte, S. Tabet, C. Neogy, and D. Kolas. http://sweetrules.projects.semwebcentral.org. Software toolkit and documentation. Version 2.0, Dec. 2004.

[Grosof2004c]
Hypermonotonic Reasoning: Unifying Nonmonotonic Logic Programs with First Order Logic. B.N. Grosof. http://ebusiness.mit.edu/bgrosof/#HypermonFromPPSWR04InvitedTalk. Slides from Invited Talk at Workshop on Principles and Practice of Semantic Web Reasoning (PPWSR04), Sep. 2004; revised Oct. 2004. Paper in preparation.

[Grosof2004d]
SweetPH: Using the Process Handbook for Semantic Web Services. B.N. Grosof and A. Bernstein. http://

ebusiness.mit.edu/bgrosof/#SweetPHSWSLF2F1204Talk. Slides from Presentation at SWSL Meeting, Dec. 9-10, 2004. *Note: Updates the design in the 2003 Working Paper "Beyond Monotonic Inheritance: Towards Semantic Web Process Ontologies" and describes implementation.*

[Grosof2004e]
SweetDeal: Representing Agent Contracts with Exceptions using Semantic Web Rules, Ontologies, and Process Descriptions. B.N. Grosof and T.C. Poon. International Journal of Electronic Commerce (IJEC), 8 (4):61-98, Summer 2004 Also available at: http://ebusiness.mit.edu/bgrosof/#sweetdeal-exceptions-ijec.

[Grosof2004f]
Semantic Web Rules with Ontologies, and their E-Business Applications. B.N. Grosof and M. Dean. Slides of Conference Tutorial (3.5-hour) at the 3rd International Semantic Web Conference (ISWC-2004). Available at: http://ebusiness.mit.edu/bgrosof/#ISWC2004RulesTutorial.

[Gruninger03a]
A Guide to the Ontology of the Process Specification Language. M. Gruninger. *Handbook on Ontologies in Information Systems.* R. Studer and S. Staab (eds.). Springer Verlag, 2003.

[Gruninger03b]
Process Specification Language: Principles and Applications. M. Gruninger and C. Menzel. *AI Magazine,* 24:63-74, 2003.

[Gruninger03c]
Applications of PSL to Semantic Web Services. M. Gruninger. *Workshop on Semantic Web and Databases. Very Large Databases Conference, Berlin.*

[Hayes04]
RDF Model Theory. Hayes, P. W3C, February 2004.

[Helland05]
Data on the Outside Versus Data on the Inside. P. Helland. *Proc. 2005 Conf. on Innovative Database Research (CIDR)*, January, 2005.

[Hull03]
E-Services: A Look Behind the Curtain. R. Hull, M. Benedikt, V. Christophides, J. Su. *Proc. of the ACM Symp. on Principles of Database Systems (PODS)*, San Diego, June, 2003.

[Kifer95]
Logical Foundations of Object-Oriented and Frame-Based Languages, M. Kifer, G. Lausen, J. Wu. Journal of ACM, 1995, 42, 741-843.

[Kifer04]
A Logical Framework for Web Service Discovery, M. Kifer, R. Lara, A. Polleres, C. Zhao. Semantic Web Services Workshop, November 2004, Hiroshima, Japan.

[Klein00a]
Towards a Systematic Repository of Knowledge About Managing Collaborative Design Conflicts. Klein, Mark. 2000. Proceedings of the Conference on Artificial Intelligence in Design. Boston, MA, USA.

[Klein00b]
A Knowledge-Based Approach to Handling Exceptions in Workflow Systems. Klein, Mark, and C. Dellarocas. 2000. Computer Supported Cooperative Work: The Journal of Collaborative Computing 9:399-412.

[Lloyd87]
Foundations of logic programming (second, extended edition). J. W. Lloyd. Springer series in symbolic computation. Springer-Verlag, New York, 1987.

[Lindenstrauss97]
Automatic Termination Analysis of Logic Programs. N. Lindenstrauss and Y. Sagiv. International Conference on Logic Programming (ICLP), 1997.

[Maier81]
Incorporating Computed Relations in Relational Databases. D. Maier, D.S. Warren. SIGMOD Conference, 1981: 176-187.

[Malone99]
Tools for inventing organizations: Toward a handbook of organizational processes. T. W. Malone, K. Crowston, J. Lee, B. Pentland, C. Dellarocas, G. Wyner, J. Quimby, C. Osborne, A. Bernstein, G. Herman, M. Klein, E. O'Donnell. *Management Science*, 45(3), pages 425--443, 1999.

[McIlraith01]
Semantic Web Services. *IEEE Intelligent Systems*, Special Issue on the Semantic Web, S. McIlraith, T. Son and H. Zeng. 16(2):46--53, March/April, 2001.

[Milner99]
Communicating and Mobile Systems: The π-Calculus. R. Milner. Cambridge University Press, 1999.

[Narayanan02]
Simulation, Verification and Automated Composition of Web Services. S. Narayanan and S. McIlraith. In *Proceedings of the Eleventh International World Wide Web Conference (WWW-11)*, May, 2002.

[Ontobroker]
Ontobroker 3.8. Ontoprise, GmbH.

[OWL Reference]
OWL Web Ontology Language 1.0 Reference. Mike Dean, Dan Connolly, Frank van Harmelen, James Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. W3C Working Draft 12 November 2002. Latest version is available at http://www.w3.org/TR/owl-ref/.

[OWL-S 1.1]
OWL-S: Semantic Markup for Web Services. David Martin, editor. Technical Overview (associated with OWL-S Release 1.1).

[Papazoglou03]
Service-Oriented Computing: Concepts, Characteristics and Directions. M.P. Papazoglou. Keynote for the 4th International Conference on Web Information Systems Engineering (WISE 2003), December 10-12, 2003.

[Perlis85]
Languages with Self-Reference I: Foundations. D. Perlis. Artificial Intelligence, 25, 1985, 301-322.

[Preist04]
A Conceptual Architecture for Semantic Web Services, C. Preist, 1993. In Proceedings of Third International Semantic Web Conference, Nov. 2004: 395-409.

[Reiter01]
Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems. Raymond Reiter. MIT Press. 2001

[Scherl03]
Knowledge, Action, and the Frame Problem. R. B. Scherl and H. J. Levesque. *Artificial Intelligence*, Vol. 144, 2003, pp. 1-39.

[Singh04]
Protocols for Processes: Programming in the Large for Open Systems. M. P. Singh, A. K. Chopra, N. V. Desai, and A. U. Mallya. *Proc. of the 19th Annual ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Vancouver, October 2004.

[SWSL Requirements]
Semantic Web Services Language Requirements. B. Grosof, M. Gruninger, et al, editors. White paper of the Semantic Web Services Language Committee.

[UDDI v3.02]
Universal Description, Discovery and Integration (UDDI) protocol. S. Thatte, editor. OASIS Standards Specification, February 2005.

[VanGelder91]
The Well-Founded Semantics for General Logic Programs. A. Van Gelder, K.A. Ross, J.S. Schlipf. Journal of ACM, 38:3, 1991, 620-650.

[WSCL 1.0]
Web Services Conversation Language (WSCL) 1.0. A. Banerji et al. W3C Note, March 14, 2002.

[WSDL 1.1]
Web Services Description Language (WSDL) 1.1. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. W3C Note, March 15, 2001.

[WSDL 2.0]
Web Services Description Language (WSDL) 2.0 -- Part 1: Core Language. R. Chinnici, M. Gudgin, J.-J. Moreau, J. Schlimmer, and S. Weerawarana. W3C Working Draft, August 3, 2004.

[WSDL 2.0 Primer]
Web Services Description Language (WSDL) Version 2.0 -- Part 0: Primer. D. Booth, C. Liu, editors. W3C Working Draft, 21 December 2004.

[WS-Choreography]
Web Services Choreography Description Language Version 1.0. N. Kavantzas, D. Burdett, et. al., editors. W3C Working Draft, December 17, 2004.

[XSLT]
XSL Transformations (XSLT) Version 1.0. J. Clark, editor. W3C Recommendation, 16 November 1999.

[XQuery 1.0]
XQuery 1.0: An XML Query Language. S. Boag, D. Chamberlin, et al, editors. W3C Working Draft 04 April 2005.

[Yang02]
Well-Founded Optimism: Inheritance in Frame-Based Knowledge Bases. G. Yang, M. Kifer. Intl. Conference on Ontologies, DataBases, and Applications of Semantics for Large Scale Information Systems (ODBASE), October 2002.

[Yang03]
Reasoning about Anonymous Resources and Meta Statements on the Semantic Web. G. Yang, M. Kifer. Journal on Data Semantics, Lecture Notes in Computer Science 2800, Springer Verlag, September 2003, 69-98.

[Yang04]
FLORA-2 User's Manual. G. Yang, M. Kifer, C. Zhao, V. Chowdhary. 2004.

Semantic Web Services Language (SWSL)

Version 1.0

Authors:

Steve Battle (Hewlett Packard) Abraham Bernstein (University of Zurich) Harold Boley (National Research Council of Canada) Benjamin Grosof (Massachusetts Institute of Technology) Michael Gruninger (NIST) Richard Hull (Bell Labs Research, Lucent Technologies) Michael Kifer (State University of New York at Stony Brook) David Martin (SRI International) Sheila McIlraith (University of Toronto) Deborah McGuinness (Stanford University) Jianwen Su (University of California, Santa Barbara) Said Tabet (The RuleML Initiative)

Abstract

This document defines the Semantic Web Services Language (SWSL), which is used to specify the Semantic Web Services Ontology (SWSO) as well as individual Web services. The language consists of two parts: SWSL-FOL, a full first-order logic language, and SWSL-Rules, as rule-based language. SWSL-FOL is primarily used for formal specification of the ontology and is intended to provide interoperability with other first-order based process models and service ontologies. In contrast, SWSL-Rules is designed to be an actual language for service specification.

Status of this document

This is one of four documents that make up the initial report of the Semantic Web Services Language Committee of the Semantic Web Services Initiative. The report defines the Semantic Web Services Framework (SWSF).

History of publication at http://www.daml.org/services/swsl/report/swsl/:

- v. 0.9: April 6, 2005
- v. 0.91: April 13, 2005
- v. 0.92: April 25, 2005
- v. 0.93: May 5, 2005

History of publication at http://www.daml.org/services/swsf/swsl/:

- v. 1.0: May 9, 2005

Table of contents

1 Introduction

This document is part of the technical report of the Semantic Web Services Language (SWSL) Committee of the Semantic Web Services Initiative (SWSI). The overall structure of the report is described in the document titled Semantic Web Services Framework Overview.

SWSL is a logic-based language for specifying formal characterizations of Web service concepts and descriptions of individual services. It includes two sublanguages: SWSL-FOL -- a full first-order logic language, which is used to specify the service ontology (SWSO), and SWSL-Rules -- a rule-based sublanguage, which can be used both as a specification and an implementation language. As a language, SWSL is domain-independent and *does not* include any constructs specific to services. Those constructs are defined by the Semantic Web Service Ontology, which appears in a separate document.

SWSL-Rules includes a novel combination of features that hitherto have not been present in a single system. However, almost all of the features of SWSL-Rules have been implemented in either FLORA-2 [*Yang04*], SweetRules [*Grosof2004b*], or the commercial Ontobroker [*Ontobroker*] system. Extensive feedback collected from the users of these systems has been incorporated in the design of the corresponding features in SWSL-Rules.

In contrast to SWSL-Rules, we do not envision the need for a full first-order reasoner based on SWSL-FOL. Instead, SWSL-FOL is intended largely as a specification language for SWSO, and specialized reasoners will be used to reason with the service ontology. In addition, SWSL-FOL will serve as a common platform to support semantic interoperability among the different first-order based service ontologies, such as OWL-S [*OWL-S 1.1*].

Relationship between SWSL-FOL and SWSL-Rules. SWSL includes two separate sublanguages, because we believe that different tasks associated with Semantic Web services are better served by different knowledge representation formalisms.

SWSL-Rules is a rule-based language with non-monotonic semantics. Such languages are better suited for tasks that have programming flavor and that naturally rely on default information and inheritance. These tasks include service discovery, contracting, policy specification, and others. In addition, rule-based languages are quite common both in the industry and research, and many people are more comfortable using them even for tasks that may not require defaults, such as service profile specification. Applications of SWSL and of the ontology built using SWSL are discussed in the Application Scenarios document.

In contrast, first-order logic is found more suitable for specifying process ontologies. One of the most prominent examples that uses this approach is PSL [*Gruninger03a*]. SWSL-FOL was developed to satisfy this need. Unfortunately, first-order and nonmonotonic semantics cannot be used together in the same language, so SWSL provides a "bridge" between the two sublanguages by describing how one can work in either sublanguage and use specifications written in the other sublanguage. This bridge is described in section titled Combining SWSL-Rules and SWSL-FOL.

The basic idea is as follows. First, as shown in Figure 2.2, both sublanguages share a common and useful core where they coincide both syntactically and semantically. Second, section Combining SWSL-Rules and SWSL-FOL describes a methodology for translating SWSL-FOL specifications into SWSL-Rules with "minimal loss." This means that inferences made using the translated specification are sound with respect to the original SWSL-FOL specification, and the "lost" inferences (i.e., formulas that are derivable from the original but not from the translated specification) are, in some sense, minimized. This approach was used in translating the axioms of PSL Core and PSL Outer Core into SWSL-Rules in section PSL in SWSL-FOL and SWSL-Rules.

The layered structure of SWSL. Both SWSL-Rules and SWSL-FOL are presented as *layered* languages. Unlike OWL, the layers are not organized based on the expressive power and computational complexity. Instead, each layer includes a number of new concepts that enhance the *modeling power* of the language. This is done in order to make it easier to learn the language and to help understand the relationship between the different features. Furthermore, most layers that extend the core of the language (either SWSL-Rules or SWSL-FOL) are *independent* from each other -- they can be implemented all at once or in any partial combination. This can provide certain guidance to vendors who might be interested only in a particular subset of the features.

Complexity. The layers of SWSL are not organized around the complexity. In fact, except for the equality layer, which boosts the complexity, all layers have the same complexity and decidability properties. For SWSL-Rules, the most important reasoning task is *query answering*. The general problem of query answering is known to be only semi-decidable. However, there are large classes of problems that are decidable in polynomial time. The best-known, and perhaps the most useful, subclass consists of rules that do not use function symbols. However, many decidable classes of rules *with* function symbols are also known [*Lindenstrauss97*].

2 The Language

2.1 Overview of SWSL-Rules and SWSL-FOL

As mentioned in the introduction, SWSL consists of two separate sublanguages, which have layered structure. This section gives an overview of these layers.

The SWSL-Rules language is designed to provide support for a variety of tasks that range from service profile specification to service discovery, contracting, policy specification, and so on. The language is layered to make it easier to learn and to simplify the use of its various parts for specialized tasks that do not require the full expressive power of SWSL-Rules. The layers of SWSL-Rules are shown in Figure 2.1.

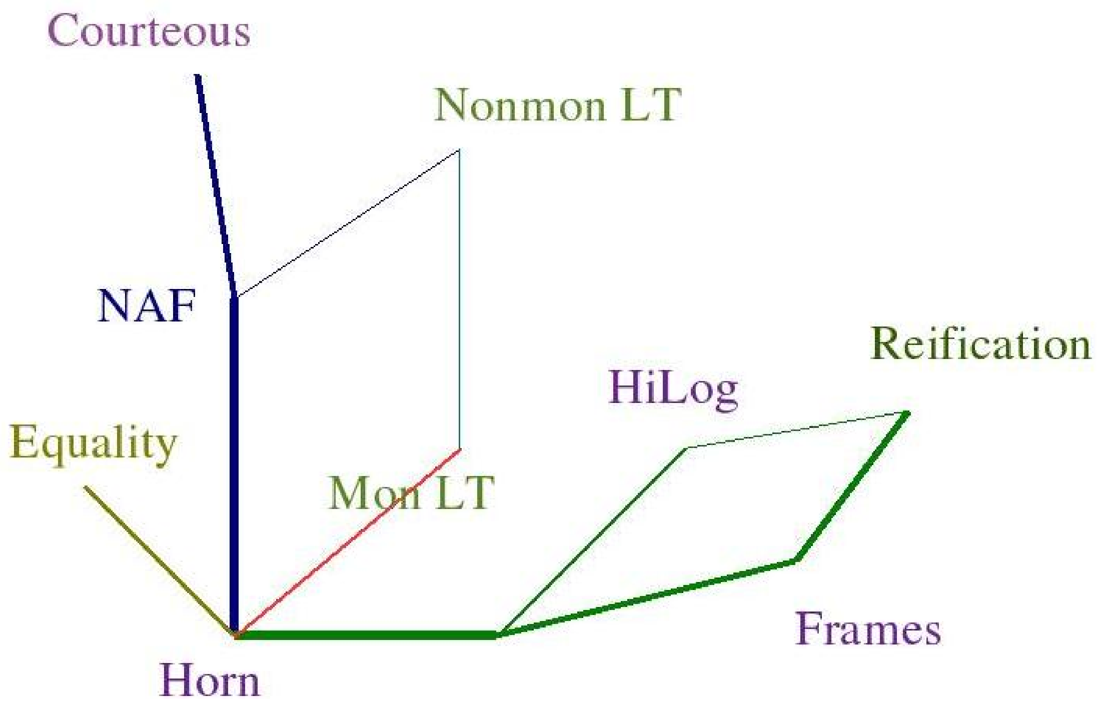

Figure 2.1: The Layered Structure of SWSL-Rules

The core of the language consists of the pure *Horn* subset of SWSL-Rules. The *monotonic Lloyd-Topor* (Mon LT) extension [*Lloyd87*] of the core permits disjunctions in the rule body and conjunction and implication in the rule head. *NAF* is an extension that allows negation in the rule body, which is interpreted as negation-as-failure. More specifically, negation is interpreted using the so called *well-founded semantics* [*VanGelder91*]. The *nonmonotonic Lloyd-Topor* extension (Nonmon LT) further permits quantifiers and implication in the rule body. The *Courteous rules* [*Grosof99a*] extension introduces two new features: restricted classical negation and prioritized rules. *HiLog* and *Frames* extend the language with a different kind of ideas. HiLog [*Chen93*] enables high degree of meta-programming by allowing variables to range over predicate symbols, function symbols, and even formulas. Despite these second-order features, the semantics of HiLog remains first-order and tractable. It has been argued [*Chen93*] that this semantics is more appropriate for many common tasks in knowledge representation than the classical second-order semantics. The *Frames* layer of SWSL-Rules introduces the most common object-oriented features, such as the frame syntax, types, and inheritance. The syntax and semantics of this extension is inspired by F-logic [*Kifer95*] and the followup works [*Frohn94*, *Yang02*, *Yang03*]. Finally, the *Reification* layer provides a mechanism for making objects out of a large class of SWSL-Rules formulas, which puts such formulas into the domain of discourse and allows reasoning about them.

All of the above layers have been implemented in one system or another and have been found highly valuable in knowledge representation. For instance, FLORA-2 [*Yang04*] includes all layers except Courteous rules and Nonmonotonic Lloyd-Topor. SweetRules [*Grosof2004b*] supports Courteous extensions, and Ontobroker [*Ontobroker*] supports Nonmonotonic Lloyd-Topor and frames.

Four points should be noted about the layering structure of SWSL-Rules.

1. The lines in Figure 2.1 represent inclusion dependencies among layers. For instance, Nonmonotonic LT layer includes both NAF and Monotonic LT. Reification includes HiLog and Frames, Courteous includes NAF, etc.
2. The different branches of Figure 2.1 are orthogonal and they *all* can be combined. For instance, the Frames and HiLog layers can be combined with the Courteous and Nonmon LT layers. Likewise, the equality layer can be combined with any other layer. Thus, SWSL-Rules is a unified language that combines all the layers into a coherent and powerful knowledge representation language.
3. Second, the Lloyd-Topor extensions and the Courteous rules extensions endow SWSL-Rules with all the normal first-order connectives. Therefore, *syntactically* SWSL-Rules contains all the connectives of the full first-order logic, which provides a bridge to SWSL-FOL. However, *semantically* the two sublanguages of SWSL are incompatible. Their semantics agree only over a relatively small, but useful subset of Horn rules. Section 5 discusses how the two sublanguages can be used together.
4. SWSL-Rules distinguishes between connectives with the classical first-order semantics and connectives that have nonmonotonic semantics. For instance, it uses two different forms of negation—`naf`, for negation-as-failure, and `neg`, for classical negation. Likewise, it distinguishes between the classical implication, <== and ==>, and the if-then connective `:-` used for rules.

SWSL-FOL is used to specify the dynamic properties of services, namely, the processes that they are intended to carry out. SWSL-FOL also has layered structure, which is depicted in Figure 2.2.

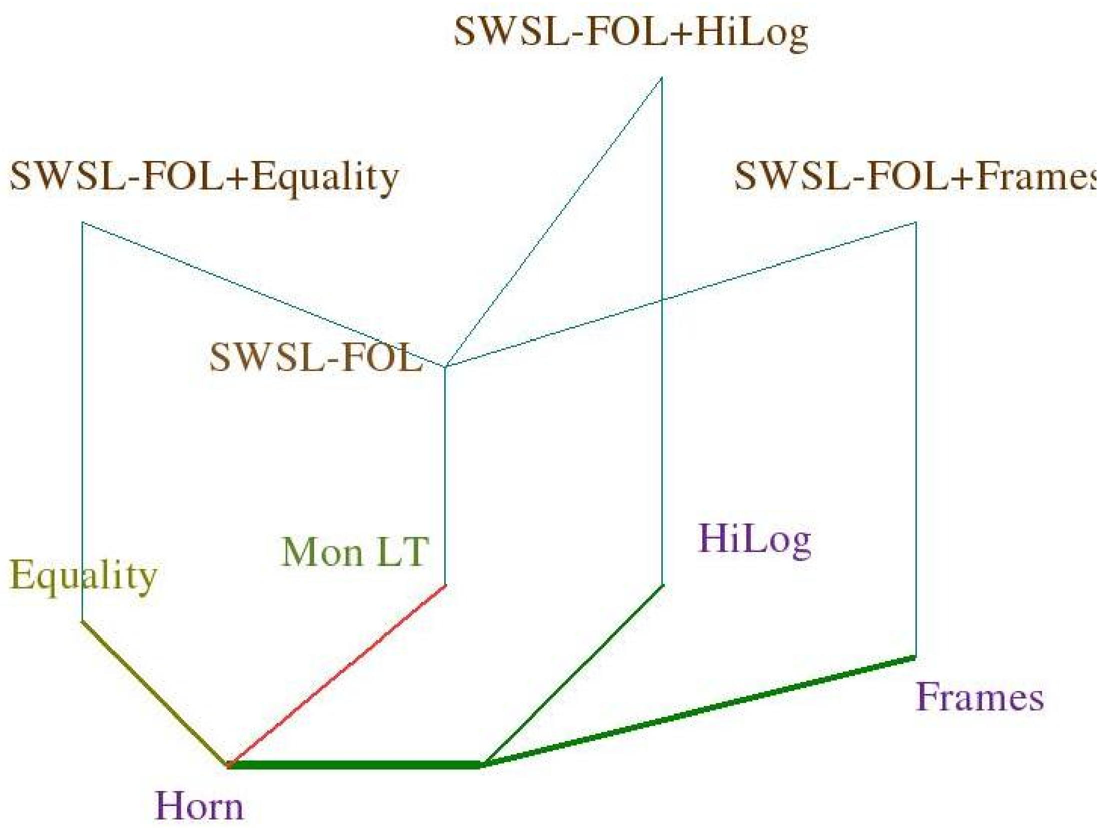

Figure 2.2: The Layers of SWSL-FOL and Their Relationship to SWSL-Rules

The bottom of Figure 2.2 shows those layers of SWSL-Rules that have monotonic semantics and therefore can be extended to full first-order logic. Above each layer of SWSL-Rules, the figure shows corresponding SWSL-FOL extension. The most basic extension is *SWSL-FOL*. The other three layers, *SWSL-FOL+Equality*, *SWSL-FOL+HiLog*, and *SWSL-FOL+Frames* extend *SWSL-FOL* both syntactically and semantically. Some of these extensions can be further combined into more powerful FOL languages. We discuss these issues in Section SWSL-FOL: The First-order Subset of SWSL.

2.2 Basic Definitions

In this section we define the basic syntactic components that are common to all layers of SWSL-Rules. Additional syntax will be added as more layers are introduced.

A *constant* is either a *numeric value*, a *symbol*, a *string*, or a *URI*.

- A ***numeric value*** is either an integer, a decimal, or a floating point number. For instance, `123`, `34.9`, `45e-11`. See the section on SWSL data types for more details on the relationship between SWSL data types and the primitive data types in XML Schema.

- A ***symbol*** is a string of characters enclosed between a pair of single quotes. For instance, `'abc#$%'`. Single quotes that are part of a symbol are escaped with the backslash. For instance, the symbol a`'bc''`d is represented as `'a\'bc\'\'d'`. The backslash is escaped with another backslash. Symbols that consist exclusively of alphanumeric characters and the underscore (_) and begin with a letter or an underscore do not need to be quoted.
- ***Strings*** are sequences of characters that are enclosed between a pair of double quote symbols, e.g., `"ab'%#cd"`. A double quote symbol that occurs in a string must be escaped with the backslash. For instance, the string `ab"cd"""gf` is represented as `"ab\"cd\"\"\"gf"`.
- A ***SWSL-URI*** can come either in the form of a *full URI* or in the abbreviated form of an *sQName*.

  A ***full URI*** is a sequence of characters that has the form of a URI, as specified by IETF, and is enclosed between `_"` and `"`. For instance, `_"http://w3.org/"`.

  An ***sQName*** has the form *prefix#local-name*. Here *prefix* is an alphanumeric symbol that is defined to be a shortcut for a URI as specified below; *local-name* is a string that must be acceptable as a path component in a URI. If *local-name* contains non-alphanumeric symbols, it must be enclosed in double quotes: e.g., "ab%20". An sQName is treated as a macro that expands into a full URI by concatenating the expansion of *prefix* (the URI represented by the prefix) with *local-name*. For the rationale behind the use of sQName see the entry for sQName in the Glossary.

A ***prefix declaration*** is a statement of the form

```
prefix prefix-name = "URI".
```

The prefix can then be used instead of the URI in sQNames. For instance, if we define

```
prefix w3 = "http://www.w3.org/TR/".
```

then the SWSL-URI `_"http://www.w3.org/TR/xquery/"` is considered to be equivalent to `w3#"TR/xquery/"`

A ***variable*** is an alphanumeric symbol (plus the underscore), which is prefixed with the ?-sign. Examples: `?_`, `?abc23`.

A ***first-order term*** is either a constant, a variable, or an expression of the form $t(t_1,...,t_n)$, where `t` is a constant and $t_1,...,t_n$ are first-order terms. Here the constant `t` is said to be used as a **function symbol** and $t_1,...,t_n$ are used as **arguments**. Variable-free terms are also called ***ground***.

Following Prolog, we also introduce special notation for lists: $[t_1,...,t_n]$ and $[t_1,...,t_n|rest]$, where $t_1,...,t_n$ and `rest` are first-order terms. The first form shows all the elements of the list explicitly and the latter shows explicitly only a prefix of the list and uses the first-order term `rest` to represent the tail. We should note that, like in Prolog, this is just a convenient shorthand notation. Lists are nothing but first-order terms that are representable with function symbols. For instance, if `cons` denotes a function symbol that prepends a term to the head of a list then `[a,b,c]` is represented as first-order term `cons(a,cons(b,c))`.

A ***first-order atomic formula*** has the same form as first-order terms except that a variable cannot be a first-order atomic formula. We do not distinguish predicates as a separate class of constants, as this is usually not necessary, since first-order atomic formulas can be distinguished from first-order terms by the context in which they appear.

As many other rule-based languages, SWSL-Rules has a special ***unification operator***, denoted =. The semantics of the unification operator is fixed and therefore it *cannot* appear in a rule head. An atomic formula of the form

$term_1 = term_2$

where both terms are ground, is true if and only if the two terms are identical. If $term_1$ and $term_2$ have variables, then an occurrence of the above formula in a rule body is interpreted as a test of whether a substitution exists that can make the two terms identical. The = predicate is related to the equality predicate `:=:` introduced by the Equality Layer, which is discussed later.

To test that two terms *do not* unify SWSL-Rules uses the ***disunification*** operator `!=`. For ground terms, $term_1$ `!=` $term_2$ iff the two terms are not identical. For non-ground terms, this is true if the two terms do not unify.

A ***conjunctive formula*** is either an atomic formula or a formula of the form

atomic formula `and` *conjunctive formula*

where `and` is a conjunction connective. Here and henceforth in similar definitions, italicized words will be meta-symbols that denote classes of syntactic entities. For instance, *atomic formula* above means ``any atomic formula.'' An ***and/or formula*** is either a conjunctive formula or a formula of either of the forms

conjunctive formula `or` *and/or formula*   *and/or formula* `and` *and/or formula*

In other words, an and/or formula can be an arbitrary Boolean combination of atomic formulas that involves the connectives `and` and `or`.

Comments. SWSL-Rules has two kinds of comments: single line comments and multiline comments. The syntax is the same as in Java. A ***single-line comment*** is any text that starts with a `//` and continues to the end of the current line. If `//` starts within a string ("...") or a symbol ('...') then these characters are considered to be part of the string or the symbol, and in this case they do not start a comment. A ***multiline comment*** begins with `/*` and end with a matching `*/`. The combination `/*` does not start a comment if it appears inside a string or a symbol. The `/*` - `*/` pairs can be nested and a nested occurrence of `*/` does not close the comment. For instance, in

```
/* start /* foobar */ end */
```

only the second `*/` closes the comment.

2.3 Horn Rules

A ***Horn rule*** has the form

```
head :- body.
```

where *head* is an atomic formula and *body* is a conjunctive formula.

A ***Horn query*** is of the form

```
?- query.
```

where *query* is a conjunctive formula.

Rules can be recursive, i.e., the predicate in the head of a rule can occur (with the same arity) in the body of the rule; or they can be mutually recursive, i.e., a head predicate can depend on itself through a sequence of rules.

All variables in a rule are considered *implicitly* quantified with ∀ outside of the rule, i.e., `∀?X,?Y,...(head :- body)`. A variable that occurs in the body of a rule but not its head can be equivalently considered as being implicitly existentially quantified in the body. For instance,

```
    ∀?X,?Y ( p(?X) :- q(?X,?Y) )
```

is equivalent to

```
    ∀?X ( p(?X) :- ∃?Y q(?X,?Y) )
```

Sets of Horn rules have the nice property that their semantics can be characterized in three different and independent ways: through the regular first-order entailment, as a minimal model (which in this case happens to be the intersection of all Herbrand models of the rule set) and as a least fixpoint of the immediate consequence operator corresponding to the rule set [*Lloyd87*].

2.4 The Monotonic Lloyd-Topor Layer

This layer extends the Horn layer with three kinds of syntactic sugar:

1. Disjunction in the rule body
2. Conjunction in the rule head
3. It introduces the new symbols of classical implication and allows their use in the rule head.

A ***classical implication*** is a statement of either of the following forms:

$formula_1$ `==>` $formula_2$ $formula_1$ `<==` $formula_2$

The ***Lloyd-Topor implication*** (abbr., LT implication) is a special case of the classical implication where the formula in the head is a conjunction of atomic formulas and the formula in the body can contain both conjunctions and disjunctions of atomic formulas.

A ***classical bi-implication*** is a statement of the form

$formula_1$ `<==>` $formula_2$

The ***Lloyd-Topor bi-implication*** (abbr., LT bi-implication) is a special case of the classical bi-implication where both formulas are conjunctions of atomic formulas.

The monotonic LT layer extends Horn rules in the following way. A rule still has the form

```
head :- body.
```

but *head* can now be a conjunction of atomic formulas and/or LT implications (including bi-implications) and *body* can consist of atomic formulas combined in arbitrary ways using the `and` and the `or` connectives.

This extension is considered a syntactic sugar, since semantically any set of extended rules reduces to another set of pure Horn rules as follows:

- *head* `:-` $body_1$ `or` $body_2$.

  reduces to

  head `:-` $body_1$.   *head* `:-` $body_2$.
- $head_1$ `and` $head_2$ `:-` *body*.

  reduces to

  $head_1$ `:-` *body*. $head_2$ `:-` *body*.
- `(`$head_1$ `<==` $head_2$`)` `:-` *body*.

  reduces to

```
    head1 :- head2 and body.
```

- (*head*$_1$ ==> *head*$_2$) :- *body*.

  reduces to

  ```
    head2 :- head1 and body.
  ```

Complex formulas in the head are broken down using the last three reductions. Rule bodies that contain both disjunctions and conjunctions are first converted into disjunctive normal form and then are broken down using the first reduction rule.

2.5 The NAF Layer

The NAF layer add the negation-as-failure symbol, `naf`. For instance,

```
p(?X,?Y) :- q(?X,?Z) and naf r(?Z,?Y).
```

In SWSL-Rules we adopt the ***well-founded semantics*** [*VanGelder91*] as a way to interpret negation as failure. This semantics has good computational properties when no first-order terms of arity greater than 0 are involved, and the well-founded model is always defined and is unique. This model is three-valued, so some facts may have the ``unknown'' truth value.

We should note one important convention regarding the treatment of variables that occur under the scope of `naf` and that do not occur anywhere outside of `naf` in the same rule. The well-founded semantics was defined only for ground atoms and the interpretation of unbound variables was left open. Therefore, if Z does not occur elsewhere in the rule then the meaning of

```
... :- ... and naf r(?X) and ...
```

can be defined as

```
... :- ... and ∃ X (naf r(?X)) and ...
```

or as

```
... :- ... and ∀ X (naf r(?X)) and ...
```

In practice, the second interpretation is preferred, and this is also a convention used in SWSL-Rules.

2.6 The Nonmonotonic Lloyd-Topor Layer

This layer introduces explicit bounded quantifiers (both `exist` and `forall`), classical implication symbols, <== and ==>, and the bi-implication symbol <==> in the rule body. This essentially permits arbitrary first-order-looking formulas in the body of SWSL-rules. We say "first-order-looking" because it should be kept in mind that the semantics of SWSL-Rules is *not* first-order and, for example, classical implication `A <== B` is interpreted in a non-classical way: as `(A or naf B)` rather than `(A or neg B)` (where `neg` denotes classical negation).

Recall that without explicit quantification, all variables in a rule are considered *implicitly* quantified with `forall` outside of the rule, i.e., `forall ?X,?Y,...(`*head* `:-` *body*`)`. A variable that occurs in the body of a rule but not its head can be equivalently considered as being implicitly existentially quantified in the body. For instance,

```
forall ?X,?Y ( p(?X) :- q(?X,?Y) )
```

is equivalent to

```
forall ?X ( p(?X) :- exist ?Y q(?X,?Y) )
```

In the scope of the `naf` operator, unbound variables have a different interpretation under negation as failure. For instance, if ?X is bound and ?Y is unbound then

```
p(?X) :- naf q(?X,?Y)
```

is actually supposed to mean

```
forall ?X ( p(?X) :- naf exist ?Y q(?X,?Y) )
```

If we allow explicit universal quantification in the rule bodies then implicit existential quantification is not enough and explicit existential quantifier is needed. This is because `forall` and `exist` do not commute and so, for example, `forall ?X exist ?Y` and `exist ?Y forall ?X` mean different things. If only implicit existential quantification were available, it would not be possible to differentiate between the above two forms.

Formally, the Nonmonotonic Lloyd-Topor layer permits the following kinds of rules. The rule ***heads*** are the same as in the monotonic LT extension. The rule ***bodies*** are defined as follows.

- Any atomic formula is a legal rule body
- If *f* and *g* are legal rule bodies then so are
  - *f* and *g*

        - *f* or *g*
        - `naf` *f*
        - *f* ==> *g*
        - *f* <== *g*
        - *f* <==> *g*
- If *f* is a legal rule body then so is
    - `exist ?X`$_1$`,...,?X`$_n$`(`*f*`)`

  where ?X$_1$, ..., ?X$_n$ are variables that occur *positively* (defined below) in *f*.
- If g_1, g_2 are legal rule bodies then
    - `forall ?X`$_1$`,...,?X`$_n$`(`g_1 ==> g_2`)`
    - `forall ?X`$_1$`,...,?X`$_n$`(`g_2 <== g_1`)`

  are legal rule bodies provided that ?X$_1$, ..., ?X$_n$ occur *positively* in g_1

Positive occurrence of a free variable in a formula is defined as follows:

- Any variable occurs positively in an atomic formula
- A free variable occurs positively in *f* `and` *g* iff it occurs positively in either *f* or *g*.
- A free variable occurs positively in *f* `or` *g* iff it occurs positively in both *f* and *g*.
- A free variable occurs positively in *f* ==> *g* iff it occurs positively in *g*. Similarly for *f* <== *g*, except that now the variable must occur positively in *f*. Since *f* <==> *g* is a conjunction of two clauses, the definition of positive occurrence follows from the previous cases: the variable must occur positively in *f* or *g*.
- A free variable occurs positively in `exist ?X`$_1$`,...,?X`$_n$`(`*f*`)` or `forall ?X`$_1$`,...,?X`$_n$`(`*f*`)` iff it occurs positively in *f*.

The semantics of Lloyd-Topor extensions is defined via a transformation into the NAF layer as shown below. The theory behind this transformation is described in [*Lloyd87*].

Lloyd-Topor transformation: The transformation is designed to eliminate the extended forms that may occur in the bodies of the rules compared to the NAF layer. These extended forms involve the various types of implication and the explicit quantifiers. Note that the rules, below, must be applied top-down, that is, to the conjuncts that appear directly in the rule body. For instance, if the rule body looks like

... `:-` ... and `((forall X exist Y (foo(Y,Y) ==> bar(X,Z))) <== foobar(Z))` and ...

then one should first apply the rule for <==, then the rules for `forall` should be applied to the result, and finally the rules for `exist`.

- Let the rule be of the form

  head `:-` *body*$_1$ and (*f* ==> *g*) and *body*$_2$.

  Then the LT transformation replaces it with the following pair of rules:

  head `:-` *body*$_1$ and `naf` *f* and *body*$_2$.   *head* `:-` *body*$_1$ and *g* and *body*$_2$.

  The transformations for <== and <==> are similar.

- Let the rule be

  head `:-` *body*$_1$ and `forall ?X`$_1$`,...,?X`$_n$`(`g_1 ==> g_2`)` and *body*$_2$.
  where `?X`$_1$`,...,?X`$_n$are free variables that occur positively in g_1.

  The LT transformation replaces this rule with the following pair of rules, where `q(?X'`$_1$`,...,?X'`$_n$`)` is a *new* predicate of arity n and `?X'`$_1$`,...,?X'`$_n$ are new variables:

  head `:-` *body*$_1$ and `naf q(?X'`$_1$`,...,?X'`$_n$`)` and *body*$_2$ `q(?X`$_1$`,...,?X`$_n$`) :-` g_1 and `naf` g_2.

  The transformation for <== is similar.

- Let the rule be

  head `:-` *body*$_1$ and `exist ?X`$_1$`,...,?X`$_n$`(`*f*`)` and *body*$_2$.

  where `?X`$_1$`,...,?X`$_n$ are free variables that occur positively in *f*.

  The LT transformation replaces this rule with the following:

  head `:-` *body*$_1$ and *f* and *body*$_2$

  That is, explicit existential quantification can be replaced in this case with implicit quantification.

The above transformations are inspired by (but are not derived from, due to a significant difference between `naf` and `neg`!) the classical tautologies `(`*f* `==>` *g*`) <==> (neg` *f* `or` *g*`)` and `forall X (f) <==> neg exist neg X (f)`, and by the fact mentioned in section The NAF Layer that `naf p(X)`, when X does not occur anywhere else in the rule, is interpreted as `forall X (naf p(X))`.

2.7 The Courteous Rules Layer

The courteous layer introduces *prioritized conflict handling*. Four new features are introduced into the syntax:

- ***rule labels***, which declare names used for prioritization between rules;
- ***classical negation*** of atoms;
- a syntactically reserved ***prioritization predicate***, which is used to specify the prioritization ordering between rules;
- mutual exclusion (***mutex***) statements, which specify the scope of what constitutes conflict.

The theory behind the courteous logic programs is described in [*Grosof2004a*, *Grosof99a*].

The courteous layer builds upon the NAF layer of SWSL.

Rule Labels: Each rule has an optional label, which is used for specifying prioritization in conjunction with the prioritization predicate (below). The syntactic form of a rule label is a term enclosed by a pair of braces: { ... }. Thus, a ***labeled rule*** has the following form:

```
{label} head :- body.
```

A *label* is a term, which may have variables. If so, these variables are interpreted as having the same scope as the implicitly quantified variables appearing in the rule expression. E.g., in the rule

```
{specialoffer(?X)} pricediscount(?X,tenpercent) :- loyalcustomer(?X).
```

the label `specialoffer(?X)` names the instance of the rule corresponding to the instance `?X`. However, the label term may not itself be a variable, so the following is illegal syntax:

```
{?X} pricediscount(?X,tenpercent) :- loyalcustomer(?X).
```

In general, labels are not unique; two or more rules (or instances of rules) may have the same label term. However, often it is convenient to specify rule labels uniquely within a particular given rulebase.

Classical Negation: The classical negation connective, `neg`, is permitted to appear within the head and/or the body of a rule. Its scope is restricted to be an atomic formula, however. Thus classical negation is restricted to appearing within a classical literal. For example:

```
neg boy(?X) :- humanchild(?X) and neg male(?X). {t14(?X,?Y)} p(?X,?Y) :- q(?X,?Y) and naf neg r(?X,?Y).
```

However, the following example is illegal syntax because `neg` negates a non-atomic formula.

```
u(?X) :- t(?X) and neg naf s(?X).
```

Note that the classical negation connective (`neg`) is also used in SWSL-FOL, the first-order subset of SWSL-Language. However, the semantics of classical negation in Courteous LP (and thus SWSL-Rules) is somewhat weaker than in FOL (and thus SWSL-FOL).

Prioritization Predicate: The prioritization predicate `_"http://www.ruleml.org/spec/vocab/#overrides"` specifies the prioritization ordering between rule labels, and thus between the rules labeled by those rule labels. The name of the prioritization predicate is syntactically reserved. In this document we will use the following prefix declaration

```
prefix r = "http://www.ruleml.org/spec/vocab/#"
```

and abbreviate the prioritization predicate using the sQName `r#overrides`. In the future, we might adopt a different prefix, such as `"http://www.swsi.org/swsl/reserved/#"`.

A statement `r#overrides(label1,label2)` indicates that the first argument, `label1`, has higher priority than the second argument, `label2`. For example, consider the following rulebase RBC1:

```
{rep} neg pacifist(?X) :- republican(?X). {qua} pacifist(?X) :- quaker(?X). {pri1} r#overrides(rep,qua).
```

Here, the prioritization atom `r#overrides(rep,qua)` specifies that `rep` has higher priority than `qua`. Continuing that example, suppose the rulebase RBC1 also includes the facts:

```
{fac1} republican(nixon). {fac2} quaker(nixon).
```

Then, under the courteous semantics, the literal `neg pacifist(nixon)` is entailed as a conclusion, and the literal `pacifist(nixon)` is *not* entailed as a conclusion, because the rule labeled `rep` has higher priority than the rule labeled `qua`.

The prioritization predicate `r#overrides`, while its name is syntactically reserved, is otherwise an ordinary predicate -- it can appear freely in rules in the head and/or body. This is useful for reasoning about the prioritization ordering.

Mutual exclusion (mutex) statements: The scope of what constitutes conflict is specified by mutual exclusion (mutex) statements, which are part of the rule base and can be viewed as a kind of integrity constraint. Each such statement says that it is contradictory for a particular pair of literals (known as the "opposers") to be inferred, if an optional condition (known as the "given") holds true. The courteous LP semantics enforce that the set of sanctioned conclusions respects (i.e., is consistent with) all the mutexes within the given rulebase. Common uses for mutexes include specifying that two unary predicates are disjoint, or that a relation is functional; examples of these uses are given below.

A mutex without a given condition has the following syntactic form:

```
!- lit1 and lit2 .
```

where `lit1` and `lit2` are classical literals. Intuitively, this statement means that it is a contradiction to derive both `lit1` and `lit2`. For example:

```
!- pricediscount(?CUST,fivepercent) and pricediscount(?CUST,tenpercent).
```

says that it is a contradiction to conclude that the discount offered to the same customer `?CUST` is both `fivepercent` and `tenpercent`. As another example,

```
!- lion(?X) and elephant(?X).
```

specifies that it is a contradiction to conclude that the same individual is both a lion and an elephant.

A mutex with a *condition* has the following syntactic form:

```
!- lit1 and lit2 | condition .
```

Here *`condition`* is syntactically similar to a rule body, and `lit1` and `lit2` are classical literals. The symbol "`|`" is a language keyword, which separates the oposing literals from the condition. For example:

```
!- pricediscount(?CUST,?Y) and pricediscount(?CUST,?Z) | ?Y != ?Z.
```

says that it is a contradiction to conclude that the discount offered to the same customer, `?CUST`, is both `?Y` and `?Z` *if* `?Y` and `?Z` are distinct values. This means that the relation `pricediscount` is functional.

Courteous LP also assumes that there is an implicit mutex between each atom *A* and its classical negation `neg` *A*. This implicit mutex is also known as a "classical" mutex.

2.8 The HiLog Layer

HiLog [*Chen93*] extends the first-order syntax with higher-order features. In particular, it allows variables to range over function symbols, predicate symbols, and even atomic formulas. These features are useful for supporting reification and in cases when an agent needs to explore the structure of an unknown piece of knowledge. HiLog further supports parameterized predicates, which are useful for generic definitions (illustrated below).

- ***HiLog term*** (abbr., H-term): A HiLog term is either a first-order term or an expression of the following form: $t(t_1,...,t_n)$, where t, t_1, ..., t_n are HiLog terms.

This definition may seem quite similar to the definition of complex first-order terms, but, in fact, it defines a vastly larger set of expressions. In first-order terms, `t` must be a constant, while in HiLog it can be any HiLog term. In particular, it can be a variable or even another first-order term. For instance, the following are legal HiLog terms:

- *Regular first-order terms*: `c, f(a,?X), ?X`
- *Variables over function symbols*: `?X(a,?Y), ?X(a,?Y(?X))`
- *Parameterized function symbols*: `f(?X,a)(b,?X(c)), ?Z(?X,a)(b,?X(?Y)(d)), ?Z(f)(g,a)(p,?X)`

We will see soon how such terms can be useful in knowledge representation.

- ***HiLog atomic formula***: Any HiLog term is also a HiLog atomic formula.

Thus, expressions like `?X(a,?Y(?X))` are atomic formulas and thus can have truth values (when the variables are instantiated or quantified). What is less obvious is that `?X` is also an atomic formula. What all this means is that atomic formulas are automatically reified and can be passed around by binding them to variables and evaluated. For instance, the following HiLog query

```
?- q(?X) and ?X.       p(a).       q(p(a)).
```

succeeds with the above database and `?X` gets bound to `p(a)`.

Another interesting example of a HiLog rule is

```
call(?X) :- ?X.
```

This can be viewed as a logical definition of the meta-predicate `call/1` in Prolog. Such a definition does not make sense in first-order logic (and is, in fact, illegal), but it is legal in HiLog and provides the expected semantics for `call/1`.

We will now illustrate one use of the *parameterized* predicates of the form `p(...)(...)`. The example shows a pair of rules that defines a generic transitive closure of a binary predicate. Depending on the actual predicate passed in as a parameter, we can get different transitive closures.

```
closure(?P)(?X,?Y) :- ?P(?X,?Y).       closure(?P)(?X,?Y) :- ?P(?X,?Z) and closure(?P)(?Z,?Y).
```

For instance, for the `parent` predicate, `closure(parent)` is defined by the above rules to be the ancestor relation; for the `edge` relation that represents edges in a graph, `closure(edge)` will become the transitive closure of the graph.

2.9 The Equality Layer

This layer introduces the full ***equality predicate***, `:=:`. The equality predicate obeys the usual congruence axioms for equality. In particular, it is transitive, symmetric, reflexive, and the logical entailment relation is invariant with respect to the substitution of equals by equals. For instance, if we are told that `bob :=: father(tom)` (`bob` is the same individual as the one denoted by the term `father(tom)`) then if `p(bob)` is known to be true then we should be able to derive `p(father(tom))`. If we are also told that `bob :=: uncle(mary)` is true then we can derive `father(tom):=: uncle(mary)`.

Equality in a Semantic Web language is important to be able to state that two different identifiers represent the same resource. For that reason, equality was part of OWL [*OWL Reference*]. Although equality drastically increases the computational complexity, some forms of equality, such as ground equality, can be handled efficiently in a rule-based language.

The equality predicate `:=:` is different from the unification operator = in several respects. First, for variable free terms, $term_1$ = $term_2$ if and only if the two terms are identical. In contrast, as we have just seen, two distinct terms can be equal with respect to `:=:`. Since `:=:` is reflective, it follows that the interpretation of `:=:` always contains the interpretation of =. Second, the unification operator = cannot appear in a rule head, while the equality predicate `:=:` can. When `:=:` occurs in the rule head (or as a fact), it is an assertion (maybe conditional) that two terms are equal. For instance, given the above definitions,

```
    p(1,2).      p(2,3).      f(a,?X):=:g(?Y,b) :- p(?X,?Y).
```

entails the following equalities between distinct terms: `f(a,1):=:g(2,b)` and `f(a,2):=:g(3,b)`.

When $term_1$ `:=:` $term_2$ occurs in the body of a rule and $term_1$ and $term_2$ have variables, this predicate is interpreted as a test that there is a substitution that makes the two terms equal with respect to `:=:` (note: equal, not identical!). For instance, in the query

```
    q(1).      q(2).      q(3).      ?- f(a,?X):=:g(?Y,b) and q(?Y).
```

one answer substitution is `?X/1,?Y/2` and the other is `?X/2,?Y/3`.

2.10 The Frames Layer

The Frames layer introduces object-oriented syntax modeled after F-logic [*Kifer95*] and its subsequent enhancements [*Yang02*, *Yang03*]. The main syntactic additions of this layer include

- Frame syntax. Frames are called *molecules* here (following the F-logic terminology).
- Path expressions.
- Notation for class membership and subclasses.
- Notation for type specification, which is given by *signature molecules*.

The object-oriented extensions introduced by the Frames layer are orthogonal to the other layers described so far and can be combined with them within the SWSL-Rules language.

As in most object-oriented languages, the three main concepts in the Frames layer of SWSL-Rules are *objects*, *classes*, and *methods*. (We are borrowing from the object-oriented terminology here rather than AI terminology, so we are refer to *methods* rather than *slots*.) Any class is also an object, and the same expression can denote an object or a class represented by this object in different contexts.

A ***method*** is a function that takes arguments and executes in the context of a particular object. When invoked, a method returns a result and can possibly alter the state of the knowledge base. A method that does not take arguments and does not change the knowledge base is called an ***attribute***. An object is represented by its *object Id*, the values of its attributes, and by the definitions of its methods. Method and attribute names are represented as objects, so one can reason about them in the same language.

An ***object Id*** is syntactically represented by a ground term. Terms that do have variables are viewed as templates for collections of object Ids—one Id per ground instantiation of all the variables in the term. By *term* we mean any expression that can bind a variable. What constitutes a legal term depends on the layer. In the basic case, by term we mean just a first-order term. If the Frames layer is combined with HiLog, then terms are meant to be HiLog terms. Later, when we introduce reification, *reification terms* will also be considered.

Molecules. Molecules play the role of atomic formulas. We first describe atomic molecules and then introduce complex molecules. Although both atomic and complex molecules play the role of atomic formulas, complex molecules are *not* indivisible. This is why they are called molecules and not atoms. Molecules come in several different forms:

- **Value molecule**. If `t`, `m`, `v` are terms then `t[m -> v]` is a value molecule.

  Here `t` denotes an object, `m` denotes a ***method invocation*** in the scope of the object `t`, and `v` denotes a value that belongs to a set returned by this invocation. We call `m` ``a method invocation'' because if `m` = $s(t_1,...,t_n)$, i.e., has arguments, then $t[s(t_1,...,t_n)$ `-> v]` is interpreted as an invocation of method `s` on arguments $t_1,...,t_n$ in the context of the object `t`, which returns a set of values that contains `v`.

  The syntax `t[m ->` $\{v_1,...,v_k\}$`]` is also supported; it means that if `m` is invoked in the context of the object `t` then it returns a set that contains $v_1,...,v_k$. Thus, semantically, such a term is equivalent to a conjunction of `t[m ->` v_1`]`, ..., `t[m ->` v_k`]`, so the expressions `t[m ->` $\{v_1,...,v_k\}$`]` is just a syntactic sugar.

- **Boolean valued molecule**. These molecules have the form `t[m]` where `t` and `m` are terms.

  Boolean molecules are useful to specify things like `mary[female]`. The same could be alternatively written as `mary[female -> true]`, but this is less natural.

- **Class membership molecule**. If `t` and `s` are terms then `t:s` is a membership molecule.

  If `t` and `s` are variable free, then such a molecule states that the object `t` is a member of class `s`. If these terms contain variables, then such a molecule can be viewed as many class membership statements, one per ground instantiation of the variables.

- **Subclass molecule**: If `t` and `s` are terms then `t::s` is a subclass molecule.

If `t` and `s` are variable free, then such a molecule states that the object `t` is a subclass of `s`. As in the case of class membership molecules, subclass molecules that have variables can be viewed as statements about many subclass relationships.

- **Signature molecule**: If `t`, `m`, `v` are terms then `t[m => v]` is a signature molecule.

  If `t`, `m`, and `v` are variable-free terms then the informal meaning of the above signature molecule is that `t` represents a class, which has a method invocation `m` which returns a set of objects of type `v` (i.e., each object in the set belongs to class `v`). If these terms are non-ground then the signature represents a collection of statements—one statement per ground instantiation of the terms.

  When `m` itself has arguments, for instance $m = s(t_1,...,t_n)$, then the arguments are interpreted as types. Thus, $t[s(t_1,...,t_n) => v]$ states that when the n-ary method `s` is invoked on object of class `t` with arguments that belong to classes $t_1, ..., t_n$, the method returns a set of objects of class `v`.

- **Boolean signature molecules**: A Boolean signature molecule has the form `t[m=>]`. Its purpose is to provide type information for Boolean valued molecules. Namely, if $m=s(t_1,...,t_n)$, then when the method `s` is invoked on an object of class `t`, the method arguments must belong to classes $t_1, ..., t_n$.
- **Cardinality constraints**: Signature molecules can have associated cardinality constraints. Such molecules have the form

  $t[s(t_1,...,t_n)\ \{min : max\} => v]$

  where *min* and *max* are non-negative integers such that *min* ≤ *max*. *Max* can also be `*`, which means positive infinity.

  Such a signature states not only that the invocation of the method `s` with arguments of type $t_1,...,t_n$ on an object of class `t` returns objects of class `v`, but also that the number of such objects in the result is no less than *min* and no more than *max*.

  The semantics of constraints in SWSL-Rules is similar to constraints in databases and is *unlike* the cardinality restrictions in OWL [*OWL Reference*]. For instance, if a cardinality constraint says that an attribute should have at least two values and the rule base derives only one then the constraint is *violated*. In contrast, OWL would *infer* that there is another, yet unknown, value. Likewise, if a cardinality constraint says that the number of elements is at most three while the rule base derives four unequal elements then the constraint is, again, violated. This should be compared to the OWL semantics, which will infer that *some* pair of derived values in fact consists of equal elements.

Signatures and type checking: Signatures are assertions about the expected types of the method arguments and method results. They typically do not have direct effect on the inference (unless signatures appear in rule bodies). The signature information is optional.

The semantics of signatures is defined as follows. First, the intended model of the knowledge base is computed (which in SWSL-Rules is taken to be the well-founded model). Then, if typing needs to be checked, we must verify that this intended model is well-typed. A ***well-typed*** model is one where the value molecules conform to their signatures. For the precise definition of well-typed models see [*Kifer95*]. (There can be several different notions of well-typed models. For instance, one for semi-structured data and another for completely structured data.)

A type-checker can be written in SWSL-Rules using just a few rules. Such a type checker is a query, which returns "No", if the model is well-typed and a counterexample otherwise. In particular, type-checking has the same complexity as querying. An example of such type checker can be found in the FLORA-2 manual [*Yang04*].

It is important to be aware of the fact that the semantics of the cardinality constraints in signature molecules is inspired by database theory and practice and it is *different* from the semantics of such constraints in OWL [*OWL Reference*]. In SWSL-Rules, cardinality constraints are restrictions on the intended models of the knowledge base, but they are not part of the axioms of the knowledge base. Therefore, the intended models of the knowledge base are determined without taking the cardinality constraints into the account. Intended models that do not satisfy these restrictions are discarded. In contrast, in OWL cardinality constraints are represented as logical statements in the knowledge base and all models are computed by taking the constraints into the account. Therefore, in OWL it is not possible to talk about knowledge base updates that violate constraints. For instance, the following signature `married[spouse {1:1} => married]` states that every married person has exactly one spouse. If `john:married` is true but there is no information about John's spouse then OWL will assume that `john` has some unknown spouse, while SWSL-Rules will reject the knowledge base as inconsistent. If, instead, we know that `john[spouse -> mary]` and `john[spouse -> sally]` then OWL will conclude that `mary` and `sally` are the same object, while SWSL-Rules will again rule the knowledge base to be inconsistent (because, in the absence of the information to the contrary — for example, if no `:=:`-statements have been given — `mary` and `sally` will be deemed to be distinct objects).

Inheritance in SWSL-Rules: Inheritance is an optional feature, which is expressed by means of the syntactic features described below. In SWSL-Rules, methods and attributes can be inheritable and non-inheritable. Non-inheritable methods/attributes correspond to class methods in Java, while inheritable methods and attributes correspond to instance methods.

The value- and signature-molecules considered so far involve *non-inheritable* attributes and methods. Inheritable methods are defined using the `*->` and `*=>` arrow types, i.e., `t[m *-> v]` and `t[m *=> v]`. For Boolean methods we use `t[*m]` and `t[m*=>]`.

Signatures obey the laws of ***monotonic inheritance***, which are as follows:

- `t:s and s[m *=> v] entails t[m *=> v]`
- `t::s and s[m *=> v] entails t[m => v]`

These laws state that type declarations for inheritable methods are inherited to subclasses in an inheritable form, i.e., they can be further inherited. However, to the *members* of a class such declarations are inherited in a non-inheritable form. Thus, inheritance of signatures is propagated through subclasses, but stops once it hits class members.

Inheritance of value molecules is more involved. This type of inheritance is ***nonmonotonic*** and it can be overridden if the same method or attribute is defined for a more specific class. More precisely,

- `t:s and s[m *-> v] entails t[m *-> v]   unless overridden or in conflict`
- `t::s and s[m *-> v] entails t[m -> v]   unless overridden or in conflict`

Similarly to signatures, value molecules are inherited to subclasses in the inheritable form and to members of the classes in the non-inheritable form. However, the key difference is the phrase "unless overridden or in conflict." Intuitively, this means that if, for example, there is a class `w` in-between `t` and `s` such that the inheritable method `m` is defined there then the inheritance from `s` is blocked and `m` should be inherited from `w` instead. Another situation when inheritance might be blocked arises due to multiple inheritance conflicts. For instance, if `t` is a subclass of both `s` and `u`, and if both `s` and `u` define the method `m`, then inheritance of `m` does not take place at all (either from `s` or from `u`; this policy can be modified by specifying appropriate rules, however). The precise model-theoretic semantics of inheritance with overriding is based on an extended form of the Well-Founded Semantics. Details can be found in [*Yang02*].

Note that signature inheritance is not subject to overriding, so *every* inheritable molecule is inherited to subclasses and class instances. If multiple molecules are inherited to a class member or a subclass, then all of them are considered to be true.

Inheritance of Boolean methods is similar to the inheritance of methods and attributes that return non-Boolean values. Namely,

- `t:s and s[m*=>] entails t[m*=>]`
- `t::s and s[m*=>] entails t[m =>]`
- `t:s and s[*m] entails t[*m]   unless overridden`
- `t::s and s[*m] entails t[m]   unless overridden`

Complex molecules: SWSL-Rules molecules can be combined into complex molecules in two ways:

- By grouping.
- By nesting.

Grouping applies to molecules that describe the same object. For instance,

```
      t[m1 -> v1] and t[m2 => v2] and t[m3 {6:9} => v3] and t[m4 -> v4]
```

is, by definition, equivalent to

```
      t[m1 -> v1 and m2 => v2 and m3 {6:9} => v3 and m4 -> v4]
```

Molecules connected by the `or` connective can also be combined using the usual precedence rules:

```
      t[m1 -> v1] and t[m2 => v2] or t[m3 {6:9} => v3] and t[m4 -> v4]
```

becomes

```
      t[m1 -> v1 and m2 => v2 or m3 {6:9} => v3 and m4 -> v4]
```

The `and` connective inside a complex molecule can also be replaced with a comma, for brevity. For example,

```
      t[m1 -> v1, m2 => v2]
```

Nesting applies to molecules in the following ``chaining'' situation, which is a common idiom in object-oriented databases:

```
      t[m -> v] and v[q -> r]
```

is by definition equivalent to

```
      t[m -> v[q -> r]]
```

Nesting can also be used to combine membership and subclass molecules with value and signature molecules in the following situations:

```
      t:s and t[m -> v]       t::s and t[m -> v]
```

are equivalent to

```
      t[m -> v]:s       t[m -> v]::s
```

respectively.

Molecules can also be ***nested inside predicates*** and terms with a semantics similar to nesting inside other molecules. For instance, `p[a->c]` is considered to be equivalent to `p(a) and a[b->c]`. Deep nesting and, in fact, nesting in any part of another molecule or predicate is also allowed. Thus, the formulas

```
      p(f(q,a[b -> c]),foo)       a[b -> foo(e[f -> g])]       a[foo(b[c -> d]) -> e]       a[foo[b -> c] -> e]       a[b -> c](q,
r)
```

are considered to be equivalent to

```
      p(f(q,a),foo) and a[b -> c]       a[b -> foo(e)] and e[f -> g]       a[foo(b) -> e] and b[c -> d]       a[foo -> e] and foo
[b -> c]       a[b -> c] and a(q,r)
```

respectively. Note that molecule nesting leads to a completely ***compositional syntax***, which in our case means that molecules are allowed in any place where terms are allowed. (Not all of these nestings might look particularly natural, e.g., `a[b -> c](q,r)` or `p(a[b -> c](?X))`, but there is no good reason to reject these nestings and thus complicate the syntax either.)

Path expressions: Path expressions are useful shorthands that are widely used in object-oriented and Web languages. In a logic-based language, a path expression sometimes allows writing formulas more concisely by eliminating multiple nested molecules and explicit variables. SWSL-Rules defines path expressions only as replacements for value molecules, since this is where this shorthand is most useful in practice.

A ***path expression*** has the form

$$t.t_1.t_2. \; ... \; .t_n$$

or

$$t!t_1!t_2! \; ... \; !t_n$$

The former corresponds to non-inheritable molecules and the latter to inheritable ones. In fact, "." and "!" can be mixed within the same path expression.

A path expression can occur anywhere where a term is allowed to occur. For instance, `a[b -> c.d]`, `a.b.c[e -> d]`, `p(a.b)`, and `X=a.b` are all legal formulas. The semantics of path expressions in the body of a rule and in its head are similar, but slightly different. This difference is explained next.

In the *body* of a rule, an occurrence of the first path expression above is treated as follows. The conjunction

$$t[t_1 \text{ -> } ?Var_1] \text{ and } ?Var_1[t_2 \text{ -> } ?Var_2] \text{ and ... and } ?Var_{n-1}[t_n \text{ -> } ?Var_n]$$

is added to the body and the occurrence of the path expression is replaced with the variable $?Var_n$. In this conjunction, the variables $?Var_1$, ..., $?Var_n$ are *new* and are used to represent intermediate values. The second path expression is treated similarly, except that the conjunction

$$t[t_1 \text{ *-> } ?Var_1] \text{ and } ?Var_1[t_2 \text{ *-> } ?Var_2] \text{ and ... and } ?Var_{n-1}[t_n \text{ *-> } ?Var_n]$$

is used. For instance, `mary.father.mother = sally` in a rule body is replaced with

```
mary[father -> ?F] and ?F[mother -> ?M] and ?M = sally
```

In the *head* of a rule, the semantics of path expressions is reduced to the case of a body occurrence as follows.. If a path expression, ρ, occurs in the head of a rule, it is replaced with a new variable, ?V, and the predicate ?V=ρ is conjoined to the body of the rule. For instance,

```
p(a.b) :- body.
```

is understood as

```
p(?V) :- body and ?V=a.b.
```

Note that since molecules can appear wherever terms can, path expressions of the form `a.b[c -> d].e.f[g -> h].k` are permitted. They are conceptually similar to XPath expressions with predicates that control the selection of intermediate nodes in XML documents. Formally, such a path expression will be replaced with the variable ?V and will result in the addition of the following conjunction:

```
a[b -> ?X[c -> d]] and ?X[e -> ?Y] and ?Y[f -> ?Z[g -> h]] and ?Z[k -> ?V]
```

It is instructive to compare SWSL-Rules path expressions with XPath. SWSL-Rules path expressions were originally proposed for F-logic [*Kifer95*] several years before XPath. The purpose was to extend the familiar notation in object-oriented programming languages and to adapt it to a logic-based language. It is easy to see that the ``*'' idiom of XPath can be captured with the use of a variable. For instance, `b/*/c` applied to object e is expressed as `e.b.?X.c`. The ``..'' idiom of XPath is also easy to express. For instance, `a/../b/c` applied to object d is expressed as `?_[?_ -> d.a].b.c`. On the other hand, there is no counterpart for the `//` idiom of XPath. The reason is that this idiom is not well-defined when there are cycles in the data (for instance, `a[b -> a]`). However, recursive descent into the object graph can be defined via recursive rules.

2.11 Reification

The ***reification layer*** allows SWSL-Rules to treat certain kinds of formulas as terms and therefore to manipulate them, pass them as parameters, and perform various kinds of reasoning with them. In fact, the HiLog layer already allows certain formulas to be reified. Indeed, since any HiLog term is also a HiLog atomic formula, such atomic formulas are already reifiable. However, the reification layer goes several steps further by supporting reification of arbitrary rule or formula that can occur in the rule head or rule body. (Provided that it does not contain explicit quantifiers -- see below.)

Formally, if *F* is a formula that has the syntactic form of a rule head, a rule body, or of a rule then *F* is also considered to be a term. This means that such a formula can be used wherever a term can occur.

Note that a reified formula represents an objectification of the corresponding formulas. This is useful for specifying ontologies where objects represent theories that can be true in some worlds, but are not true in the present world (and thus those theories cannot be asserted in the present world). Examples include the effects of actions: effects of an action might be true in the world that will result after the execution of an action, but they are not necessarily true now.

In general, reification of formulas can lead to logical paradoxes [*Perlis85*]. The form of reification used in SWSL-Rules does not cause paradoxes, but other unpleasantries can occur. For instance, the presences of a truth axiom (**`true`**`(?X) <--> ?X`) can render innocently looking rule-bases inconsistent. However, as shown in [*Yang03*], the form of reification in SWSL-Rules does not cause paradoxes as long as

- rule heads do not contain classical negation; and
- a rule head cannot be a variable, i.e., as long as the rules of the form `?X :- body` (which are legal in HiLog) are disallowed.

We therefore adopt the above restrictions for all layers of SWSL-Rules (but not for SWSL-FOL).

As presented above, reification introduces syntactic ambiguity, which arises due to the nesting conventions for molecules. For instance, consider the following molecule:

```
a[b -> t]
```

Suppose that `t` is a reification of another molecule, `c[d -> e]`. Since we have earlier said that any formula suitable to appear in the rule body can also be viewed as a term, we can expand the above formula into

```
a[b -> c[d -> e]]
```

But this is ambiguous, since earlier we defined the above as a commonly used object-oriented idiom, a syntactic sugar for

```
a[b -> c] and c[d -> e]
```

Similarly, if we want to write something like `t[b -> c]` where `t` is a reification of `f[g -> h]` then we cannot write `f[g -> h][b -> c]` because this nested molecule is a syntactic sugar for `f[g -> h] and f[b -> c]`. To resolve this ambiguity, we introduce the reification operator, `${...}`, whose only role is to tell the parser that a particular occurrence of a nested molecule is to be treated as a term that represents a reified formula rather than as syntactic sugar for the object-oriented idiom.

Note that the explicit reification operator is not required for HiLog predicates because there is no ambiguity. For instance, we do not need to write `${p(?X)}` below (although it *is* permitted and is considered the same as `p(?X)`):

```
a[b -> p(?X)]
```

This is because `a[b -> p(?X)]` does *not* mean `a[b -> p(?X)] and p(?X)`, since the sugar is used only for nested molecules.

In contrast, explicit reification is needed below, if we want to reify `p(?X[foo -> bar])`:

```
a[b -> p(?X[foo -> bar])]
```

Otherwise `p(?X[foo -> bar])` would be treated as syntactic sugar for sugar for

```
a[b -> p(?X)] and ?X[foo -> bar]
```

Therefore, to reify `p(?X[foo -> bar])` in the above molecule one must write this instead:

```
a[b -> ${p(?X[foo -> bar])}]
```

Example. Reification in SWSL-Rules is very powerful and yet it doesn't add to the complexity of the language. The following fragment of a knowledge base models an agent who believes in the modus ponens rule:

```
john[believes -> ${p(a)}].       john[believes -> ${p(?X) ==> q(?X)}].       // modus ponens       john[believes -> ?A] :-
       john[believes -> ${?B ==> ?A}] and john[believes -> ?B].
```

Since the agent believes in `p(a)` and in the modus ponens rule, it can infer `q(a)`. Note that in the above we did not need explicit reification of `p(a)`, since no ambiguity can arise. However, we used the explicit reification anyway, for clarity.

Syntactic rules. Currently SWSL-Rules does not permit explicit quantifiers under the scope of the reification operator, because the semantics for reification given in [Yang03, Kifer04] does not cover this case. So not every formula can be reified. More specifically, the formulas that are allowed under the scope of the reification operator are:

- The formulas that are allowed in the rule head or quantifier-free formulas in the rule body.
- Quantifier-free rules.

The implication of these restrictions is that every term that represents a reification of a SWSL-Rules formula has only free variables, which can be bound outside of the term. Each such term can therefore be viewed as a (possibly infinite) set of reifications of the ground instances of that formula.

2.12 Skolemization in SWSL-Rules

It is often necessary to specify existential information in the head of a rule or in a fact. Due to the limitations of the logic programming paradigm, which trades the expressive power for executional efficiency, such information cannot be specified directly. However, existential variables in the rule heads can be *approximated* through the technique known as Skolemization [Chang73]. The idea of Skolemization is that in a formula of the form $\forall\ Y_1...Y_n \exists\ X\ ...\ \phi$ the existential variable `X` can be removed and replaced everywhere in ϕ with the function term $f(Y_1...Y_n)$, where `f` is a *new* function symbol that does not occur anywhere else in the specification. The rationale for such a substitution is that, for any query, the original rule base is unsatisfiable if and only if the transformed rule base is unsatisfiable [Chang73]. This implies that the query to the original rule base can be answered if and only if it can be answered when posed against the Skolemized rule base. However, from the point of view of logical entailment, the Skolemized rule base is stronger than the original one, and this is why we say that Skolemization only *approximates* existential quantification, but is not equivalent to it.

Skolemization is defined for formulas in *prenex normal form*, i.e., formulas where all the quantifiers are collected in a prefix to the formula and apply to the entire formula. A formula that is not in the prenex normal form can be converted to one in the prenex normal form by a series of equivalence transformations [Chang73].

SWSL-Rules supports Skolemization by providing special constants `_#` and `_#1`, `_#2`, `_#3`, and so on. As with other constants in SWSL, these symbols can be used both in argument positions and in the position of a function. For instance, `_#(a,_#,_#2(c,_#2))` is a legal function term.

Each occurrence of the symbol `_#` denotes a new constant. Generation of such a constant is the responsibility of the SWSL-Rules compiler. For instance, in `_#(a,_#,_#2(c,_#2))`, the two occurrences of `_#` denote two different constants that do not appear anywhere else. In the first case, the constant is in the position of a function symbol. The numbered Skolem constants, such as `_#2` in our example, also denote a new constant that does not occur anywhere else in the rule base. However, the different occurrences of the same numbered symbol in *the same rule* denote the *same* new constant. Thus, in the above example the two occurrences of `_#2` denote the same new symbol. Here is a more complete example:

```
holds(a,_#1) and between(1,_#1,5).       between(minusInf, _#(?Y), ?Y)   :-  timepoint(?Y) ?Y != minusInf.
```

In the first line, the two occurrences of `_#1` denote the same new Skolem constant, since they occur in the scope of the same rule. In the second line, the occurrence of `_#` denotes a new Skolem function symbol. Since we used `_#` here, this symbol is distinct from any other constant. Note, however, that even if we used `_#1` in the second rule, that symbol would have denoted a distinct new function symbol, since it occurs in a separate rule and there is no other occurrence of `_#1` in that rule.

The Skolem constants in SWSL-Rules are in some ways analogous to the blank nodes in RDF. However, they have the semantics suitable for a rule-based language and it has been argued in [*Yang03*] that the Skolem semantics is superior to RDF, which relies on existential variables in the rule heads [*Hayes04*].

2.13 SWSL-Rules and XML Schema Data Types

SWSL-Rules supports the primitive XML Schema data types. However, since SWSL-Rules is quite different from XML, it adapts the lexical representation for XML data types to the form that is more suitable for a logic-based language. The translation from the XML lexical representation of primitive data types to SWSL-Rules is straightforward.

The general rule is that each primitive value is represented by a function term whose functor symbol is the name of the primitive data type prefixed with an underscore (_). The arguments of the term represent the various components of the primitive data type. For instance, `_string("abc")`, `_date(2005,7,18)`, `_decimal(123.56)`, `_integer(321)`, `_float(23e5)`, and so on.

The string, decimal, integer, and float data types have a shorthand notation (some of which had been seen before). Thus, `_string("abc")` is abbreviated to `"abc"`, `_decimal(123.56)` to `123.56`, `_integer(321)` to `321`, and `_float(23e5)` to `23e5`.

Other primitive data types are represented using a similar notation. For instance, the duration of 1 year, 2 months, 3 days, 10 hours, and 30 minutes is represented as `_duration(1,2,3,10,30,0)` where the first argument of `_duration` represents years and the last seconds. The same negative duration is represented as `-_duration(1,2,3,10,30,0)`. For another example, the values of the `dateTime` type are represented as `_dateTime(2005,10,29,15,55,40)`.

It is often necessary to exchange values of primitive data types between applications. Since the internal representations of the data types vary from language to language, ***serialization*** into a commonly agreed representation has been used for this purpose. SWSL-Rules supports serialization of primitive data types via the built-in predicate `_serialize`. It takes three arguments: a SWSL-Rules value of a SWSL-Rules data type, a URI that denotes the target of serialization, and a result, which is a string that contains the serialized value. Currently, the only target is `http://www.w3.org/2001/XMLSchema`, which refers to XML Schema 1.0. Other targets will be added as necessary (for example, for XML Schema 1.1 when it is released). Example: `_serialize(_date(2005,1,1),_"http://www.w3.org/2001/XMLSchema",?Result)` binds `?Result` to `"2005-01-01"`.

The predicate `_serialize` is intended to work both ways: for serialization and deserialization. Deserialization occurs when the last argument is bound to a string representation of a data type and the first argument is unbound. For instance, `_serialize(?Result,_"http://www.w3.org/2001/XMLSchema","2005-01-01")` binds `?Result` to `_date(2005,1,1)`.

2.14 Semantics of SWSL-Rules

A single point of reference for the model-theoretic semantics of SWSL-Rules will be given in a separate document. Here we will only give an overview and point to the papers where the semantics of the different layers were defined separately.

First, we note that the semantics of the Lloyd-Topor leyers -- both monotonic and nonmonotonic -- is transformational and was given in Sections 2.4 and 2.6. Similarly, the Courteous layer is defined transformationally and is described in [*Grosof2004a*].

The model theory of NAF is given by the well-founded semantics as described in [*VanGelder91*]. The model theory behind HiLog is described in [*Chen93*] and F-logic is described in [*Kifer95*]. The semantics of inheritance that is used in SWSL-Rules is defined in [*Yang02*]. The model theory of reification is given in [*Yang03*] and was further extended to reification of rules in [*Kifer04*].

The semantics of the Equality layer is based on the standard semantics (for instance, [*Chang73*]) but is modified by the ***unique name assumption***, which states that syntactically distinct terms are unequal. This modification is described in [*Kifer95*], and we summarize it here. First, without equality, SWSL-Rules makes the unique name assumption. With equality, the unique name assumption is modified to say that terms that cannot be proved equal with respect to `:=:` are assumed to be unequal. In other words, SWSL-Rules makes a closed world assumption about explicit equality.

Other than that, the semantics of `:=:` is standard. The interpretation of this predicate is assumed to be an equivalence relation with ***congruence properties***. A layman's term for this is "substitution of equals by equals." This means that if, for example, `t:=:s` is derived for some terms `t` and `s` then, for any formula ϕ, it is true if and only if ψ is true, where ψ is obtained from ϕ by replacing some occurrences of `t` with `s`.

Overall, the semantics of SWSL-Rules has nonmonotonic flavor even without NAF and its extension layers. This is manifested by the use of the unique name assumption (modified appropriately in the presence of equality) and the treatment of constraints. To explain the semantics of constraints, we first need to explain the idea of ***canonic models***.

In classical logic, all models of a set of formulas are created equal and are given equal consideration. Nonmonotonic logics, on the other hand, carefully define a subset of models, which are declared to be ***canonical*** and logical entailment is considered only with respect to this subset of models. Normally, the canonical models are so-called ***minimal models***, but not all minimal models are canonical.

Any rule set that does not use the features of the NAF layer and its extensions is known to have a unique minimal model, which is also its canonical model. This is an extension of the well-known fact for Horn clauses in classical logic programming [*Lloyd87*]. With NAF, a rule set may have multiple incomparable minimal models, and it is well-known that not all of these models appropriately capture the intended meaning of rules. However, it turns out that one such model can be distinguished, and it is called the ***well-founded model*** [*VanGelder91*]. A formula is considered to be true according to the SWSL-Rules semantics if and only if it is true in that one single model, and the formula is false if and only if it is false in that model.

Now, in the presence of constraints, the semantics of SWSL-Rules is defined as follows. Given a rulebase, first its canonical model is determined. In this process, all constraints are *ignored*. Next, the constraints are checked in the canonical model. If *all* of them are true, the rulebase is said to be consistent. If at least one constraint is false in the canonical model, the constraint is said to be violated and the rulebase is said to be inconsistent.

2.15 SWSL-FOL: The First-order Subset of SWSL

The SWSL language includes all the connectives used in first-order logic and, therefore, syntactically first-order logic is a subset of SWSL. When the semantics of first-order connectives differs from their nonmonotonic interpretation, new nonmonotonic connectives are introduced. For instance, first-order negation, `neg`, has a nonmonotonic counterpart `naf` and first-order implications <== and ==> have a nonmonotonic counterpart `:-`.

It follows from the above that SWSL-Rules and SWSL-FOL share significant portions of their syntax. In particular, every connective used in SWSL-FOL can also be used in SWSL-Rules. However, not every first-order formula in SWSL-FOL is a rule and the rules in SWSL-Rules are not first-order formulas (because of "`:-`"). Therefore, neither SWSL-FOL is a subset of SWSL-Rules nor the other way around. Furthermore, even though the classical connectives `neg` and ==>/<== can occur in SWSL-Rules, they are embedded into an overall nonmonotonic language and their semantics cannot be said to be exactly first-order.

Formally, ***SWSL-FOL*** consists of the following formulas:

- First-order atomic formulas
- If ϕ and ψ are SWSL-FOL formulas then so are ϕ `and` ψ, ϕ `or` ψ, `neg` ϕ, ϕ ==> ψ, ϕ <== ψ, and ϕ <==> ψ.
- If ϕ is a SWSL-FOL formula and X is a variable, then the following are also SWSL-FOL formulas: `exist ?X` (ϕ) and `forall ?X` (ϕ). SWSL-FOL allows to combine quantifiers of the same sort, so `exist ?X,?Y` (ϕ) is the same as `exist ?X exist ?Y` (ϕ).

As in the case of SWSL-Rules, we will ise the period (".") to designate the end of a SWSL-FOL formula.

SWSL defines three extensions of SWSL-FOL. The first extension adds the equality operator, `:=:`, the second incorporates the object-oriented syntax from the Frames layer of SWSL-Rules, the third does the same for the HiLog layer.

Formally, ***SWSL-FOL+Equality*** has the same syntax as SWSL-FOL, but, in addition, the following atomic formulas are allowed:

- *term* :=: *term*

SWSL-FOL+Frames has the same syntax as SWSL-FOL except that, in addition, the following is allowed:

- SWSL-Rules molecules, as defined in the Frames layer are valid SWSL-FOL formulas.
- The path expressions defined of the SWSL-Rules Frames syntax are not used in SWSL-FOL. In SWSL-Rules, path expressions are interpreted differently in the rule head and body. Since SWSL-FOL does not distinguish the head of a rule from its body, the path expression syntax is not well-defined in this context.

SWSL-FOL+HiLog extends SWSL-FOL by allowing HiLog terms and HiLog atomic formulas instead of first-order terms and first-order atomic formulas.

Each of these extensions is not only a syntactic extension of SWSL-FOL but also a semantic extension. This means that if ϕ and ψ are formulas in SWSL-FOL then $\phi \models \psi$ in SWSL-FOL if and only if the same holds in SWSL-FOL+Equality, SWSL-FOL+Frames, and SWSL-FOL+HiLog. We will say that SWSL-FOL+Equality, SWSL-FOL+Frames, and SWSL-FOL+HiLog are ***conservative semantic extensions*** of SWSL-FOL.

SWSL-FOL+HiLog and SWSL-FOL+Frames can be combined both syntactically and semantically. The resulting language is a conservative semantic extension of both SWSL-FOL+HiLog and SWSL-FOL+Frames. Similarly, SWSL-FOL+Equality and SWSL-FOL+Frames can be combined and the resulting language is a conservative extension of both. Interestingly, combining SWSL-FOL+Equality with SWSL-FOL+HiLog leads to a conservative extension of SWSL-FOL+HiLog, but *not* of SWSL-FOL+Equality! More precisely, if ϕ and ψ are formulas in SWSL-FOL+Equality and $\phi \models \psi$ then the same holds in SWSL-FOL+HiLog. However, there are formulas such that $\phi \models \psi$ holds in SWSL-FOL+HiLog but not in SWSL-FOL+Equality [*Chen93*].

2.16 Semantics of SWSL-FOL

The semantics of the first-order sublanguage of SWSL is based on the standard first-order model theory and is monotonic. The only new elements here are the higher-order extension that is based on HiLog [*Chen93*] and the frame-based extension based on F-logic [*Kifer95*]. The respective references provide a complete model theory for these extensions, which extends the standard model theory for first-order logic.

2.17 Future Extensions

To enhance the power of the SWSL-Rules language, a number of extensions are being planned, as described below.

- **If-then-else**. The `if` *`test`* `then` *`test1`* is sometimes more convenient and familiar than the ==> operator. More important, however, is the fact that the more complete idiom, `if` *`test`* `then` *`test1`* **`else`** *`test2`*, is known to be very useful and common in rule-based languages. Although the `else`-part can be expressed with negation as failure, this is not natural and most well-developed languages support the *if-then-else* idiom directly. This idiom may be added to SWSL-Rules later.

- **Aggregate operators**. Aggregate operators, such as sum, average, etc., are important database operations. XML languages such as XPath, XSLT, and XQuery all have support for aggregation. Of these, only XQuery permits aggregation over general set comprehension. A future extension of SWSL-Rules will allow aggregation in the style of FLORA-2, which supports explicit set comprehension and nested aggregation. The general syntax of such aggregation is:

  ```
  ?Result = aggregate{?Var [GroupingVarList] | Query }
  ```

  where *aggregate* can be `max`, `min`, `avg`, `count`, `sum`, `collectset`, `collectbag`. The last two aggregates return lists of the instantiations of `?Var` that satisfy *Query* without the duplicates (`collectset`) and with possible duplicates (`collectbag`).

The grouping variables provide the functionality similar to `GROUP BY` of SQL. They have the effect that the aggregation produces one list of results per every instantiation of the variables in *GroupingVarList* for which *Query* has a solution. The variable `?Result` gets successively bound to each such list (one list at a time).

- **Constraints.** Constraints play a very important role in database and knowledge base applications. As a future extension, SWSL-Rules will have database-style constraints. Database constraints are different in nature from restrictions used in Description Logic. Whereas restrictions in Description Logic are part of the same logical theory as the rest of the statements and are used to *derive* new statements, constraints in databases are not used to derive new information. Instead, they serve as tests of correctness for the canonical models of the knowledge base. In this framework, canonical models (e.g., the well-founded model [*VanGelder91*]) are first computed without taking constraints into account. These models are then checked against the constraints. The models that do not satisfy the constraints are discarded. In the case of the well-founded semantics, which always yields a single model, testing satisfaction of the constraints validates whether the knowledge base is in a consistent state.

- **Procedural attachments, state changes à la Transaction Logic, situated logic programs**. A ***procedural attachment*** is a predicate or a method that is implemented by an external procedure (e.g., in Java or Python). Such a procedure can have a side effect on the real world (e.g., sending an email or activating a device) or it can receive information from the outside world. First formalizations of these ideas in the context of database and rule based languages appeared in [*Maier81*, *Chimenti89*]. These ideas were recently explored in [*Grosof2004a*] in the context of e-commerce. Transaction Logic [*Bonner98*] provides a seamless integration of these concept into the logic.

  An attached procedure can be specified by a link statement, which associates a predicate or a method with an external program. The exact details of the syntax have not been finalized, but the following is a possibility:

  ```
  attachment relation/Arity name-of-java-procedure(integer,string,...)
  ```

  This syntax can be generalized to include object-oriented methods.

  Another necessary extension involves *update primitives* - primitives for changing the underlying state of the knowledge. These primitives can add or delete facts, and even add or delete rules. A declarative account of such update operations in the context of a rule-based language is given by Transaction Logic [*Bonner98*]. This logic also can also be used to represent triggers (also known as ECA rules) [*Bonner93*].

- **Predicates with named arguments**. For predicates with *ordered* arguments, named attributes are essentially supported by the current syntax. For instance, if -> is viewed as an infix binary function symbol, then `p(foo -> 1, bar -> 2)` is a valid term in SWSL-Rules. Predicates with *unordered* arguments can make unification exponential and are unlikely to be supported in the future.

- **The "rest"-variables**. The ``rest" notation à la SCL can be useful in metaprogramming. A rest-variable binds to a list of variables or terms and it always occurs as the last variable of a term. During unification with another term, such a variable binds to a list of arguments of that term beginning with argument corresponding to the variable till the rest of the term (whence the name of such variables). For instance, in the following term, `?R` is a rest-variable:

  ```
  p(?X,?Y | ?R)
  ```

  If this term is unified with `p(?Z,f,?Z,q)`, then `?X` binds to `?Z`, `?Y` to `f`, and `?R` to the list `[?Z,q]`.

- **Non-ground identity relation**, ==. This predicate is true if the arguments are identical up to variable renaming. This predicate is not declarative but can be very useful, as demonstrated by the extensive practice of logic programming.

3 Combining SWSL-Rules and SWSL-FOL

SWSL includes two fundamentally distinct knowledge representation languages:

1. SWSL-Rules -- a declarative rule-based language based on the logic programmin/deductive database paradigm; and
2. SWSL-FOL -- a classical first order logic based language

In this section, we discuss how -- and also why -- to **combine** knowledge expressed in SWSL-Rules with knowledge expressed in SWSL-FOL.

First, it is worthwhile to review the motivations for having the two distinct knowledge representation languages.

SWSL-Rules is especially well suited to represent available knowledge and desired patterns of reasoning for several tasks in semantic Web services:

- authorization policies (for security, access control, confidentiality, privacy, and other kinds of trust);
- contracts (partial or complete, proposed or finalized);
- monitoring of processes to recognize and handle exceptions or other dynamic conditions (e.g., monitoring of performance of contracts to detect and respond to violations of contract provisions such as late delivery or non-payment);
- advertising, discovery, and matchmaking (e.g., advertisements and requests for quotation or requests for proposals can be regarded as partial contract proposals);
- semantic mediation, especially translation mappings that mediate between different ontologies or contexts and thus between knowledge expressed in those different ontologies or contexts (e.g., to translate from the output of one service to the input expected by another service); and
- object-oriented ontologies that use default inheritance with priorities and/or cancellation (e.g., in the manner of the Process Handbook [*Process Handbook*] [*Bernstein2003*] [*Grosof2004d*]).

In particular, the capabilities of SWSL-Rules for logical nonmonotonicity (negation-as-failure and/or Courteous prioritized conflict handling) is used heavily in many use case scenarios for each of the above tasks and the associated kinds of knowledge.

SWSL-FOL is especially well suited to represent available knowledge and desired patterns of reasoning for several other tasks in semantic Web services, especially revolving around the process model:

- Composition of services, and associated planning using process models;
- Analysis, verification, and validation of services in terms of their process models; and
- Ontologies expressed in first order classical logic, e.g., in Description Logic (for example, in OWL-DL).

In particular, the capabilities of SWSL-FOL for disjunction, reasoning by cases, contrapositive reasoning, and/or existentials are used heavily in many use case scenarios for each of the above tasks and its associated kinds of knowledge.

SWSL-Rules and SWSL-FOL overlap largely in syntax, and SWSL-Rules includes almost all of the connectives of SWSL-FOL. The deeper issue, however, is the semantic relationship between SWSL-Rules and SWSL-FOL.

For several purposes it is desirable to *combine* knowledge expressed in the SWSL-Rules form with knowledge expressed in the SWSL-FOL form. One important such purpose is:

- LP rules "on top of" FOL ontologies. "On top of" here means that some of the predicates mentioned in the set of rules are defined via ontological knowledge expressed in FOL. Such FOL ontologies can often be viewed as "background" knowledge.

For example, the predicates might be classes or properties defined via OWL-DL axioms, i.e., expressed in the Description Logic fragment of FOL.

In terms of semantics, it is desirable to have reasoning in SWSL-Rules *respect* as much as possible the information contained in such background FOL ontologies. In particular, it is desirable to enable sufficient completeness in the semantic combination to ensure that the conclusions drawn in SWSL-Rules will be (classically) *not inconsistent* with the SWSL-FOL ontologies.

Ideally, there would be one well-understood overall knowledge representation formalism that subsumes both SWSL-Rules and SWSL-FOL. This would provide the general theoretical basis for combining arbitrary SWSL-Rules knowledge with arbitrary SWSL-FOL knowledge. Unfortunately, finding such an umbrella formalism is still an open issue for basic research. Instead, the current scientific understanding provides only a limited theoretical basis for combining SWSL-Rules knowledge with SWSL-FOL knowledge. On the bright side, there are limited expressive cases for which it is well-understood theoretically how to do such combination.

The Venn diagram of relationships between the different formalisms, given in Figure 2.1 illustrates the most salient aspects of the current scientific understanding.

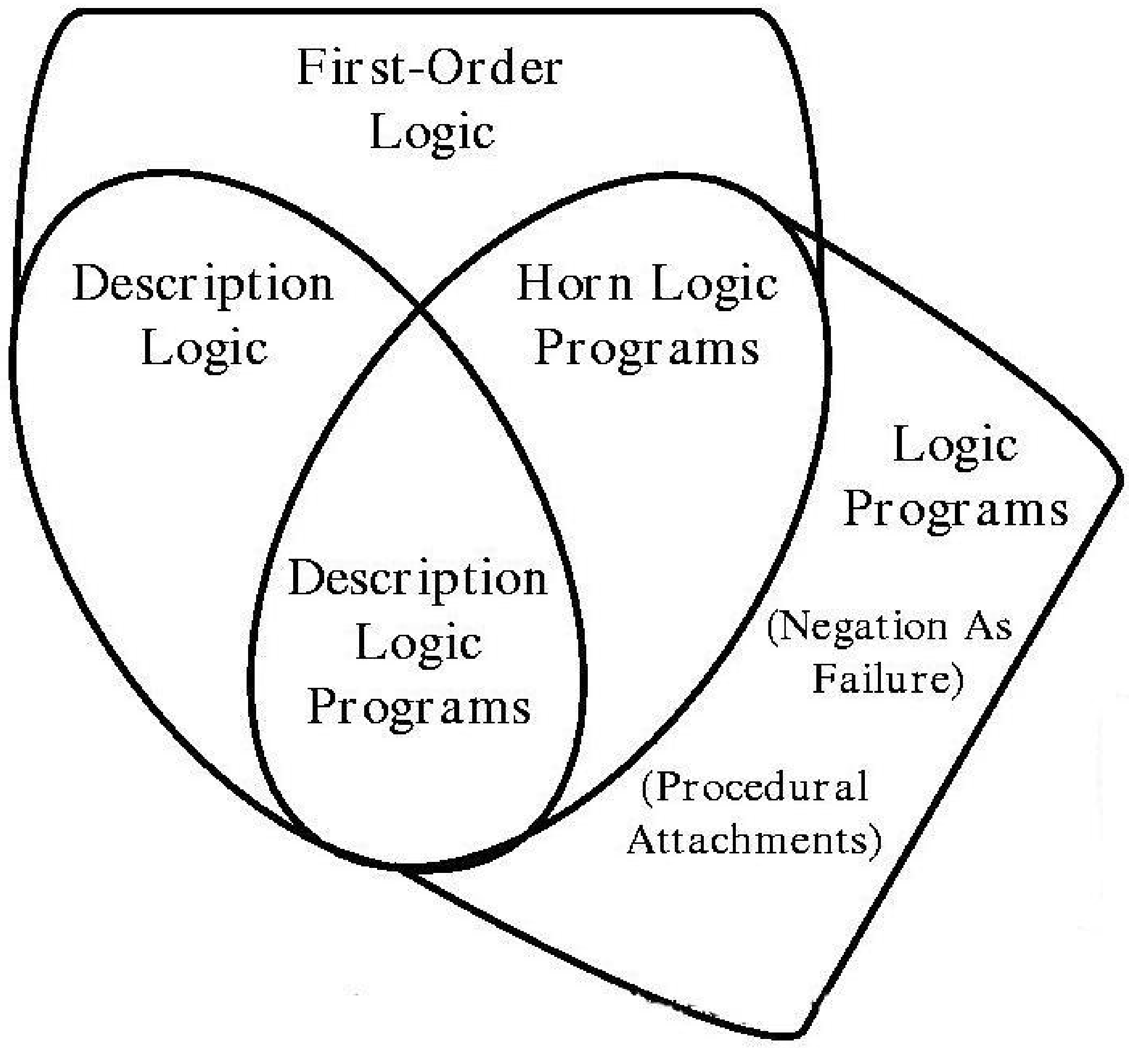

Figure 2.1: The relationships among different formalisms

The shield shape represents first-order logic-based formalisms. The (diagonally-rotated) bread-slice shape shows the expressivity of the logic programming based paradigms. These overlap partially -- in the Horn rules subset. FOL includes expressiveness beyond the overlap, notably: positive disjunctions; existentials; and entailment of non-ground and non-atomic conclusions. Likewise, LP includes expressiveness beyond the overlap, such as negation-as-failure, which is logically nonmonotonic. Description Logic (cf. OWL-DL), depicted as an oval shape, is a fragment of FOL.

Horn FOL is another fragment of FOL. Horn LP is a slight weakening of Horn FOL. "Weakening" here means that the conclusions from a given set of Horn premises that are entailed according to the Horn LP formalism are a subset of the conclusions entailed (from that same set of premises) according to the Horn FOL formalism. However, the set of ground atomic conclusions is the same in the Horn LP as in the Horn FOL. For most practical purposes (e.g., relational database query answering), Horn LP is thus essentially similar in its power to the Horn FOL.

Horn LP is a fragment of both FOL and nonmonotonic LP -- i.e., of both SWSL-Rules and SWSL-FOL. Horn LP is thus a limited "bridge" that provides a way to pass information -- either premises, or ground-atomic conclusions -- from FOL to LP, or vice versa. Knowledge from FOL that is in the Horn LP subset of expressiveness can be easily combined with general LP knowledge. Vice versa, knowledge from LP that is in the Horn LP subset of expressiveness can be easily combined with general FOL knowledge. Description Logic Programs (DLP) [*Grosof2003a*] represent a fragment of Horn LP. It likewise acts as a "bridge" between Description Logic (i.e., OWL-DL) and LP.

Note that, technically, LP uses a different logical connective for implication (":-" in SWSL syntax) than FOL uses. When we speak of Horn LP as a fragment of FOL, we are viewing this LP implication connective as mapped into the FOL implication connective (also known as *material implication*).

Horn LP as "bridge". To summarize, there is some initial good news about semantic combination:

- The Horn LP case is a "bridge" between SWSL-Rules and SWSL-FOL.
- The DLP case is a "bridge" between SWSL-Rules and OWL-DL.

Builtin predicates. Another case of well behaved semantic combination is for ***builtin predicates*** that are purely informational, e.g., that represent arithmetic comparisons or operations such as less-than or multiplication. Technically, in LP these can be viewed as procedural attachments. But alternatively, they can be viewed as predicates that have fixed extensions. Their semantics in both FOL and LP can thus be viewed essentially as virtual knowledge base consisting of a set of ground facts. This thus falls into the Horn LP fragment.

Hypermonotonic reasoning as "bridge". Recently, a new theoretical approach called ***hypermonotonic reasoning*** [*Grosof2004c*] has been developed to enable a case of "bridging" between (nonmon) LP and FOL that is considerably more expressive than Horn LP.

We will now describe in more detail some preliminary results about this hypermonotonic reasoning approach that bear upon the relationship of LP to FOL and thus upon how to combine LP knowledge with FOL knowledge.

Courteous LP (including its fragment: LP with negation-as-failure) can be viewed as a weakening of FOL, under a simple mapping of Courteous LP rules/conclusions into FOL. "Weakening" here means that for a given set of premises, the set of conclusions entailed in the Courteous LP formalism is in general a subset of the set of conclusions entailed by the FOL formalism. In other words:

- *(Courteous) LP is sound but incomplete relative to FOL.*

This fundamental relationship between the formalisms provides an augmentation to the theoretical basis for combining knowledge in LP (i.e., SWSL-Rules) with knowledge in FOL.

Consider a set of rules S in LP and a set of formulas B in FOL. Let T be a translation mapping from the language of S to the language of B. S is said to be ***hypermonotonic*** with respect to B and T when S is sound but incomplete relative to B, under the mapping T. That is, when the conclusions entailed in S from a given set of premises P are in general always a subset of the conclusions entailed in B from the translated premises of S.

Define CLP2 to be the fragment of the Courteous LP formalism in which explicit negation-as-failure is omitted (i.e., prohibited). Each rule and mutex in CLP2 can be mapped quite straightforwardly and intuitively to a clause in FOL: simply replace the LP implication connective (":-" in SWSL-Rules syntax) by the FOL implication connective. Observe that this is the same mapping/translation that was considered in relating the Horn LP to FOL. Each ground-literal conclusion in CLP2 can also be mapped, in the same fashion, into a ground-literal in FOL.

The restriction on Courteous LP to avoid explicit negation-as-failure is not very onerous essentially, since the great majority of use cases in which explicit negation-as-failure is employed can be reformulated during manual authoring of rules so as to avoid it as a construct. More generally, the mapping can be extended, by complicating it a bit, to permit explicit negation-as-failure.

Going in the reverse direction, every clause in FOL can also be mapped into CLP2, in such a way that the resulting CLP information is a weakening of the FOL clause that nevertheless preserves much of the strength of the FOL clause. This reverse-translation mapping from FOL to CLP is complicated somewhat by the *directional* nature of the LP implication connective. "Directional" here means having a direction from body towards head. Each LP rule can be viewed as a directed clause. Consider a FOL clause *C* that consists of a disjunction of *m* literals:

- (universal closure of:) L1 or ... or Lm.

Here, each Li is an atom or a classically-negated atom. When mapping *c* to CLP2, there are *m* possible choices of one for each possible choice of which literal is to be made head of the LP rule. Each possible choice corresponds to a different rule -- the LP rule in which literal Li is chosen as head has the form:

- `Li :- neg L1, ..., neg Li-1, neg Li+1, ..., neg Lm .`

Altogether, the FOL clause *C* is mapped into a set of *m* LP rules:

- `L1 :- neg L2, neg L3, ..., neg Lm .`
- `L2 :- neg L1, neg L3, ..., neg Lm .`
- ...
- `Lm :- neg L1, neg L2, ..., neg Lm-1 .`

where neg (neg A) is replaced equivalently by A. This set of rules is called the "omni-directional" set of rules for that clause -- or, more briefly, the *"omni rules"* for that clause.

In general, FOL axioms need not be clausal since they may include existential quantifiers. However, often *skolemization* can be performed to represent such existentials in a manner that preserves soundness (as is usual for skolemization). A refinement of the reverse translation mapping above is to exploit such skolemization in order to relax the requirement of clausal form. We use such skolemization particularly for head existentials.

Automatic weakened translation of FOL ontologies into SWSL-Rules. In the ontologies aspect of SWSL, it is desirable to have a "bridging" technique to automatically translate FOL ontologies into SWSL-Rules in such a manner as to preserve soundness (from an FOL viewpoint) but to be nevertheless fairly strong (i.e., capture much of the strength/content of the original FOL axioms). We have adopted, as an experimental "bridging" approach, the reverse translation mapping technique described above in order to map FOL ontologies into SWSL-Rules (heavily using the Courteous feature). In particular, we have applied this technique to map the axioms of PSL Core and Outer Core into SWSL-Rules so as to create a weakened version of that ontology that can be utilized within SWSL-Rules. Because some of these PSL axioms include existentials, we utilize the skolemization refinement described above, particularly for head existentials. The mapping from the PSL axioms to SWSL-Rules is given in appendix PSL in SWSL-FOL and SWSL-Rules.

The precise algorithm used to obtain the SWSL-Rules translation for a given axiom in SWSL-FOL is as follows:

```
Input: a formula F in SWSL-FOL.
Output: a set of rules R, expressed in SWSL-Rules.

1) Translate F into formula F1 in Prenex Normal Form.

2) Skolemize F1 to get F2, which is in Skolem Normal Form.

3) Write F2 as a set S of clauses.

4) For each clause C in S, produce the omnidirectional set of rules for C (as defined above).

R then is the union of all the omnidirectional sets of rules produced by (4).
```

4 Serialization of SWSL in RuleML

SWSL is serialized in XML using RuleML. RuleML-style serialization of SWSL enables interoperation with other XML applications for rules and provides an encoding for transporting SWSL-Rules via the SOAP infrastructure of Web services.

RuleML integrates various rule paradigms via common set of concepts and defines a family of rule-based, Web-enabled sublanguages with various degrees of expressiveness. This section applies the RuleML approach to serialization of SWSL. This is done mostly by reusing and sometimes extending the existing RuleML sublanguages. In addition, a new sublanguage for the serialization of HiLog is developed.

Serialization of the presentation syntax of SWSL-Rules amounts to construction of explicit parse trees and then representing these trees linearly as XML markup that is compliant with XML Schema of the appropriate RuleML sublanguages. Starting with Version 0.89, the XML Schema specification of RuleML supports SWSL-Rules.

Serialization of SWSL-FOL does not require any new constructs, and it is done by repurposing existing RuleML features. Serialization of SWSL-FOL is discussed at the end of this section, in Section 4.5.

Conceptually, RuleML models XML trees as objects and thus divides all XML tags into class descriptors, called *type tags*, and property descriptors, called *role tags*. This conceptual object-oriented model implies that type tags and role tags must alternate, which is known as *striped XML syntax*. For instance, in F-logic and RDF, classes can have properties, which point to classes, which have properties that point to classes, etc. Similarly, in the striped XML syntax, a type tag has role tags as subelements, whose children are again type tags, etc. When the role of a subelement is clear from the context, its tag may be skipped for brevity, as in RDF's StripeSkipping.

4.1 Serialization of the HiLog Layer

HiLog terms. The HiLog serialization uses the *type tag* `Hterm` for HiLog terms, `Con` for constants, and `Var` for variables. Since HiLog allows arbitrary terms to be used in the position of predicate and function symbols, the RuleML serialization allows not only constants but also variables and Hterms under the `op` *role tag* . The following illustrates the main aspects of the HiLog serialization.

- ***Regular first-order terms***. For instance, the HiLog terms `c`, `f(a,?X)`, `?X` are represented by the following three XML fragments, respectively:

```
<Con>c</Con>


<Hterm>
   <op><Con>f</Con></op>
   <Con>a</Con>
   <Var>X</Var>
</Hterm>


<Var>X</Var>
```

- ***Variables over function symbols***. For instance, the terms `?X(a,?Y)`, `?X(a,?Y(?X))` are serialized as follows:

```
<Hterm>
   <op><Var>X</Var></op>
   <Con>a</Con>
   <Var>Y</Var>
</Hterm>
```

```
<Hterm>
   <op><Var>X</Var></op>
   <Con>a</Con>
   <Hterm>
      <op><Var>Y</Var></op>
      <Var>X</Var>
   </Hterm>
</Hterm>
```

- ***Parameterized function symbols***. For instance, the HiLog terms `f(?X,a)(b,?X(c))`, `?Z(?X,a)(b,?X(?Y)(d))`, `?Z(f)(g,a)(p,?X)` will be serialized as shown below:

```
<Hterm>
   <op>
      <Hterm>
         <op><Con>f</Con></op>
         <Var>X</Var>
         <Con>a</Con>
      </Hterm>
   </op>
   <Con>b</Con>
   <Hterm>
      <op><Var>X</Var></op>
      <Con>c</Con>
   </Hterm>
</Hterm>
```

```
<Hterm>
   <op>
      <Hterm>
         <op><Var>Z</Var></op>
         <Var>X</Var>
         <Con>a</Con>
      </Hterm>
   </op>
   <Con>b</Con>
   <Hterm>
      <op>
         <Hterm>
            <op><Var>X</Var></op>
            <Var>Y</Var>
         </Hterm>
      </op>
      <Con>d</Con>
   </Hterm>
</Hterm>
```

```
<Hterm>
   <op>
      <Hterm>
         <op>
            <Hterm>
               <op><Var>Z</Var></op>
               <Con>f</Con>
            </Hterm>
         </op>
         <Con>g</Con>
         <Con>a</Con>
      </Hterm>
   </op>
   <Con>p</Con>
   <Var>X</Var>
</Hterm>
```

HiLog atomic formulas. Since any HiLog term is also a HiLog atomic formula, the RuleML serialization for these formulas is the same as for HiLog terms. The following example shows an encoding of a query, which uses the `Query` element of RuleML:

```
?- q(?X) and ?X.
```

```
<Query>
   <And>
      <Hterm>
         <op><Con>q</Con></op>
         <Var>X</Var>
      </Hterm>
      <Var>X</Var>
   </And>
<Query>
```

Another interesting example is a HiLog rule

```
call(?X) :- ?X.
```

which is a logical definition of the meta-predicate `call/1` in Prolog. This is translated using the RuleML tags `Implies`, `head`, and `body`, as follows:

```
<Implies>
   <head>
      <Hterm>
         <op><Con>call</Con></op>
         <Var>X</Var>
      </Hterm>
   </head>
   <body>
      <Var>X</Var>
   </body>
</Implies>
```

4.2 Serialization of Explicit Equality

The explicit equality predicate `:=:` is serialized using the RuleML's element `Equal`. For example,

```
f(a,?X):=:g(?Y,b) :- p(?X,?Y).
```

is serialized as

```
<Implies>
   <head>
      <Equal>
         <Hterm>
            <op><Con>f</Con></op>
            <Con>a</Con>
            <Var>X</Var>
         </Hterm>
         <Hterm>
            <op><Con>g</Con></op>
            <Var>Y</Var>
            <Con>b</Con>
         </Hterm>
      </Equal>
   </head>
   <body>
      <Hterm>
         <op><Con>p</Con></op>
         <Var>X</Var>
         <Var>Y</Var>
      </Hterm>
   </body>
</Implies>
```

4.3 Serialization of the Frames Layer

To serialize the Frames layer of SWSL-Rules we need to show the serialization of the various molecules and path expressions introduced by F-logic.

Molecules. The serialization of molecules uses slotted atoms, which have an `oid` but often do not have an `op`. The overall structure of F-logic molecules (except for class membership and subclassing) is as follows:

```
Atom ::= oid op? slot*
```

- ***Value molecules***. If `t`, `m`, `v` are terms then the value molecule `t[m -> v]` is serialized as follows:

  ```
  <Atom><oid>t'</oid><slot>m' v'</slot></Atom>
  ```

  Here and elsewhere we use primes to represent recursive RuleML serialization. For instance, `t'`, `m'`, and `v'` denote RuleML serializations of `t`, `m`, and `v`, respectively. For instance, `o[f(a,b) -> 3]` would be represented by the following fragment:

  ```
  <Atom>
     <oid><Con>o</Con></oid>
     <slot>
        <Hterm><Con>f</Con><Con>a</Con><Con>b</Con></Hterm>
        <Con>3</Con>
     </slot>
  </Atom>
  ```

The syntax `t[m -> {v1,...,vk}]` is also supported: the set-valued result is serialized using the `Set` tag:

```
<Atom>
   <oid>t'</oid>
   <slot>
      m'
      <Set>v1',...,vk'</Set>
   </slot>
</Atom>
```

- ***Boolean molecules***. These molecules have the form `t[m]` where `t` and `m` are terms. They are serialized using singleton `slot` elements. For instance, `mary[female]` is represented as follows:

```
<Atom>
   <oid><Con>mary</Con></oid>
   <slot>
      <Con>female</Con>
   </slot>
</Atom>
```

- ***Class membership molecule***. Class membership molecules of the form `t:s` are serialized using the `InstanceOf` element:

```
<InstanceOf>t' s'</InstanceOf>
```

where `t'` and `s'` represent RuleML serializations of `t` and `s`.

- ***Subclass molecule***. The subclass molecules of the form `t::s` are represented using the `SubclassOf` element as follows:

```
<SubclassOf>t' s'</SubclassOf>
```

As before, `t'` and `s'` represent RuleML serializations of `t` and `s`, respectively.

- ***Signature molecule***. Signature molecules of the form `t[m => v]` are represented using the `Signature` element, where the prime represents RuleML serialization of the corresponding term:

```
<Signature><oid>t'</oid><slot>m' v'</slot></Signature>
```

- ***Boolean signature molecules***. A Boolean signature molecule has the form `t[=>m]`. Its RuleML serialization uses singleton `slot` elements within a `Signature` element:

```
<Signature><oid>t'</oid><slot>m'</slot></Signature>
```

- ***Cardinality constraints***. Signature molecules can have associated cardinality constraints. Such molecules have the form

`t[s(t1,...,tn) {min : max} => v]`

In RuleML this becomes:

```
<Signature>
   <oid>t'</oid>
   <slot mincard="min" maxcard="max">
      <Hterm><Con>s'</Con>t1'...tn'</Hterm>
      v'
   </slot>
</Signature>
```

Nested molecules. Direct serialization of nested molecules is not currently supported. Instead, they must first be broken into conjunctions of non-nested molecules and then serialized.

Slot access and path expressions. Serialization of slot access uses the RuleML `Get` primitive. Serialization of path expressions is supported via the polyadic RuleML `SlotProd` element. For example, `room.ceiling.color` becomes the following:

```
<Get>
   <oid><Con>room</Con></oid>
   <SlotProd><Con>ceiling</Con><Con>color</Con></SlotProd>
</Get>
```

4.4 Serialization of Reification

SWSL-Rules supports reification of F-logic molecules, formulas that can occur in the head or the body of a rule, and of the rules themselves. The only restriction is that explicit quantifiers cannot occur under the scope of the reification operator. The main idea behind RuleML serialization of such statements is that the corresponding serialized statement must be embedded within a `Reify` element.

To illustrate, consider the following molecule:

```
a[b -> ${p(X[foo -> bar])}]
```

Since the reified statement (`p(X[foo -> bar])`) is an `Hterm`, this tag becomes the child of the `Reify` element.

```
<Hterm>
   <oid><Con>a</Con></oid>
   <slot>
      <Con>b</Con>
      <Reify>
         <Hterm>
            <op><Con>p</Con></op>
            <Hterm>
               <oid><Var>X</Var></oid>
               <slot><Con>foo</Con><Con>bar</Con></slot>
            </Hterm>
         </Hterm>
      </Reify>
   </slot>
</Hterm>
```

The following example illustrates serialization of a reified rule.

```
john[believes -> ${p(?X) implies q(?X)}].
```

The corresponding serialization is shown below. Since the top-level tag of a rule is `Implies`, this tag becomes the child of the `Reify` element.

```
<Hterm>
   <oid>john</oid>
   <slot>
      <Con>believes</Con>
      <Reify>
         <Implies>
            <body>
               <Hterm>
                  <op><Con>p</Con></op>
                  <Var>X</Var>
               </Hterm>
            </body>
            <head>
               <Hterm>
                  <op><Con>q</Con></op>
                  <Var>X</Var>
               </Hterm>
            </head>
         </Implies>
      </Reify>
   </slot>
</Hterm>
```

4.5 Serialization of SWSL-FOL

Serialization of SWSL-FOL reuses the existing FOL RuleML sublanguage. The serialization is accomplished through the following rules:

- First-order atomic formulas are serialized as `<Atom>...</Atom>`. What actually goes between the `Atom` tags depends on the type of the atomic formula (i.e., whether it is a predicate formula or a F-logic frame).
- If φ and ψ are serialized SWSL-FOL formulas then so are `<And>` φ ψ `</And>`, `<Or>` φ ψ `</Or>`, `<Neg>` φ `</Neg>`, `<Implies>` φ ψ `</Implies>` (SWSL-FOL's φ `impliedBy` ψ must be first replaced with ψ `implies` φ), and `<Equivalent>` φ ψ `</Equivalent>` (for SWSL-FOL's `iff`).
- If `X` is a variable and φ is a serialized SWSL-FOL formula, then the following are also SWSL-FOL serialized formulas: `<Exists>` `<Var>X</Var>` φ `</Exists>` and `<Forall>` `<Var>X</Var>` φ `</Forall>`. Serialized SWSL-FOL also allows to combine quantifiers of the same sort and reduce repetitiveness. For instance, `<Exists>` `<Var>X</Var>` `<Var>Y</Var>` φ `</Exists>` is a shorthand for `<Exists>` `<Var>X</Var>` `<Exists>` `<Var>Y</Var>` φ `</Exists>` `</Exists>`.

5 Glossary

Activity

Activity. In the formal PSL ontology, the notion of activity is a basic construct, which corresponds intuitively to a kind of (manufacturing or processing) activity. In PSL, an activity may have associated *occurrences*, which correspond intuitively to individual instances or executions of the activity. (We note that in PSL an activity is not a class or type with occurrences as members; rather, an activity is an object, and occurrences are related to this object by the binary predicate `occurrence_of`.) The occurrences of an activity may impact fluents (which provide an abstract representation of the "real world"). In FLOWS, with each service there is an associated activity (called the "service activity" of that service). The service activity may specify aspects of the internal process flow of the service, and also aspects of the messaging interface of that service to other services.

Channel
: *Channel*. In FLOWS, a channel is a formal conceptual object, which corresponds intuitively to a repository and conduit for messages. The FLOWS notion of channel is quite primitive, and under various restrictions can be used to model the form of channel or message-passing as found in web services standards, including WSDL, BPEL, WS-Choreography, WSMO, and also as found in several research investigations, including process algrebras.

FLOWS
: ***F**irst-order **L**ogic **O**ntology for **W**eb **S**ervices*. FLOWS, also known as SWSO-FOL, is the first-order logic version of the Semantic Web Services Ontology. FLOWS is an extension of the PSL-OuterCore ontology, to incorporate the fundamental aspects of (web and other electronic) services, including service descriptors, the service activity, and the service grounding.

Fluent
: *Fluent*. In FLOWS, following PSL and the situation calculii, a fluent is a first-order logic term or predicate whose value may vary over time. In a first-order model of a FLOWS theory, this being a model of PSL-OuterCore, time is represented as a discrete linear sequence of *timees*, and fluents has a value for each time in this sequence.

Grounding
: *Grounding*. The SWSO concepts for describing service activities, and the instantiations of these concepts that describe a particular service activity, are *abstract* specifications, in the sense that they do not specify the details of particular message formats, transport protocols, and network addresses by which a Web service is accessed. The role of the *grounding* is to provide these more concrete details. A substantial portion of the grounding can be acheived by mapping SWSO concepts into corresponding WSDL constructs. (Additional grounding, e.g., of some process-related aspects of SWSO, might be acheived using other standards, such as BPEL.)

Message
: *Message*. In FLOWS, a message is a formal conceptual object, which corresponds intuitively to a single message that is created by a service occurrence, and read by zero or more service occurrences. The FLOWS notion of message is quite primitive, and under various restrictions can be used to model the form of messages as found in web services standards, including WSDL (1.0 and 2.0), BPEL, WS-Choreography, WSMO, and also as found in several research investigations. A message has a *payload*, which corresponds intuitively to the body or contents of the message. In FLOWS emphasis is placed on the knowledge that is gained by a service occurrence when reading a message with a given payload (and the knowledge needed to create that message.

Occurrence
: *Occurence (of a service)*. In FLOWS, a service *S* has an associated FLOWS activity *A* (which generalizes the notion of PSL activity). An *occurrence* of *S* is formally a PSL occurrence of the activity *A*. Intuitively, this occurrence corresponds to an instance or execution (from start to finish) of the activity *A*, i.e., of the process associated with service *S*. As in PSL, an occurrence has a starting time time and an ending time.

PSL
: *Process Specification Language*. The Process Specification Language (PSL) is a formally axiomatized ontology [*Gruninger03a*, *Gruninger03b*] that has been standardized as ISO 18629. PSL provides a layered, extensible ontology for specifying properties of processes. The most basic PSL constructs are embodied in PSL-Core; and PSL-OuterCore incorporates several extensions of PSL-Core that includes several useful constructs. (An overview of concepts in PSL that are relevant to FLOWS is given in Section 6 of the Semantic Web Services Ontology document.)

QName
: *Qualified name*. A pair (*URI*, *local-name*). The *URI* represents a namespace and *local-name* represents a name used in an XML document, such as a tag name or an attribute name. In XML, QNames are syntactically represented as *prefix*:*local-name*, where *prefix* is a macro that expands into a concrete URI. See Namespaces in XML for more details.

ROWS
: ***R**ules **O**ntology for **W**eb **S**ervices*. ROWS, also known as SWSO-Rules, is the rules-based version of the Semantic Web Services Ontology. ROWS is created by a relatively straight-forward, almost faithful, transformation of FLOWS, the First-order Logic Ontology for Web Services. As with FLOWS, ROWS incorporates fundamental aspects of (web and other electronic) services, including service descriptors, the service activity, and the service grounding. ROWS enables a rules-based specification of a family of services, including both the underlying ontology and the domain-specific aspects.

Service
: *(Formal) Service*. In FLOWS, a service is a conceptual object, that corresponds intuitively to a web service (or other electronically accessible service). Through binary predicates a service is associated with various service descriptors (a.k.a. non-functional properties) such as Service Name, Service Author, Service URL, etc.; an *activity* (in the sense of PSL) which specifies intuitively the process model associated with the service; and a *grounding*.

Service contract
: Describes an agreement between the service requester and service provider, detailing requirements on a service occurrence or family of service occurrences.

Service descriptor
: *Service Descriptor*. This is one of several non-functional properties associated with services. The Service Descriptors include Service Name, Service Author, Service Contract Information, Service Contributor, Service Description, Service URL, Service Identifier, Service Version, Service Release Date, Service Language, Service Trust, Service Subject, Service Reliability, and Service Cost.

Service offer description
: Describes an abstract service (i.e. not a concrete instance of the service) provided by a service provider agent.

Service requirement description
: Describes an abstract service required by a service requester agent, in the context of service discovery, service brokering, or negotiation.

sQName
: *Serialized QName*. A serialized QName is a shorthand representation of a URI. It is a macro that expands into a full-blown URI. sQNames are *not* QNames: the former are URIs, while the latter are pairs (*URI*, *local-name*). Serialized QNames were originally introduced in RDF as a notation for shortening URI representation. Unfortunately, RDF introduced confusion by adopting the term QName for something that is different from QNames used in XML. To add to the confusion, RDF uses the syntax for sQNames that is identical to XML's syntax for QNames. SWSL distinguishes between QNames and sQNames, and uses the syntax *prefix#local-name* for the latter. Such an sQName expands into a full URI by concatenating the value of *prefix* with *local-name*.

URI
: *Universal Resource Identifier*. A symbol used to locate resources on the Web. URIs are defined by IETF. See Uniform Resource Identifiers (URI): Generic Syntax for more details. Within the IETF standards the notion of URI is an extension and refinement of the notions of Uniform Resource Locator (URL) and Relative Uniform Resource Locators.

6 References

[Berardi03]
Automatic composition of e-services that export their behavior. D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. In *Proc. 1st Int. Conf. on Service Oriented Computing (ICSOC)*, volume 2910 of *LNCS*, pages 43--58, 2003.

[Bernstein2000]
How can cooperative work tools support dynamic group processes? Bridging the specificity frontier. A. Bernstein. In *Proc. Computer Supported Cooperative Work (CSCW'2000)*, 2000.

[Bernstein2002]
Towards High-Precision Service Retrieval. A. Bernstein, and M. Klein. In *Proc. of the first International Semantic Web Conference (ISWC'2002)*, 2002.

[Bernstein2003]

Beyond Monotonic Inheritance: Towards Semantic Web Process Ontologies. A. Bernstein and B.N. Grosof (NB: authorship sequence is alphabetic). Working Paper, Aug. 2003. Available at: http://ebusiness.mit.edu/bgrosof/#beyond-mon-inh-basic.

[Bonner93]
Database Programming in Transaction Logic. A.J. Bonner, M. Kifer, M. Consens. *Proceedings of the 4-th Intl.~Workshop on Database Programming Languages*, C. Beeri, A. Ohori and D.E. Shasha (eds.), 1993. In Springer-Verlag Workshops in Computing Series, Feb. 1994: 309-337.

[Bonner98]
A Logic for Programming Database Transactions. A.J. Bonner, M. Kifer. Logics for Databases and Information Systems, J. Chomicki and G. Saake (eds.). Kluwer Academic Publishers, 1998: 117-166.

[Bruijn05]
Web Service Modeling Ontology (WSMO). J. de Bruijn, C. Bussler, J. Domingue, D. Fensel, M. Hepp, M. Kifer, B. König-Ries, J. Kopecky, R. Lara, E. Oren, A. Polleres, J. Scicluna, M. Stollberg. *DERI Technical Report.*

[BPML 1.0]
A. Arkin. Business Process Modeling Language. BPMI.org, 2002

[BPEL 1.1]
Business Process Execution Language for Web Services, Version 1.1 . S. Thatte, editor. OASIS Standards Specification, May 5, 2003.

[Bultan03]
Conversation specification: A new approach to design and analysis of e-service composition. T. Bultan, X. Fu, R. Hull, and J. Su. In *Proc. Int. World Wide Web Conf. (WWW)*, May 2003.

[Chang73]
Symbolic Logic and Mechanical Theorem Proving. C.L. Chang and R.C.T. Lee. Academic Press, 1973.

[Chen93]
HiLog: A Foundation for Higher-Order Logic Programming. W. Chen, M. Kifer, D.S. Warren. Journal of Logic Programming, 15:3, February 1993, 187-230.

[Chimenti89]
Towards an Open Architecture for LDL. D. Chimeti, R. Gamboa, R. Krishnamurthy, VLDB Conference, 1989: 195-203.

[deGiacomo00]
ConGolog, A Concurrent Programming Language Based on the Situation Calculus. G. de Giacomo, Y. Lesperance, and H. Levesque. Artificial Intelligence, 121(1--2):109--169, 2000.

[Fu04]
WSAT: A Tool for Formal Analysis of Web Services. X. Fu, T. Bultan, and J. Su. *16th International Conference on Computer Aided Verification (CAV)*, July 2004.

[Frohn94]
Access to Objects by Path Expressions and Rules. J. Frohn, G. Lausen, H. Uphoff. Intl. Conference on Very Large Databases, 1994, pp. 273-284.

[Gosling96]
*The Java language specification.*Gosling, James, Bill Joy, and Guy L. Steele. 1996. Reading, Mass.: Addison-Wesley.

[Grosof99a]
A Courteous Compiler From Generalized Courteous Logic Programs To Ordinary Logic Programs. B.N. Grosof. IBM Report included as part of documentation in the IBM CommonRules 1.0 software toolkit and documentation, released on http://alphaworks.ibm.com. July 1999. Also available at: http://ebusiness.mit.edu/bgrosof/#gclp-rr-99k.

[Grosof99b]
A Declarative Approach to Business Rules in Contracts. B.N. Grosof, J.K. Labrou, and H.Y. Chan. Proceedings of the 1st ACM Conference on Electronic Commerce (EC-99). Also available at: http://ebusiness.mit.edu/bgrosof/#econtracts+rules-ec99.

[Grosof99c]
DIPLOMAT: Compiling Prioritized Default Rules Into Ordinary Logic Programs (Extended Abstract of Intelligent Systems Demonstration). B.N. Grosof. IBM Research Report RC 21473, May 1999. Extended version of 2-page refereed conference paper appearing in Proceedings of the National Conference on Artificial Intelligence (AAAI-99), 1999. Also available at: http://ebusiness.mit.edu/bgrosof/#cr-ec-demo-rr-99b.

[Grosof2003a]
Description Logic Programs: Combining Logic Programs with Description Logic. B.N. Grosof, I. Horrocks, R. Volz, and S. Decker. Proceedings of the 12th International Conference on the World Wide Web (WWW-2003). Also available at: http://ebusiness.mit.edu/bgrosof/#dlp-www2003.

[Grosof2004a]
Representing E-Commerce Rules Via Situated Courteous Logic Programs in RuleML. B.N. Grosof. Electronic Commerce Research and Applications, 3:1, 2004, 2-20. Preprint version is also available at: http://ebusiness.mit.edu/bgrosof/#.

[Grosof2004b]
SweetRules: Tools for Semantic Web Rules and Ontologies, including Translation, Inferencing, Analysis, and Authoring. B.N. Grosof, M. Dean, S. Ganjugunte, S. Tabet, C. Neogy, and D. Kolas. http://sweetrules.projects.semwebcentral.org. Software toolkit and documentation. Version 2.0, Dec. 2004.

[Grosof2004c]
Hypermonotonic Reasoning: Unifying Nonmonotonic Logic Programs with First Order Logic. B.N. Grosof. http://ebusiness.mit.edu/bgrosof/

#HypermonFromPPSWR04InvitedTalk. Slides from Invited Talk at Workshop on Principles and Practice of Semantic Web Reasoning (PPWSR04), Sep. 2004; revised Oct. 2004. Paper in preparation.

[Grosof2004d]
SweetPH: Using the Process Handbook for Semantic Web Services. B.N. Grosof and A. Bernstein. http://ebusiness.mit.edu/bgrosof/#SweetPHSWSLF2F1204Talk. Slides from Presentation at SWSL Meeting, Dec. 9-10, 2004. *Note: Updates the design in the 2003 Working Paper "Beyond Monotonic Inheritance: Towards Semantic Web Process Ontologies" and describes implementation.*

[Grosof2004e]
SweetDeal: Representing Agent Contracts with Exceptions using Semantic Web Rules, Ontologies, and Process Descriptions. B.N. Grosof and T.C. Poon. International Journal of Electronic Commerce (IJEC), 8(4):61-98, Summer 2004 Also available at: http://ebusiness.mit.edu/bgrosof/#sweetdeal-exceptions-ijec.

[Grosof2004f]
Semantic Web Rules with Ontologies, and their E-Business Applications. B.N. Grosof and M. Dean. Slides of Conference Tutorial (3.5-hour) at the 3rd International Semantic Web Conference (ISWC-2004). Available at: http://ebusiness.mit.edu/bgrosof/#ISWC2004RulesTutorial.

[Gruninger03a]
A Guide to the Ontology of the Process Specification Language. M. Gruninger. *Handbook on Ontologies in Information Systems.* R. Studer and S. Staab (eds.). Springer Verlag, 2003.

[Gruninger03b]
Process Specification Language: Principles and Applications. M. Gruninger and C. Menzel. *AI Magazine,* 24:63-74, 2003.

[Gruninger03c]
Applications of PSL to Semantic Web Services. M. Gruninger. *Workshop on Semantic Web and Databases. Very Large Databases Conference, Berlin.*

[Hayes04]
RDF Model Theory. Hayes, P. W3C, February 2004.

[Helland05]
Data on the Outside Versus Data on the Inside. P. Helland. *Proc. 2005 Conf. on Innovative Database Research (CIDR)*, January, 2005.

[Hull03]
E-Services: A Look Behind the Curtain. R. Hull, M. Benedikt, V. Christophides, J. Su. *Proc. of the ACM Symp. on Principles of Database Systems (PODS)*, San Diego, June, 2003.

[Kifer95]
Logical Foundations of Object-Oriented and Frame-Based Languages, M. Kifer, G. Lausen, J. Wu. Journal of ACM, 1995, 42, 741-843.

[Kifer04]
A Logical Framework for Web Service Discovery, M. Kifer, R. Lara, A. Polleres, C. Zhao. Semantic Web Services Workshop, November 2004, Hiroshima, Japan.

[Klein00a]
Towards a Systematic Repository of Knowledge About Managing Collaborative Design Conflicts. Klein, Mark. 2000. Proceedings of the Conference on Artificial Intelligence in Design. Boston, MA, USA.

[Klein00b]
A Knowledge-Based Approach to Handling Exceptions in Workflow Systems. Klein, Mark, and C. Dellarocas. 2000. Computer Supported Cooperative Work: The Journal of Collaborative Computing 9:399-412.

[Lloyd87]
Foundations of logic programming (second, extended edition). J. W. Lloyd. Springer series in symbolic computation. Springer-Verlag, New York, 1987.

[Lindenstrauss97]
Automatic Termination Analysis of Logic Programs. N. Lindenstrauss and Y. Sagiv. International Conference on Logic Programming (ICLP), 1997.

[Maier81]
Incorporating Computed Relations in Relational Databases. D. Maier, D.S. Warren. SIGMOD Conference, 1981: 176-187.

[Malone99]
Tools for inventing organizations: Toward a handbook of organizational processes. T. W. Malone, K. Crowston, J. Lee, B. Pentland, C. Dellarocas, G. Wyner, J. Quimby, C. Osborne, A. Bernstein, G. Herman, M. Klein, E. O'Donnell. *Management Science*, 45(3), pages 425--443, 1999.

[McIlraith01]
Semantic Web Services. *IEEE Intelligent Systems*, Special Issue on the Semantic Web, S. McIlraith, T.Son and H. Zeng. 16(2):46--53, March/April, 2001.

[Milner99]
Communicating and Mobile Systems: The π-Calculus. R. Milner. Cambridge University Press, 1999.

[Narayanan02]
Simulation, Verification and Automated Composition of Web Services. S. Narayanan and S. McIlraith. In *Proceedings of the Eleventh International World Wide Web Conference (WWW-11)*, May, 2002.

[Ontobroker]
Ontobroker 3.8. Ontoprise, GmbH.

[OWL Reference]
OWL Web Ontology Language 1.0 Reference. Mike Dean, Dan Connolly, Frank van Harmelen, James Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. W3C Working Draft 12 November 2002. Latest version is available at http://www.w3.org/TR/owl-ref/.

[OWL-S 1.1]
OWL-S: Semantic Markup for Web Services. David Martin, editor. Technical Overview (associated with OWL-S Release 1.1).

[Papazoglou03]
Service-Oriented Computing: Concepts, Characteristics and Directions. M.P. Papazoglou. Keynote for the 4th International Conference on Web Information Systems Engineering (WISE 2003), December 10-12, 2003.

[Perlis85]
Languages with Self-Reference I: Foundations. D. Perlis. Artificial Intelligence, 25, 1985, 301-322.

[Preist04]
A Conceptual Architecture for Semantic Web Services, C. Preist, 1993. In Proceedings of Third International Semantic Web Conference, Nov. 2004: 395-409.

[Reiter01]
Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems. Raymond Reiter. MIT Press. 2001

[Scherl03]
Knowledge, Action, and the Frame Problem. R. B. Scherl and H. J. Levesque. *Artificial Intelligence*, Vol. 144, 2003, pp. 1-39.

[Singh04]
Protocols for Processes: Programming in the Large for Open Systems. M. P. Singh, A. K. Chopra, N. V. Desai, and A. U. Mallya. *Proc. of the 19th Annual ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Vancouver, October 2004.

[SWSL Requirements]
Semantic Web Services Language Requirements. B. Grosof, M. Gruninger, et al, editors. White paper of the Semantic Web Services Language Committee.

[UDDI v3.02]
Universal Description, Discovery and Integration (UDDI) protocol. S. Thatte, editor. OASIS Standards Specification, February 2005.

[VanGelder91]
The Well-Founded Semantics for General Logic Programs. A. Van Gelder, K.A. Ross, J.S. Schlipf. Journal of ACM, 38:3, 1991, 620-650.

[WSCL 1.0]
Web Services Conversation Language (WSCL) 1.0. A. Banerji et al. W3C Note, March 14, 2002.

[WSDL 1.1]
Web Services Description Language (WSDL) 1.1. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. W3C Note, March 15, 2001.

[WSDL 2.0]
Web Services Description Language (WSDL) 2.0 -- Part 1: Core Language. R. Chinnici, M. Gudgin, J.-J. Moreau, J. Schlimmer, and S. Weerawarana. W3C Working Draft, August 3, 2004.

[WSDL 2.0 Primer]
Web Services Description Language (WSDL) Version 2.0 -- Part 0: Primer. D. Booth, C. Liu, editors. W3C Working Draft, 21 December 2004.

[WS-Choreography]
Web Services Choreography Description Language Version 1.0. N. Kavantzas, D. Burdett, et. al., editors. W3C Working Draft, December 17, 2004.

[XSLT]
XSL Transformations (XSLT) Version 1.0. J. Clark, editor. W3C Recommendation, 16 November 1999.

[XQuery 1.0]
XQuery 1.0: An XML Query Language. S. Boag, D. Chamberlin, et al, editors. W3C Working Draft 04 April 2005.

[Yang02]
Well-Founded Optimism: Inheritance in Frame-Based Knowledge Bases. G. Yang, M. Kifer. Intl. Conference on Ontologies, DataBases, and Applications of Semantics for Large Scale Information Systems (ODBASE), October 2002.

[Yang03]
Reasoning about Anonymous Resources and Meta Statements on the Semantic Web. G. Yang, M. Kifer. Journal on Data Semantics, Lecture Notes in Computer Science 2800, Springer Verlag, September 2003, 69-98.

[Yang04]
FLORA-2 User's Manual. G. Yang, M. Kifer, C. Zhao, V. Chowdhary. 2004.

Semantic Web Services Ontology (SWSO)

Version 1.0

Authors:
Steve Battle (Hewlett Packard) Abraham Bernstein (University of Zurich) Harold Boley (National Research Council of Canada) Benjamin Grosof (Massachusetts Institute of Technology) Michael Gruninger (NIST) Richard Hull (Bell Labs Research, Lucent Technologies) Michael Kifer (State University of New York at Stony Brook) David Martin (SRI International) Sheila McIlraith (University of Toronto) Deborah McGuinness (Stanford University) Jianwen Su (University of California, Santa Barbara) Said Tabet (The RuleML Initiative)

Abstract

This document defines the Semantic Web Services Ontology (SWSO). The ontology is expressed in two forms: FLOWS, the First-order Logic Ontology for Web Services; and ROWS, the Rules Ontology for Web Services, produced by a systematic translation of FLOWS axioms into the SWSL-Rules language. FLOWS has been specified in SWSL-FOL, the first-order logic language developed by SWSO's sister SWSL effort.

Status of this document

This is one of four documents that make up the initial report of the Semantic Web Services Language Committee of the Semantic Web Services Initiative. The report defines the Semantic Web Services Framework (SWSF).

History of publication at http://www.daml.org/services/swsl/report/swso/:

- v. 0.9: April 6, 2005
- v. 0.91: April 13, 2005
- v. 0.92: April 25, 2005
- v. 0.93: May 5, 2005

History of publication at http://www.daml.org/services/swsf/swso/:

- v. 1.0: May 9, 2005

Table of contents

This document also refers to Section 2 in the Application Scenarios document.

1 Introduction

This document is part of the technical report of the Semantic Web Services Language (SWSL) Committee of the Semantic Web Services Initiative (SWSI).

The overall structure of the report is described in the document titled Semantic Web Services Framework Overview.

This document presents the Semantic Web Services Ontology (SWSO). This includes a description of the conceptual model underlying the ontology, and a description of a first-order logic (FOL) axiomatization which defines the model-theoretic semantics of the ontology. The axiomatization is called "SWSO-FOL" or equivalently, FLOWS -- the **F**irst-order **L**ogic **O**ntology for **W**eb **S**ervices -- and is expressed using SWSL-FOL. (The full axiomatization is available in Appendix B.) As noted in Section 5, These same axioms can be expressed (or partially expressed in some cases) in SWSL-Rules. The SWSL-Rules formalization of the axioms, called SWSO-Rules or ROWS, is listed in Appendix C.

The goal of FLOWS is to enable reasoning about the semantics underlying Web (and other eletronic) services, and how they interact with each other and with the "real world". FLOWS does not strive for a complete representation of web services, but rather for an abstract model that is faithful to the semantic aspects of service behavior. (An approach to "grounding" FLOWS specifications into real Web services, e.g., that use WSDL for messaging, is discussed in Section 4). Following the lead of the situation calculii [*Reiter01*], and in particular the situation calculus semantics [*Narayanan02*] of OWL-S [*OWL-S 1.1*], the changing portions of the real world are modeled abstractly using the notion of *fluents*. These are first-order logic predicates and terms that can change value over time. The FLOWS model provides infrastructure for representing messages between services; the focus here is on the semantic content of a message, rather than, for example, the specifics of how that content is packaged into an XML-based message payload. (Again, Grounding will address these issues.) FLOWS also provides constructs for modeling the internal processing of Web services.

FLOWS is intended to enable reasoning about essential aspects of Web service behavior, for a variety of different purposes and contexts. Some targeted purposes are to support (a) descriptions of Web services that enable automated discovery, composition, and verification, and (b) creation of declarative descriptions of a Web service, that can be mapped (either automatically or through a systematic process) to executable specifications. A variety of contexts can be supported, including: (i) modelling a service as essentially a black box, where only the messaging is observable; (ii) modelling the internal atomic processes that a service performs, along with the impact these processes have on the real world (e.g., inventories, financial accounts, commitments); (iii) modelling many properties of a service, including all message APIs, all or some of the internal processing, and some or all of the internal process and data flows. Of course, the usability of a Web service description may depend on how much or how little information is included.

It is important to note that the specification of the FLOWS ontology in first-order logic does not presuppose that the automated reasoning tasks described above will be realized using a first-order logic theorem prover. We can certainly use FOL to specify solutions to these tasks, using notions of entailment and satisfiability. Nevertheless, while some tasks may naturally be realized through theorem proving, it has been our experience that most AI automated reasoning tasks are addressed by special-purpose reasoners, rather than by general-purpose reasoners such as theorem provers. For example, we anticipate specialized programs being developed to perform Web service discovery, Web service composition and Web service verification. Thus, the role of FLOWS is not necessarily to provide an executable specification, but rather to provide an ontology -- a definition of concepts -- with a well-defined model theoretical semantics that will enable specification of these tasks and that will support the development of specialized reasoners that may be provably correct with respect to these specifications.

FLOWS captures the salient, functional elements of various models of Web services found in the literature and in industrial standards. In FLOWS, an emphasis is placed on enabling the formal representation and study of the following formalisms, among others.

- Models focused on specifying semantic Web services [*McIlraith01*] including the OWL-S [*OWL-S 1.1*] model of atomic processes, their inputs, outputs, preconditions and effects, and the associated notion of impacts "on the world" and testing conditions "about the world"; and the WSMO [*Bruijn05*] ontology for describing the intended goals of Web services.
- Various process models for combining atomic services, including the OWL-S process model and in particular the FOL situation calculus sematics and Petri Net semantics for OWL-S [*Narayanan02*], automata and process algebra based models (e.g., pi-calculus [*Milner99*], BPEL [*BPEL 1.1*], Roman model [*Berardi03*], Conversation model [*Bultan03*], Guarded finite-state automata [*Fu04*]).
- The fundamental emphasis of standards such as WSDL [*WSDL 1.1*], BPEL, WSCL [*WSCL 1.0*], WS-Choreography [*WS-Choreography*], on the sending of "messages" between "web services"; and more broadly, of Services Oriented Architectures [*Papazoglou03*, *Helland05*])
- Various models of interaction between Web services (e.g., WS-Choreography, WSCL, Protocols such as OWL-P [*Singh02*] and the relationship of such "global" protocols with the "local" behaviors they might imply (e.g., results from the conversation model [*Bultan03*]).

FLOWS represents an attempt to extend on the work of OWL-S, to incorporate a variety of capabilities not within the OWL-S goals. OWL-S was focussed on providing an ontology of Web services that would facilitate automated discovery, enactment and composition of Web services. It was not focused on interoperating with or providing a semantics for (at its outset, nonexistent) industry process modeling formalisms such as BPEL. As such, OWL-S concepts such as messages are abstracted away and only reintroduced in tis grounding to WSDL. A primary difference between FLOWS and OWL-S is the expressive power of the underlying language. FLOWS is based on first-order logic, which means that it can express considerably more than can be expressed using OWL-DL (upon which OWL-S is based). (For example, this immediately includes *n*-ary predicates and richly quantified sentences.) The use of first-order logic enables a more refined approach than possible in OWL-S to representing different forms of data flow that can arise in Web services. A second difference is that FLOWS strives to explicitly model more aspects of Web services than OWL-S. This includes primarily the fact that FLOWS can readily model (i) process models using a variety of different paradigms; and (ii) data flow between services, which is acheived either through message passing or access to shared fluents (which might also be thought of as shared data). Indeed, FLOWS provides the ability to model, albeit in an abstract, semantically motivated manner, several key Web services standards, including WSDL, BPEL, and WS-Choreography.

A key premise of FLOWS is that an appropriate foundation for formally describing Web services can be built as a family of *PSL extensions*. PSL -- the Process Specification Language -- is a formally axiomatized ontology [*Gruninger03a*, *Gruninger03b*] that has been standardized as ISO 18629. PSL was originally developed to enable sharing of descriptions of manufacturing processes. FLOWS refines aspects of PSL with Web service-specific concepts and extensions. An overview of concepts in the Process Specification Language that are relevant to FLOWS is given in Section 6 below.

A primary goal of FLOWS is to provide a formal basis for accurately specifying "application domains" based broadly on the paradigms of Web services and/or services oriented architecture (SOA). While FLOWS captures key fundamental aspects of the Web services and SOA paradigms, it also strives to remain very flexible, to allow for the multitude of variations already present in these domains and likely to arise in coming years. Another goal of FLOWS is to enable formalization of a diversity of process modeling formalisms within the ontology. As such, FLOWS, like PSL is designed to facilitate interoperation between diverse process modeling formalisms.

In a typical usage of FLOWS, an application domain is created by combining the FLOWS axioms with additional logical sentences to form a (first order logic) theory. Speaking loosely, each sentence in such a theory can be viewed as a *constraint* or restriction on the models (in the sense of mathematical logic) that satisfy the theory. In particular, and following the spirit of Golog [*Reiter01*] and similar languages, even process model constructs such as `while` or `if-then-else` correspond formally to constraints rather than procedural elements. Similarly, the passing of information, which is often represented using variable assignment in Web services, is specified in the formal model in epistemic terms, following the approach taken in [*Scherl03*]. Again, the approach taken here is intended to provide maximal flexibility in terms of formally modeling the wide range of existing and future web services and SOA models. For example, it is possible ot look at data flow and information sharing at an abstract level based on what a service "knows", or to specialize this to models that involve imperative variable assignment.

Following the high-level structure of OWL-S, FLOWS has three major components: Service Descriptors, Process Model, and Grounding. In the formal ontology, these three elements are associated with services by representing a formal service as a conceptual object, and using relations to associate specific artifacts with the service. For example, `service-author` is a binary relation that associates an author to a service, and `service-activity` is a binary relation that associates a unique (PSL) "complex activity" to a service; the complex activity provides a partial or complete specification of the process model associated with the service. (See Section 3) below for more detail.)

Most domain ontology developers are motivated to provide a logical description of their Web service with a view to some specific set of automations tasks, including perhaps Web service discovery, invocation, composition, verification or monitoring. Each task requires some subset or *view* of the entire FLOWS ontology. For example, to perform automated Web service discovery, we might need axioms from both the Service Descriptors, and select axioms from the Process Model. For Web service invocation, we might only need the inputs and outputs of atomic processes comprising a service in FLOWS-Core, and so on. To cross cut our three broad subontologies (Service Descriptors, Process Model, and Grounding) and to create ontologies that are task-specific, it is possible to project views onto the FLOWS ontology. The concept of views is not developed in this document, however.

This document is organized as follows. Service Descriptors are described in Section 2. The Process Model portion of the ontology, described in Section 3, represents the most substantial advance over OWL-S. This section is quite extensive, presenting the core ontology (FLOWS-Core), several formal extensions of it, and a more informal discussion concerning the relationship of FLOWS to other service models. The section also includes an intuitive description of the conceptual model underlying FLOWS, and links to a formal axiomatization of FLOWS (in Appendix B). (Also relevant is Appendix A, which shows how to specify key elements of PSL in both SWSL-FOL and SWSL-Rules). Section 4 discusses grounding for FLOWS. As noted above, (most of) FLOWS can be systematically translated into the SWSL-Rules language; this is discussed in Section 5. (An axiomatization for ROWS is presented in Appendix C). Section 6 provides a brief review of PSL and the approach of [*Scherl03*] for incorporating knowledge into the situation calculus, for readers who have not been exposed to these before. The document closes with a Glossary (Section 7), and References (Section 8).

The description of FLOWS presented in this document also refers to Section 2 of the document Application Scenarios, that forms part of this W3C submission. That section provides some extended examples which illustrate key aspects of the conceptual model underlying FLOWS.

2 Service Descriptors

Service descriptors provide basic information about a Web service. This descriptive information may include non-functional meta-information and/or provenance information. Service descriptors are often used to support automated Web service discovery. What follows is an initial set of domain-independent properties. Our intention is that these be expanded as needs dictate. Indeed many of the service descriptors that are most effective for service discovery are domain specific.

The initial list captures basic identifying information including name, authorship, textual description, version, etc. The set of descriptors also includes information describing (potentially subjective) notions of quality of service such as reliability. Note that this is an initial set of properties derived predominantly from the OWL-S Profile [*OWL-S 1.1*]). The set is expected to be extended with properties that are domain specific such as those contained in the OWL-S appendices and those in WSMO's [*Bruijn05*] coverage service descriptor (or "non-functional property" as it is called in their proposed specification). The set may also be expanded with properties commonly used in other standards such as those found in Dublin Core. In the following list of properties, those properties derived from OWL-S are so noted. Some of these properties are similar or identical to those included in WSMO's list of non-functional properties.

Following the description of each individual property is a listing of the FLOWS relation that captures the concept within the ontology.

1. *Service Name*: This refers to the name of the service and may be used as a unique identifier. This corresponds directly to the OWL-S property serviceName. In the future, we may find additional name-related properties to have value such as alternateName (including other synonymous names) and presentationName (including a string that would be presented in user interfaces).

   ```
   name(service,service_name)
   ```

2. *Service Author*: Services could have single or multiple authors. Authors could be people or organizations. This property augments the OWL-S contactInformation property.

   ```
   author(service,service_author)
   ```

3. *Service Contact Information*: This property contains a pointer for people or agents requiring more information about the service. While it could be anything, it is likely to be an email address.

   ```
   contact(service,contact_info)
   ```

4. *Service Contributor*: This property describes the entity that is responsible for making updates to the service. Note that while it is related to the author, it may not be identical to the author since an original author may not be the identified contributor or maintainer of a service. Note also, that while it is related to the contactInformation property, it is not identical since the author, contributor, and contact point may all be distinct entities.

   ```
   contributor(service,service_contributor)
   ```

5. *Service Description*: Services may have a textual description. Typical descriptions would include a summary of what the service offers and any requirements that the service has. This property is simply a renaming of the OWL-S textDescription property.

   ```
   description(service,service_description)
   ```

6. *Service URL*: This property is filled with one (or more) URL's associated with the service. This property is included in order to facilitate interoperation where agents need to count on URLs as values.

   ```
   url(service,service_URL)
   ```

7. *Service Identifier*: This property is filled by an unambiguous reference to the service.

   ```
   identifier(service,service_identifier)
   ```

8. *Service Version*: As with all software programs, there may be multiple versions of services and this property contains an identifier for the specific service at a specific time. This is likely to be a revision number in a source code control system.

   ```
   version(service,service_version)
   ```

9. *Service Release Date*: This property contains a date of the release of the service. It would be anticipated that it would be compatible with Dublin Core's date. It may be used, This is a date of the service so that, for example, to search for services not functions would be able to find services that are not more than 2 years old.

   ```
   releaseDate(service,service_release_date)
   ```

10. *Service Language*: This property contains an identifier for the language of the service. This property is expected to contain an identifier for a logical programming language and could also contain an ISO language tag.

    ```
    language(service,service_language)
    ```

11. *Service Trust*: This property contains an entity describing the trustworthiness of the service.

    ```
    trust(service,service_trust)
    ```

12. *Service Subject*: This property contains an entity indicating the topic of the service.

    ```
    subject(service,service_subject)
    ```

13. *Service Reliability*: This property contains an entity used to indicate the dependability of the service. Different measurements may be useful in determining the fillers such as the number of failures in a given time period.

    ```
    reliability(service,service_reliability)
    ```

14. *Service Cost*: This property contains an entity used to indicate a cost of the service. While it could be a monetary unit, it could also be a complex description of charging plans.

    ```
    cost(service,service_cost)
    ```

The translation of service descriptors into ROWS can be found here.

Link to ROWS Axiomatization

Readers may also be interested in the discussion of inputs, (conditional) outputs, preconditions and conditional effects of the complex activity that describes the process model of a service. This is discussed towards the end of Subsection 3.1.2.

3 Process Model

Since PSL is a generic ontology for processes, we need to specify extensions to provide concepts that are useful in the context of Web services. In particular, the FLOWS process model adds two fundamental elements to PSL, namely (i) the structured notion of atomic process as found in OWL-S, and (ii) infrastructure for specifying various forms of data flow. (The FLOWS process model relies on constructs already available in PSL for representing basic aspects of process flow, and also provides some extensions for them.)

Following the modular organization of the PSL ontology, the FLOWS process model is layered and partitioned into natural groupings. When specifying an application domain, it is thus easy to incorporate only the ontological building blocks that are needed.

The FLOWS process model is created as a family of extensions of PSL-OuterCore, The fundamental extension of PSL-OuterCore for services is called "FLOWS-Core". As will be seen, this extension is quite minimalist, and provides an abstract representation only for (web) services, their impact "on the world", and the transmission of messages between them.

Figure 3.1 shows various families of process modeling constructs, including PSL-OuterCore, FLOWS-Core, and several others. These include models from standards, such as the process model portion of BPEL (and a refinement of BPEL that incorporates typical assumptions made when using BPEL in practice), and models from the research literature, such as guarded automata. A line between a family *F* of constructs and a family *F'* above *F* indicates that it is natural to view *F'* as a family that includes most or all of the constructs in *F*. The OWL-S process model is not shown explicitly in Figure 3.1, because it has been integral in the design of the FLOWS-Core and the Control Constructs extension. Of course, the different families of modeling constructs shown in the figure should not be viewed as comprehensive.

In some cases, it may be useful to create PSL extensions of FLOWS-Core, to formally specify the properties of certain families of constructs shown in Figure 3.1. Indeed, as part of the Process Model of the FLOWS ontology presented below several extensions are specified; these are indicated in the figure by the rectangles above FLOWS-Core. We note that FLOWS-Core can serve as a mathematical foundation for representing several other aspects and models from the Web services and SOA standards and literature, and it seems likely that additional PSL extensions of FLOWS-Core will be created in the future.

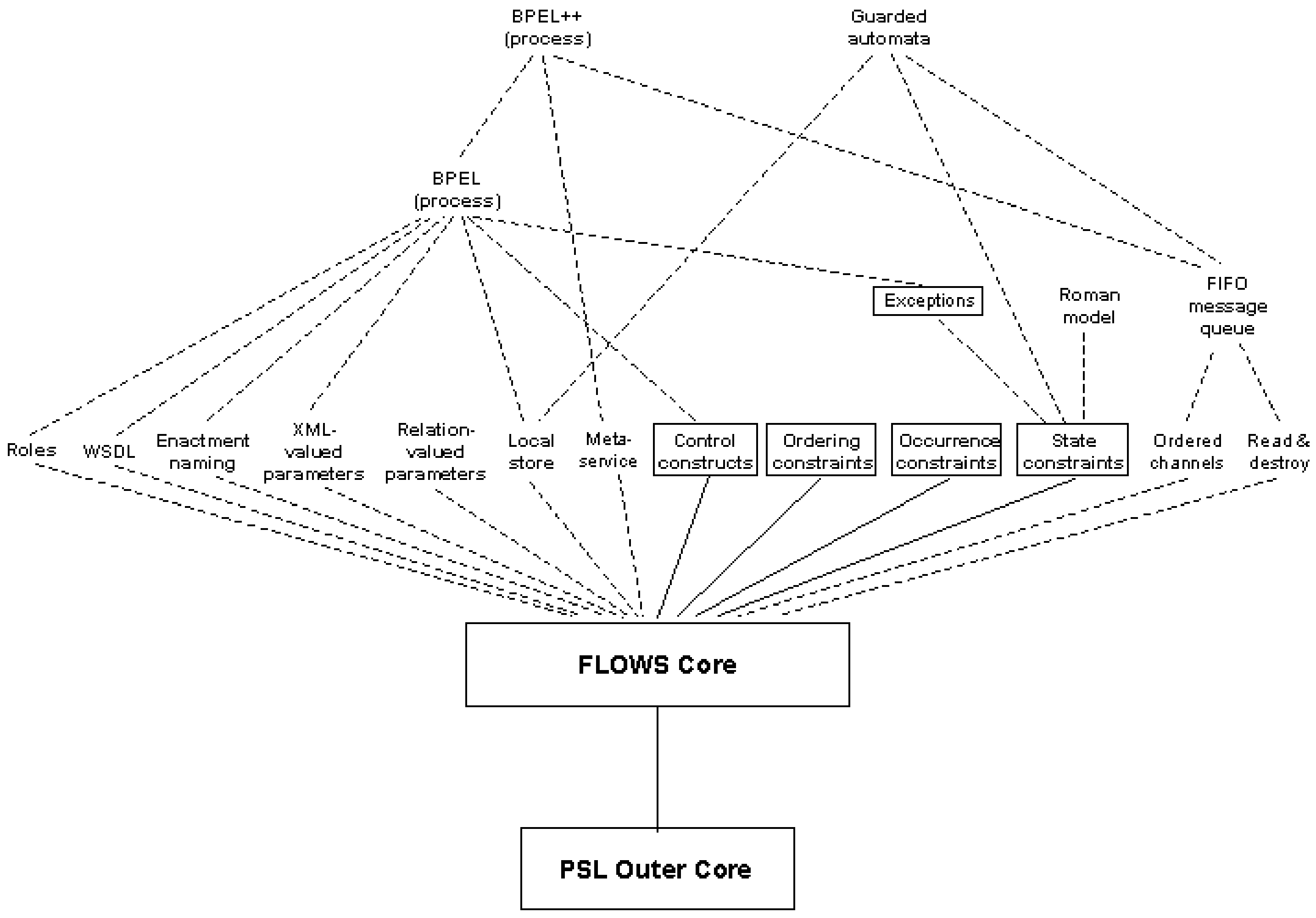

Figure 3.1: Selected families of process modeling constructs relevant to Web services

Following this methodology, the FLOWS process model ontology is composed of a core set of axioms, referred to as FLOWS-Core, and a set of extension ontologies. FLOWS currently consists of six ontology modules that specify core intuitions about the activities associated with a service together with classes of composite activities that are used to express different constraints on the occurrence of services and their subactivities.

- ***FLOWS-Core*** introduces the basic notions of services as activities composed of atomic activities (Subsection 3.1).
- ***Control Constraints*** axiomatize the basic constructs common to workflow-style process models. In particular, the Control Constraints in FLOWS include the concepts from the process model of OWL-S (Subsection 3.2).
- ***Ordering Constraint*** allow the specification of activities defined by sequencing properties of atomic processes (Subsection 3.3).
- ***Occurrence Constraints*** support the specification of nondeterministic activities within services (Subsection 3.4).
- ***State Constraints*** support the specification of activities whose activities are triggered by states (of an overall system) that satisfy a given condition (Subsection 3.5).
- ***Exception Constraints*** provide some basic infrastructure for modeling exceptions (Subsection 3.6).

The table below lists the FLOWS ontologies and the concepts defined in those ontologies.

The FLOWS Process Model Ontology Modules

Module	Major Concepts
FLOWS-Core	Service AtomicProcess composedOf message channel
Control Constraints	Split Sequence Unordered Choice IfThenElse Iterate RepeatUntil
Ordering Constraints	OrderedActivity
Occurrence Constraints	OccActivity
State Constraints	TriggeredActivity
Exception Constraints	Exception

The specification of concepts in FLOWS is presented using some or all of the following:

- *Intended Semantics* Each concept is given an intuitive definition in English.
- *Axiomatization* The intended semantics of concepts in FLOWS is formally axiomatized in SWSL-FOL using the ontology of ISO 18629 (Process Specification Language) as the foundation.
- *Reference Grammar* Process descriptions are often domain-specific. As such, they are not part of the FLOWS ontology. Nevertheless, FLOWS imposes a syntactic form on these SWSL-FOL domain-specific axioms. These are specified using a Reference Grammar.
- *Process Description Syntax* A user-friendly syntax for domain-specific process descriptions is provided. This syntax serves as a macro language that expands into the SWSL-FOL sentences specified in the Reference Grammar.
- *Examples* Where appropriate, we illustrate concepts with a small example. Further examples may be found in the Applications document.

After the presentation of FLOWS-Core and its formal extensions, Subsection 3.7 describes some possible additional extensions of FLOWS-Core, corresponding to the entries in Figure 3.1 not enclosed in rectangles.

3.1 FLOWS-Core

The FLOWS-Core process model is intended to provide a formal basis for specifying essentially any process model of Web services and service composition that has arisen or might arise in the future. As such, the underlying conceptual process model focuses on the most essential aspects of Web services and their interaction.

The principles guiding the FLOWS-Core process model can be summarized as follows:

1. In typical usage of the FLOWS-Core (which follows the spirit of PSL-OuterCore), one creates an *application domain theory* for a given application domain, which incorporates the predicates, terms, and axioms of FLOWS-Core, along with additional domain-specific predicates, terms and constraints.
2. Predicates and terms in the underlying first-order logic language of an application domain theory are used as an abstraction of the "world". Following PSL (and the situation calculi), ***fluents*** are predicates and terms whose values may change as the result of activity occurrences. We generally use the term ***relation*** to refer to a predicate whose value does not change as the result of an activiy occurrence. We informally distinguish between two categories of relations and fluents: ***domain-specific***, which focus on the "real world", or more specifically, on information about the application domain; and ***service-specific***, which focus on the infrastructure used to support the Web services model, in particular on messages and message transmission.
3. As noted above, a formal service is modeled as a conceptual object, and for each service *S* there is exactly one associated (PSL-OuterCore) complex activity *A*. (These are related by the predicate `service-activity(?s,?a)`.) As in PSL, the complex activity *A* may have one or more *occurrences*, each of which corresponds intuitively to a possible execution of *A* (subject to the constraints in the relevant application domain theory). The notion of 'occurrence' of activity *A* here is essentially the same as the notion of '*enactment*' of a workflow specification as found in the workflow literature.) Speaking intuitively, in an application domain theory, the complex acivity *A* might be specified completely, or only partially. We use the phrase ***service activity*** to refer to a PSL complex activity, such as *A*, that is associated to a formal service in some application domain.

4. We model the atomic actions involved in service activities as discrete "occurrences" of "atomic activities" (in the sense of PSL-OuterCore). This includes (a) the activities that are "world-changing", in that they modify facts "in the world" (e.g., plane reservations, commitments to ship products, aspects of financial transfer, modification to inventory databases) and also (b) activities that support aspects of transferring messages between Web services, and (c) activities that create, destroy, or modify channels (see below).
5. Following the spirit of OWL-S, a primary focus in FLOWS-Core is on what are called here (***FLOWS***) ***atomic processes***; these are atomic activities that have input parameters, output parameters, pre-conditions and conditional effects. The pre-conditions of atomic processes will refer to both fluents and time-invariant relations. The conditional effects of atomic processes can have impact on fluents. In the interest of flexibility, FLOWS-Core does not require that every atomic activity in a service activity be a FLOWS atomic process, nor does FLOWS-Core require that every FLOWS atomic process be a subactivity of a service activity.
6. The flow of information *between Web services* can occur in essentially two ways: (a) via message passing and (b) via shared access to the same "real world" fluent (e.g., an inventory database, a reservations database). With regards to message passing, FLOWS-Core models ***messages*** as (conceptual) objects that are created and (possibly) destroyed, and that their life-span has a non-zero duration. This follows the spirit of standards such as WSDL and BPEL, the spirit of at least some works on Services Oriented Architecture [*Helland05*], and indeed the physical reality involved in Web service communication. Messages have types, which indicate the kind of information that they can transmit.
7. FLOWS-Core includes the ***channel*** construct, which provides a convenient mechanism for giving some structure to how messages are transmitted between services. Intuitively, a channel holds messages that have been "sent" and may or may not have been "received". (The notion of 'channel' is inspired by, but not identical to, the notion of 'channel' found in many process algebras; it is also inspired by the notion of channels found in BPEL and elsewhere.) In a given application domain, channels may be used to restrict which service activity occurrences can communicate with which other service activity occurrences, and to impose further structure on the communication via messages (e.g., requiring intuitively that messages are stored in FIFO queues). In an application domain the family of channels might be fixed, or might be dynamically created, destroyed, and/or modified. (Some particular approaches to structuring how messages are passed are provided by the possible extensions named Ordered Channels, Read & Destroy, and FIFO Message Queue; other approaches can also be represented in application domains built on FLOWS-Core.)
8. To represent the acquisition and dissemination of information *inside a Web service* we follow the spirit of the ontology for "knowledge-producing actions" developed in [*Scherl03*], which formed the basis for the situation calculus semantics of OWL-S inputs and effects [*Narayanan02*]. (Some background on the knowledge-producing actions framework is provided in Subsection 6.2 below). Inputs are conceived as knowledge-preconditions, and outputs as knowledge-effects. That is, the needed input values must be known prior to an occurrence (i.e., execution) of an atomic process. Also, following the occurrence of an atomic process, values of associated outputs are known (assuming appropriate conditions in the conditional effect are satisfied). This treatment of information provides for the use of a notation **Knows**(*P*), where *P* is a logical formula, which has the intuitive meaning in our context that a service activity occurrence "knows" that *P* is true (at some point in time). This approach provides a declarative basis that enables reasoning about what information is and is not available to a service activity occurrence at a given time. This is essential, since in the real world the mechanisms available to Web services for gathering information are typically limited to (a) receiving messages and (b) performing atomic processes that interact with domain-specific fluents. Further, in a given application domain it is typical to restrict a service so that it can gain access to a specified set of domain-specific relations and fluents. Using FLOWS-Core it will thus be possible to precisely specify and study properties of Web services related to data flow and data access. (In the current axiomatization the epistemic fluents related to **Knows** intuitively correspond to knowledge held by the overall system. A planned refinement is to relativize these fluents, in order to represent the knowledge of individual service occurrences.)

FLOWS-Core does not provide any explicit constructs for the structuring of processing inside a Web service. (Some minimal constructs are available in PSL OuterCore, including primarily the `soo_precedes` predicate, which indicates that one atomic activity occurrence in a complex activity occurrence precedes in time another atomic activity occurrence.) This is intentional, as there are several models for the internal processing in the standards and literature (e.g., BPEL, OWL-S Process Model, Roman model, guarded automata model), and many other alternatives besides (e.g., rooted in Petri nets, in process algebras, in workflow models, in telecommunications). We introduce (possible) extensions of FLOWS-Core to represent some of these service-internal process models (e.g., Control Constructs, Roman model, Guarded automata).

When using FLOWS-Core, it is often useful to model humans (and organizations and other non-service agents) as specialized formal services (or combinations of services, corresponding to the different roles that they might play). We are not proposing that all aspects of human behavior can or should be captured using FLOWS-Core constructs, but rather that a useful abstraction of human (and organization) behaviors, *in the context of interaction with Web services*, is to focus on the message creation/reading activities that humans perform, and on the knowledge that they can convey to other services or obtain from them (via messages). It is typical to assume that humans cannot directly perform atomic processes for testing or directly impacting domain-specific fluents, but must rather achieve that by invoking standard (web) services. For an abstract representation (*H*, *B*) of human *H* with complex activity *B*, in some cases it is natural to view occurrences of *B* as relatively short-lived (e.g., corresponding to one session with an on-line book seller), and in other cases to view occurrences of *B* as long-lived (e.g., the life-long family of interactions between the human and the on-line bookseller). In some cases it might be natural to model a human as embodying several formal services (*H_1*, *B_1*), ..., (*H_n*, *B_n*), corresponding to different roles that the human might play at different times.

We note that FLOWS-Core can be used to faithfully represent and study service process models that do not include all of the notions just listed. For example, OWL-S focuses on world-changing atomic processes and largely ignores the variations that can arise in connection with message passing. A given application domain can be represented in FLOWS-Core in such a way that the message passing is essentially transparent, so that it provides a faithful simulation of an OWL-S version of the domain. Alternatively, a domain theory can be specified in FLOWS-Core which essentially ignores the ability of atomic processes to modify facts "in the world". Such theories offer the possibility of faithfully simulating domains defined in terms of standards such as WSDL, BPEL, and WS-Choreography, which focus largely on service I/O signatures, and the process and data flow involved with message passing.

In the remainder of this subsection we provide more detail concerning the major concepts of FLOWS-Core.

3.1.1 Service

A service is an object. Associated with a service are a number of service descriptors, as described in Section 2. Also associated with every service is an activity that specifies the process model of the service. We call these activities *service activities*. These service activities are PSL activities. A service occurrence is an occurrence of the activity that is associated with the service.

We associate three relations with services:

```
service(thing)
service_activity(service,activity)
service_occurrence(service,activity_occurrence)
```

Link to Axiomatization

Readers may also be interested in the discussion of inputs, (conditional) outputs, preconditions and conditional effects of the complex activity that describes the process model of a service. This is discussed towards the end of the next subsection.

3.1.2 Atomic Process

A fundamental building block of the FLOWS process model for Web services is the concept of an atomic process. An atomic process is a PSL activity that is generally (but not mandated to be) a subactivity of the activity associated with a service (i.e., the activity referenced in the relation `service_activity (service,activity)` introduced in Section 3.1.1). An atomic process is directly invocable, has no subprocesses and can be executed in a single step. Such an activity can be used on its own to describe a simple Web service, or it can be composed with other atomic or composite processes (i.e., complex activities) in order to provide a specification of a process workflow. The composition is encoded as ordering constraints on atomic processes using the FLOWS control constructs described in Section 2.2.2. Associated with an atomic process are zero or more parameters that capture the inputs, outputs, preconditions and (conditional) effects (IOPEs) of the atomic process. More on IOPEs in a moment.

Types of Atomic Processes

In FLOWS we use atomic processes to model both the domain-specific activities associated with a service, such as `put_in_cart(book)` or `checkout (purchases)`, but also the atomic processes associated with producing, reading and destroying the messages that are sent between Web services. We distinguish these categories of atomic processes as follows:

1. *Domain-specific Atomic Process*: Intuitively, this kind of atomic process is focused on accessing and possibly modifying domain-specific relations and fluents. It is not able to directly manipulate messages or channels, or in other words, it cannot access or modify the service-specific relations and fluents concerned with messages or channels. In particular, this kind of activity is intended to model exclusively (a) the knowledge that a process needs to execute (i.e., the input parameter values), (b) the pre-conditions about domain-specific relations and fluents that must hold for successful execution, (c) the impact the execution of the activity has on domain-specific fluents, and (d) the knowledge acquired by execution (from the output parameter value s, and from whether the execution was successful or not). Occurrences of activities of this kind do not include anything concerning messages -- no producing, reading or destroying of them. (The occurrences may implicitly rely on previous message handling activities, e.g., because it uses for its input knowledge that was acquired by previously reading a message.)
2. *Produce_Message*: Occurrences of activities of this kind create a new message object. If a channel is used for the transmission of this message, then immediately after the message is produced it will be present in that channel. An epistemic pre-condition for producing the message will be that appropriate values are "known" by the service occurrence for populating the parameter values of the message. (In some domains, another pre-condition on producing a message will be that there is "room" in some channel or queue that is used to model the message transmission system.)
3. *Read_Message*: Occurrences of activities of this kind are primarily knowledge producing. Specifically, immediately after a Read_Message occurrence information about the payload of the message will be known. Thus, there is a close correspondence between the treatment of output values from a domain-specific atomic process, and the treatment of the payload of a message resulting from a Read_Message occurrence.
4. *Destroy_Message*: Occurrences of activities of this kind have the effect of destroying a message; after that time the message cannot be read.
5. *Channel Manipulation Atomic Processes* are used to create and destroy channels, and to modify their properties (what services can serve as source or target for the channel, and what message types can be transmitted across them). These are discussed briefly in Section 3.1.5 below.

In the FLOWS ontology, the message-specific atomic activities are distinguished by the following unary relations.

```
Produce_Message(atomic_process)
Read_Meassage(atomic_process)
Destroy_Message(atomic_process)
```

where `atomic_process` is an a FLOWS atomic process.

Atomic Process IOPEs

Associated with each atomic process are (multiple) input, output, precondition and (conditional) effect parameters (IOPEs). The inputs and outputs are the inputs and outputs to the program that realizes the atomic process. The preconditions are any conditions that must be true of the world for the atomic process to be executed. In most cases, there will be no precondition parameters since most software (and Web services are generally software) has no physical preconditions for execution, at least at the level at which we are modeling it. Finally, the conditional effects of the atomic process are the side effects of the execution of the atomic process. They are conditions in the world that are true following execution of the process, e.g., that a client's account was debited, or a book sent to a particular address. These effects, may be conditional, since they may be predicated on some state of the world (for example, whether a book is in stock).

In the FLOWS ontology, IOPEs are associated with an atomic process via the following binary and ternary relation. The inputs and outputs of an atomic process are represented by functional fluents that take on a particular value e.g.`bookname(book)="Green Eggs and Ham"`. We acknowledge that the actual input may be conveyed to the atomic process (e.g., as a result of a read-message process or via some internal communication) as rich structured data, but it is ultimately representable within our ontology as a conjunction of its component fluent parts. The preconditions and effects of an atomic process are expressed as arbitrary first-order logic formulae. Since formulae are not reified in FLOWS (note that fluents are reified, and further that formulae are reified in SWSL-Rules), we must associate a relational fluent with each formula we wish to represent in our IOPE relations. This fluent is true iff the formula we associate it with is true. We use this `trick' both for preconditions, for effects and for the conditions associated with conditional outputs and conditional effects. In the case where an output or effect is unconditional, the condition fluent is `true`.

```
input(atomic_process, input_fluent)
output(atomic_process, cond_formula_fluent, output_fluent)
precond(atomic_process, precond_formula_fluent)
effect(atomic_process, cond_formula_fluent,effect_formula_fluent)
```

where

- `atomic_process` is an atomic process, and
- `input_fluent` and `output_fluent` are functional fluents, that serves as the input (output, respectively) for `atomic_process`. This functional fluent is a term in the language of the domain theory. (Recall that the relation `input` (respectively `output`) might associate multiple input_fluents or output_fluents with a single atomic process.)
- `cond_formula_fluent` is a relational fluent that is true if and only if the condition upon which an output or effect is predicated, is true.
- `precond_formula_fluent` is a relational fluent that is true if and only if the precondition formula is true.
- `effect_formula_fluent` is a relational fluent that is true if and only if the effect formula is true.

These relations have also been systematically translated into ROWS, the SWSL-Rules ontology, as described in Section 2.5.

Formal models of software often include the inputs and outputs of the software, but not their side effects. Formal models of processes, such as those used in manufacturing or robotics, generally model physical preconditions and effects, but not inputs and outputs. To relate IOPEs in FLOWS, we intepret the inputs and (conditional) outputs of an atomic process as knowledge preconditions and knowledge effects [*Scherl03*], respectively, following [*McIlraith01*], [*Narayanan02*]. Our axiomatization of atomic processes makes explicit that if a fluent is an input (output, respectively) of an atomic process then it is a knowledge precondition (effect, respectively) of that process. Despite PSL's underpinnings in situation calculus, there is no explicit treatment of epistemic fluents (knowledge fluents). As such, included in the axiomatization of FLOWS-Core are the axioms for epistemic fluents.

Link to Axiomatization

Domain-Specific Axioms for Atomic Processes

Earlier in this section, we identified 5 classes of atomic processes. The atomic processes associated with messages and channels are discussed further in subsection sections relating specifically to messages and channels. Here we discuss the axiomatization of domain-specific atomic processes.

Domain-specific atomic processes are *domain specific* and as such are not part of the FLOWS ontology per se. Nevertheless, FLOWS imposes a syntactic form on the axioms that comprise a domain-specific axiomatization of an atomic process. Below we provide a reference grammar for the axioms associated with inputs, outputs preconditions and effects. In practice, we do not expect users of FLOWS to write these axioms, but rather to provide a specification of salient features of their process IOPEs using a high-level process description syntax that is then compiled into the syntactically correct FLOWS axioms. The syntax we propose is below, followed by the reference grammar.

Process Description Syntax

The IOPEs for atomic processes may be specified using the following presentation syntax. Note that both outputs and effects may be conditional. The BNF grammar below, captures this. A `psuedo_state_formula` is a state formula whose occurrence argument is suppressed. Non-fluent formulae in `psuedo_state_formula`s may be differentiated by prefixing the relation name with an asterisk (*).

```
< atomicprocess_name > {
         Atomic
         input < input_fluent_name >
         output < output_fluent_name > |  < psuedo_state_formula >
 < output_fluent_name >
         precondition < psuedo_state_formula >
         effect < psuedo_state_formula > | < psuedo_state_formula >
; < psuedo_state_formula >
}
```

Reference Grammar

- Precondition Axioms (link) A precondition of an atomic process is a formula that states that the atomic process cannot be executed until this formulae holds.
- Effect Axioms (link) An effect of an atomic process is a formula that states that if certain conditions hold, then execution of the atomic process will result in other conditions holding.
- Input Axioms (link) Inputs are treated as knowledge preconditions. As such, an input axiom of an atomic process is a formula that states that the atomic process must know the value of that input. In order for atomic process to execute, all (knowledge) preconditions must hold.
- Output Axioms (link) Likewise, outputs are knowledge effects. A knowledge effect of an atomic process is aformula that states that if certain conditions hold, then executing the atomic process will result in other knowledge conditions holding. If the atomic process knows the output, then the knowledge effect holds.

Walking Through an Example

Process Description Syntax Here is the presentation syntax for an atomic process called `simple_buy_book`. It has two inputs, `client_id` and `book_name`, and three outputs: i) a confirmation of the request, ii) the book price, if the book is in stock, and iii) an order rejection, if the book is out of stock. This simple atomic process also has one effect. The atomic process will debit the client's account (assumed to be on file and accessible through the client_id) by the price of the book, if the book is in stock. Note that as with the majority of non-device Web services, there are no (physical) preconditions associated with the execution of that atomic process. The process must merely know its inputs in order to execute.

```
simple_buy_book     {
        Atomic
        input client_id
        input book_name
        output request_confirm(client_id,book_name)
        output (name(book,book_name) and in_stock(book)) price(book)
        ouptput (name(book,book_name) and not in_stock(book)) reject(client_id,book_name)
        effect (name(book,book_name) and in_stock(book)) debit(client_id,price(book))
}
```

Domain-Specific Axiomatization The process description syntax is manually or automatically compiled into the following `input` and `output` relations:

```
input(simple_buy_book, client_id)
input(simple_buy_book, book_name)
output(simple_buy_book, true, request_confirm(client_id,book_name))
output(simple_buy_book, condition1(book_name), price(book))
output(simple_buy_book, condition2(book_name), reject(client_id,book_name))
effect(simple_buy_book, condition1(book_name),debit(client_id,price(book))
```

In order to avoid reifying formulae, we associate a unique fluent with the condition formula of an output or effect, so that we may talk about it in the relations `output` and `effect`.

```
forall ?occ,?booknm,?book
      ((condition1(?booknm,?occ) ==> (name(?book,?booknm) and in_stock(?book, ?occ))) and
      (name(?book,?booknm) and in_stock(?book,?occ)) ==> condition1(?booknm,?occ))

forall ?occ,?booknm,?book
      ((condition2(?booknm,?occ) ==> (name(?book,?booknm) and not in_stock(?book,?occ))) and
      (name(?book,?booknm) and in_stock(?book,?occ)) ==> condition2(?booknm,?occ))
```

Following the reference grammar, the IOPEs declared in the presentation syntax are manually and automatically compiled into IOPE axioms according to the reference grammar.

Input Axioms, i.e., Knowledge-Precondition Axioms

Each input fluent becomes a knowledge precondition of the occurrence of the `simple_buy_book` atomic process. Note that there are no non-knowledge preconditions to the execution of this atomic process.

```
forall ?occ,?clientid,?booknm,?book
       (occurrence_of(?occ,simple_buy_book(?clientid,?booknm)) and legal(?occ)
       ==>
           (holds(Kref(book_name(?s)),?occ) and holds(Kref(client_id(?s)),?occ))
```

Effect Axiom

The sole effect of execution of this atomic process is that the client's account will be debited, if the book is in stock.

```
forall ?occ,?clientid,?booknm,?book
        (occurrence_of(?occ,simple_buy_book(?clientid,?booknm)) and
        name(?book,?booknm) and prior(in_stock(?book,?occ)
        ==>
           holds(debit(?clientid,price(?book,?s,?occ)),?occ)
```

Output Axioms, i.e., Knowledge-Effect Axioms

A confirmation request is an unconditional output of the `simple_buy_book` atomic process.

```
forall ?occ,?clientid,?booknm,?book
        occurrence_of(?occ,simple_buy_book(?clientid,?booknm))
        ==>
           holds(Kref(request_confirm(?clientid,?booknm)),?occ)
```

If the book is in stock, then the price of the book will be an output, i.e., a knowledge effect of the `simple_occ_book` atomic process.

```
forall ?occ,?clientid,?booknm,?book
        (occurrence_of(?occ,simple_buy_book(?clientid,?booknm)) and
        name(?book,?booknm) and prior(in_stock(?book,?occ))
        ==>
            holds(Kref(price(?book)),?occ)
```

If the book is not in stock, then an order rejection message is output. It is a knowledge effect of the `simple_occ_book` atomic process.

```
forall ?occ,?clientid,?booknm,?book
        (occurrence_of(?occ,simple_buy_book(?clientid,?booknm)) and
        name(?book,?booknm) and not prior(in_stock(?book,?occ)))
        ==>
           holds(Kref(reject(?clientid,?booknm)),?occ)
```

Building on this introduction, the example in Section 2.1 of the Application Scenarios document describes a simple hypothetical service that involves several domain-specific and message-handling atomic processes. Application Scenario Example 2.1(a) illustrates, at a high level, how the sequences of atomic process occurrences in an occurrence (execution) of the overall system might be constrained by the FLOWS-Core ontology axioms and an application domain theory. **Service IOPEs**

Service IOPEs, or more accurately, the IOPEs of the complex activity associated with a service, are not formally defined. Inputs, outputs, preconditions and effects are only specified for atomic processes. Nevertheless, for a number of automation tasks, including automated Web service discovery, enactment and composition, it may be compelling to consider the inputs, outputs, preconditions and effects of the complex activity that describes the full process model of the service, `service_activity(service,activity)`. In some cases, these properties may be inferred from the IOPEs of the constituent atomic processes, producing IOPEs conditional on the control constraints defining the service. As an alternative to repeated inference, these *computed* IOPEs may be added to the representation of the complex activity defining the service.

We provide a means for the axiomatizer to define the IOPEs of a complex activity, using the relations and process description syntax described above for atomic processes. Note that these relations are merely descriptive and though it is desirable, the ontology does not mandate that they reflect the IOPEs of their constituent atomic processes.

The IOPEs for complex activities are systematically translated into ROWS using the SWSL-Rules language. They are commonly used for Web service discovery.

Link to ROWS Axiomatization for IOPEs.

3.1.3 composedOf

In general, the activity associated with a service can be decomposed into subactivities associated with other services, and constraints can be specified on the ordering and conditional execution of these subactivities. The composedOf relation specifies how one activity is decomposable into other activities.

Link to Axiomatization

3.1.4 Message

A key aspect of several Web services standards, including WSDL, BPEL, and WS-Choreography, is the explicit representation and use of messages. This fundamental approach to modeling data flow is largely absent from OWL-S. To enable the direct study of the semantic implications of messages, they are modeled as explicit objects in the FLOWS-Core ontology.

(Although messages are represented as explicit objects, it is nevertheless possible with FLOWS to create an application domain in which the messaging is transparent. This might be done by incorporating specific constraints into the application domain, or by using a views mechanism.)

Associated with messages are *message_type* and *payload* (or "body"), and perhaps other attributes and properties. Messages are produced by atomic processes occurrences of kind Produce_Message. Every message has a *message_type*, which is again an object in the FLOWS ontology. As detailed below, these message types will be associated with one or more (abstract) functional fluents, which, intuitively speaking, will provide information on how the payloads of messages of a given message type impact the knowledge of the service activity occurrences reading the message, and the knowledge pre-conditions of the service attempting to produce the message. FLOWS-Core is agnostic about the form of message type itself. It could be an XML datatype, a database schema, or an OWL-like class description of a complex object.

Information about messages can be represented in FLOWS-Core using a 3-ary relation `message_info`.

```
message_info(msg, msg_type, payload)
```

where

- *msg* is a message;
- *msg_type* is the message type associated with *msg*;
- *payload* is, intuitively, the value or "body" associated with the message, generally containing structured data.

An important property of messages, which is captured in the axiomatization of FLOWS-Core, is that the type and payload value associated with a message are immutable. This is suggested by the fact that `messages_info` is a relation rather than a fluent -- the tuples in relation `messages_info` are not time-varying. (At first glance, the reader may find this counter-intuitive: If a message is created and destroyed through time, then how can information about its payload be fixed or known "before" and "after" the message's existence? The answer is that an interpretation of a FLOWS-Core theory holds information about the full history of the set of possible occurrences (executions) of the application domain being modelled. From that perspective, which is "outside" of the time modeled in the domain, the relationship between a given message object and its payload is independent of the creation and destruction times of the message.)

In FLOWS-Core there are three relations which relate atomic process occurrences to the messages they access or manipulate.

```
produces(o,msg), reads(o,msg), destroy_message(o,msg)
```

where

- *o* is an AtomicProcess activity occurrence that is reading/producing/destroying the message (respectively)
- *msg* is the message that is produced/read/destroyed

(Additional constraints may apply to these activities if channels are in use. This is considered in the next subsection.)

Since FLOWS-Core does not commit to the form of message types, but only to the kinds of information that they can convey, FLOWS-Core includes a 2-ary relation `described_by`.

```
described_by(msg_type, io_fluent)
```

where

- *msg_type* is a message type, and
- *io_fluent* is a functional fluent, that serves as an input for the message producing process and as an output of the message reading process. (Similar to atomic processes in general) this functional fluent is a term in the language of the domain theory, which, intuitively speaking, is restricted in the theory so that it takes appropriate values at those times when there is an occurrence of a Produce_Message or Read_Message atomic process. (The relation `described_by` might associate multiple io_fluents with a single *msg_type*, corresponding to the different component fluent parts that can be encoded into the payloads of messages having this message type. Intuitively, the association of multiple io_fluents with a message type should be interepreted as a conjunction, e.g., saying that a message will hold both an ISBN and a book title.)

Analogous to the treatment of the inputs and outputs of domain-specific atomic processes, the io_fluent(s) associated with message types are constrained by the domain theory to correspond to appropriate values at the times when a message of given type is being produced or read. In this way, the relation `described_by` aids in defining the knowledge effects of a Read_Message activity as well as the knowledge pre-conditions that are present for a Produce_Message atomic process.

A Produce_Message atomic process occurrence has the effect of creating a message (and possibly its placement on a channel). The payload (or "body") of the message, in many cases, is some or all of the output of some previous atomic process occurrence. Indeed, a knowledge pre-condition for producing a message will be knowing the values needed to populate this payload. This knowledge might be derived from parts or all of the outputs (knowledge effects) of several previous domain-specific and/or read-message atomic process occurrences. In the case of communication between services, reading a message containing particular io_fluent(s), establishes that subsequently this io_fluent is known (to the service occurrence that read the message). As such the io_fluent(s) can be used to establish the knowledge preconditions of another atomic process occurrence. Once again, establishment of knowledge preconditions and effects is domain-specific, and thus is not within the purview of the FLOWS ontology. Rather, we provide a reference grammar for the Produce_Message and Read_Message atomic processes, which is consistent with the reference grammar for Atomic Process.

In many models of Web services found in the standards and the research literuature, if a message is read by some service then it is not available to be read by any other service. Because of the separation of the Read_Message and Destroy_Message atomic processes, FLOWS-Core is more general than those models. In particular, in some application domains, many service activity occurrences can read a message before it is "destroyed".

As noted previously, the example of Section 2.1 (in the Application Scenarios document) illustrates the notions of domain-specific and message-handling atomic processes. Example 2.1(b) there illustrates in particular how constraints on the relationships of domain-specific atomic prcesses can be used to infer, in the context of an occurrence of the overall system, that messages with certain characteristics must have been transmitted. The example of Section 2.2 provides additional illustrations of message-handling atomic processes.

Since messages are read, produced and destroyed via atomic processes, the production, reading and destruction of messages may be explicitly encoded by a domain axiomatizer, as special atomic processes, following the syntax for atomic process axioms described in the Reference Grammar above. Nevertheless, the FLOWS ontology is sufficiently expressive that the existence of necessary activity occurrences related to messages can be inferred from the axiomatization. This illustrates one aspect of the power of the FLOWS ontology.

Link to Axiomatization

3.1.5 Channel

As noted before, *channels* are objects in the FLOWS-Core ontology, used as an abstraction related to message-based communication between Web services. Intuitively, a channel holds messages that have been "sent" and may or may not have been "received". In FLOWS-Core, there is no requirement that a message has to be "sent" using a channel. However, it the message is associated with a channel, then it must satisfy a variety of axioms.

The notion of 'channel' is inspired by, but not identical to, the notion of 'channel' found in many process algebras. It is typical in process algebras that the act of transmitting a message *m* via a channel involves two simultaneous actions, in which one process sends *m* and another process that receives *m*. In the basic notion of channels provided in FLOWS-Core there is no requirement for this form of simultaneity.

In FLOWS-Core, messages in transit may be associated with at most one channel (corresponding intuitively to the idea that the message is being sent across that channel). As part of an application domain, a channel might be associated with a single source service and single target service, or might have multiple sources and/or targets.

Channels might be pre-defined, i.e., included in the definition of an application domain. Or channels can be created/destroyed "on the fly" by atomic process occurrences in an application domain. For this reason, fluents are used to hold most of the information about channels.

We now introduce the fluents of FLOWS-Core that are used to hold information about channels and their associated messages. Three of the fluents are

```
channel_source(c,s), channel_target(o,s'), channel_mtype(o,msg_type)
```

where

- *c* is channel
- *s* is a service, where occurrences of this service may place messages (of appropriate type) onto this channel. (A service may have write-access to multiple channels, even for the same types of messages.)
- *s'* is a service, where occurrences of this service may read messages (of appropriate type) that are present in this channel. (A service may have read-access to multiple channels, even for the same types of messages.)
- *msg_type* is a message type, which indicates that messages of this type may be placed on this channel. (A channel may support multiple message types.)

The fourth fluent for channels in FLOWS-Core is

```
channel_mobject(c,msg)
```

where

- *c* is channel
- *msg* is a message, which is "currently" on channel *c*, meaning that *msg* has been produced by some atomic process occurrence, the atomic process occurrence is a sub-occurrence of a service occurrence that was eligible to place the message on *c*, and that *msg* has not yet been destroyed.

As noted above, FLOWS-Core provides the possibility that channels can be created and destroyed. This might be accomplished by (i) services which correspond to human administrators, (ii) specialized, automated services which have essentially an administrative role, (iii) ordinary Web services, or (iv) atomic processes which are not associated with any service. In any of these cases, the following kinds of atomic processes are supported.

1. *Create_Channel*
2. *Add_Channel_Source*
3. *Add_Channel_Target*
4. *Add_Channel_MType*
5. *Delete_Channel_Source*
6. *Delete_Channel_Target*
7. *Delete_Channel_MType*
8. *Destroy_Channel*

In most cases these have straight-forward semantics. In the case of destroying a channel, all messages on that channel are viewed as simultaneously "destroyed".

An illustration involving the dynamic creation and modification of channels is provided in Section 2.2(d) (in the Application Scenarios document)

Channels are not required to exist; however, if they do exist, then they satisfy the following constraints:

- If a message is contained in a channel, then it is produced by an occurrence of a service that is a source for the channel.
- If a message is contained in a channel, then it is read by an occurrence of a service that is a target for the channel.

Link to Axiomatization

3.2 Control Constraints

Most Web services cannot be modeled as simple atomic processes. They are better modeled as complex activities that are compositions of other activities (e.g., other complex activities and/or atomic processes). The Control Constraints extension to FLOWS-Core provides a set of workflow or programming language style control constructs (e.g., sequence, if-then-else, iterate, repeat-until, split, choice, unordered) that provide a means of specifying the behavior of Web services as a composition of activities. Control constraints impose constraints on the evolution of the complex activity they characterize. As such, specifications may be partial or complete. Reflecting some of PSL's underpinnings in the situation calculus, and by extension Golog, these control constructs follow the style of Web service modeling of atomic and composite processes presented in [*McIlraith01*] and [*OWL-S 1.1*].

3.2.1 Split

The subactivity occurrences of a Split activity are partially ordered, such that every linear extension of the ordering corresponds to a branch of the activity tree for the Split activity.

Link to Axiomatization

Link to Reference Grammar Process descriptions for Split activities specify the subactivity occurrences whose ordering is constrained by two relations from the PSL Ontology: soo_precedes (which imposes a linear ordering on subactivity occurrences) and strong_parallel (which allows all possible linear orderings on subactivity occurrences).

Process Description Syntax

```
< activity_name > {     Split   occurrence < subocc_varname > < subactivity_name >      < subocc_varname1 >
soo_precedes < subocc_varname2 >        < subocc_varname1 > strong_parallel < subocc_varname2 >}
```

Examples

```
buy_product(?Buyer) {   Split   occurrence ?occ1 transfer(?Fee,?Buyer,?Broker)  occurrence ?occ2 transfer(?
Cost,?Buyer,?Seller)    ?occ1 strong_parallel ?occ2}
```

```
forall ?y split(buy_product(?y))forall ?occ,?Buyer      occurrence_of(?occ,buy_product(?Buyer))
==>             (exists ?occ1,?occ2,?Fee,?Cost,?broker,?Seller                  occurrence_of(?occ1,transfer(?
Fee,?Buyer,?Broker)) and                        occurrence_of(?occ2 transfer(?Cost,?Buyer,?Seller))
and                     subactivity_occurrence(?occ1,?occ) and                  subactivity_occurrence(?occ2,?
occ) and                        strong_parallel(?occ1,?occ2))
```

3.2.2 Sequence

The subactivity occurrences of a Sequence activity are totally ordered (that is, the activity tree for the activity contains a unique branch).

Link to Axiomatization

Reference Syntax Process descriptions for Sequence activities specify the subactivity occurrences whose ordering is constrained by one relations from the PSL Ontology: soo_precedes (which imposes a linear ordering on subactivity occurrences).

Process Description Syntax

```
< activity_name > {        Sequence        occurrence < subocc_varname > < subactivity_name >        <
subocc_varname1 > soo_precedes < subocc_varname2 >}
```

Examples

```
transfer(?Amount,?Account1,?Account2)  {        Sequence        occurrence ?occ1 withdraw(?Amount,?
Account1)       occurrence ?occ2 withdraw(?Amount,?Account2)    ?occ1 soo_precedes ?occ2}
```

```
forall ?x,?y,?z sequence(transfer(?x,?y,?z))forall ?occ occurrence_of(?occ,transfer(?Amount,?Account1,?
Account2)) ==>          (exists ?occ1,?occ2                     occurrence_of(?occ1,withdraw(?Amount,?
Account1)) and                  occurrence_of(?occ2,deposit(?Amount,?Account2)) and
subactivity_occurrence(?occ1,?occ) and                  subactivity_occurrence(?occ2,?occ)
and                     soo_precedes(?occ1,?occ2,transfer(?Amount,?Account1,?Account2))
```

3.2.3 Unordered

The subactivity occurrences of a Split activity are partially ordered, such that all subactivities are incomparable and every linear extension of the ordering corresponds to a branch of the activity tree.

Link to Axiomatization

Reference Syntax Process descriptions for Unordered activities specify the subactivity occurrences of the activity.

Process Description Syntax

```
< activity_name > {        Unordered         occurrence < subocc_varname > < subactivity_name >}
```

Examples

```
ConferenceTravel {      Unordered       occurrence ?occ1 book_flight    occurrence ?occ2 book_hotel
occurrence ?occ3 register}

Unordered(ConferenceTravel)forall ?occ  occurrence_of(?occ,ConferenceTravel) ==>        (exists ?occ1,?occ2,?
occ3            bag(?occ) and           occurrence_of(?occ1,book_flight) and            occurrence_of(?occ2,
book_hotel) and         occurrence_of(?occ3,register) and               subactivity_occurrence(?occ1,?occ)
and             subactivity_occurrence(?occ2,?occ) and         subactivity_occurrence(?occ3,?occ))
```

3.2.4 Choice

A Choice activity is a nondeterministic activity in which only one of the subactivities occurs.

Link to Axiomatization

Reference Syntax Process descriptions for Choice activities specify the subactivity occurrences of the activity.

Process Description Syntax

```
< activity_name > {        Choice         occurrence < subocc_varname > < subactivity_name >}
```

Examples

```
travel(?Destination)  { Choice  occurrence ?occ1 book_flight(?Destination)      occurrence ?occ2 book_train(?
Destination)}

forall ?x choice(travel,?x)forall ?occ,?Destination     occurrence_of(?occ,travel(?Destination))
==>             ((exists ?occ1                  occurrence_of(?occ1,(book_flight,?Destination))
and                     subactivity_occurrence(?occ1,?occ))             or              (exists ?
occ2                    occurrence_of(?occ2,(book_train,?Destination)) and
subactivity_occurrence(?occ2,?occ)))
```

3.2.5 IfThenElse

An IfThenElse activity is a nondeterministic activity such that the subactivity which occurs depends on the state condition that holds prior to the activity occurrence.

Link to Axiomatization

Reference Syntax Process descriptions for IfThenElse activities consist state constraints on the subactivity occurrences of the activity. Each constraint specifies a state condition and the subactivity that occurs when the state condition is satisfied.

Process Description Syntax

```
< activity_name > {        IfThenElse         if < state_formula > then occurrence < subocc_varname > <
subactivity_name >}
```

Examples

```
travel(?Destination, ?Account)  {       IfThenElse      if greater(budget_balance(?Account),1000) then
occurrence ?occ1 book_deluxe(?Destination)      if greater(1001,budget_balance(?Account)) then occurrence ?
occ2 book_economy(?Destination)}
```

```
forall ?x IfThenElse(travel,?x)forall ?occ,?Destination (occurrence_of(?occ,travel(?Destination) and    prior
(greater(budget_balance(?Account),1000),?occ))  ==>     (exists ?occ1           occurrence_of(?occ1,
book_deluxe(?Destination)) and          subactivity_occurrence(?occ1,?occ))forall ?occ,?Destination
(occurrence_of(?occ,travel(?Destination) and    prior(greater(1000,budget_balance(?Account)),?occ))
==>     (exists ?occ2           (occurrence_of(?occ2,book_economy(?Destination)) and
subactivity_occurrence(?occ2,?occ))
```

3.2.6 Iterate

An Iterate activity, is composed of a subactivity that occurs an indeterminate number of times.

Link to Axiomatization

Reference Syntax Process descriptions for Iterate activities specify the subactivity occurrences of the activity.

Process Description Syntax

```
< activity_name > {        Iterate        occurrence < subocc_varname > < subactivity_name >}
```

Examples

3.2.7 RepeatUntil

In a RepeatUntil activity, there are repeated occurrences of the subactivity until the state condition holds.

Link to Axiomatization

Link to Reference Syntax Process descriptions for RepeatUntil activities specify the subactivity occurrences of the activity, and the state condition that constrains the end of the occurrence of the activity.

Process Description Syntax

```
< activity_name > {        Iterate        while < state_formula > then occurrence < subocc_varname > <
subactivity_name >}
```

Examples

3.3 Ordering constraints

The intent of this extension is to provide a family of simple constraints based on sequencing properties of atomic processes. This enables succinct specification of requirements such as that payment must be received before shipping.

3.3.1 OrderedActivity

Any branch of the activity tree of an OrderedActivity satisfies the ordering constraints in the process description. Note that this is weaker than the Control Constraints, since there may exist sequences of subactivity occurrences that satisfy the ordering constraints but which do not correspond to branches of the activity tree for an OrderedActivity.

Link to Axiomatization

Reference Syntax Process descriptions for OrderedActivity specify the subactivity occurrences and sentences using the basic ordering relation `min_precedes` that is axiomatized in the PSL Ontology.

Process Description Syntax

The presentation syntax provides three constructs that facilitate the specification of ordering constraints:

- The symbol `;` is used to specify a linear ordering constraint over the subactivity occurrences of an activity occurrence;
- The symbol `,` is used to specify a partial ordering constraint over the subactivity occurrences of an activity occurrence;

- The symbol `!` is used to specify an ordering constraint in which some possible orderings over subactivity occurrences are not allowed.

```
< activity_name > {        OrderedActivity        occurrence < subocc_varname > < subactivity_name >        <
subocc_varname1 > ; < subocc_varname2 >        < subocc_varname1 > , < subocc_varname2 >        ! <
subocc_varname >}
```

Examples

1. *In each path in the activity tree for S that contains an occurrence of a and an occurrence of b, the subactivity a occurs before the subactivity b.*

```
Activity1 {     OrderedActivity occurrence ?occ1 a      occurrence ?occ2 b      ?occ1 ; ?occ2}
```

```
forall ?occ,?occ1,?occ2 (occurrence_of(?occ,S) and      occurrence_of(?occ1,a) and      occurrence_of(?
occ2,b) and     subactivity_occurrence(?occ1,?occc) and  subactivity_occurrence(?occ2,?occ))
==>     min_precedes(?occ1,?occ2,S)
```

2. *In each path in the activity tree for S, if there is an occurrence of a and sometime after the occurrences of b and c are partially ordered.*

```
Activity2 {     OrderedActivity occurrence ?occ1 a      occurrence ?occ2 b      occurrence ?occ3
c       ?occ1 ; (?occ2 , ?occ3)}
```

```
forall ?occ,?occ1,?occ2 (occurrence_of(?occ,S) and      occurrence_of(?occ1,a) and      occurrence_of(?
occ2,b) and     occurrence_of(?occ2,c) and      subactivity_occurrence(?occ1,?occ) and
subactivity_occurrence(?occ2,?occ))     ==>     (min_precedes(?occ1,?occ2,S) and        min_precedes(?
occ1,?occ3,?S))
```

3. *In each path in the activity tree for S, there is an occurrence of a and sometime after there then: (i) there is an occurrence of b and a later occurrence of c, and also (ii) there is an occurrence of d and later an occurrence of e.*

```
Activity3 {     OrderedActivity occurrence ?occ1 a      occurrence ?occ2 b      occurrence ?occ3
c       occurrence ?occ2.d      occurrence ?occ5 e      ?occ1 ; ?occ2   ?occ2 ; ?occ3   ?occ1 ; ?
occ2.   ?occ2.; ?occ5}
```

```
forall ?occ,?occ1,?occ2,?occ3,?occ2.?occ5       (occurrence_of(?occ,S) and
occurrence_of(?occ1,a) and                        subactivity_occurrence(?occ1,?occ)
and                         occurrence_of(?occ2,b) and                        subactivity_occurrence(?
occ2,?occ) and                        occurrence_of(?occ3,c) and
subactivity_occurrence(?occ3,?occ) and                        occurrence_of(?occ2.d)
and                         subactivity_occurrence(?occ2.?occ) and                        occurrence_of
(?occ5,e) and                         subactivity_occurrence(?occ5,?occ))                  ==>
(min_precedes(?occ1,?occ2,S) and                          min_precedes(?occ2,?occ3,S)
and                            min_precedes(?occ1,?occ2.S) and                          min_precedes(?
occ2.?occ5,S))
```

4. *In each path in the activity tree for S, there is no occurrence of b after the occurrence of a.*

```
Activity5 {     OrderedActivity occurrence ?occ1 a      occurrence ?occ2 b      ?occ1 ; ! ?occ2}
```

```
forall ?occ,?occ1,?occ2        (occurrence_of(?occ,S) and                        occurrence_of(?occ1,?
a) and                         subactivity_occurrence(?occ1,?occ) and
occurrence_of(?occ2,b) and                        subactivity_occurrence(?occ2,?occ))
==> (not min_precedes(?occ1,?occ2,S))
```

3.4 Occurrence constraints

The intent of this extension is to make it easy to specify constaints requiring that there are indeed occurrences of certain activities, e.g., payments, or the transmission of certain messages.

3.4.1 OccActivity

Any branch of the activity tree of an OccActivity satisfies the occurrence constraints on subactivities.

Link to Axiomatization

Reference Syntax Process descriptions for OccActivity are sentences that specify the existence or nonexistence of subactivity occurrences within occurrences of the activity.

Process Description Syntax The presentation syntax provides three constructs that facilitate the specification of occurrence constraints:

- The symbol & is used to specify a conjunctive constraint over the subactivity occurrences of an activity occurrence;
- The symbol + is used to specify a disjunctive constraint over the subactivity occurrences of an activity occurrence;
- The symbol `&tilde` is used to specify a constraint in which some subactivity occurrences are not allowed.

```
< activity_name > {        OccActivity        occurrence < subocc_varname > < subactivity_name >        <
subocc_varname1 > & < subocc_varname2 >        < subocc_varname1 > + < subocc_varname2 >        ~ <
subocc_varname >}
```

Examples

1. *In each path in the activity tree for S, there is an occurrence of a and an occurrence of b.*

```
Activity1 {     OccActivity     occurrence ?occ1 a      occurrence ?occ2 b      ?occ1 & ?occ2}
```

```
forall ?occ     occurrence_of(?occ,S) ==>               (exists ?occ1,?occ2
occurrence_of(?occ1,a) and                      occurrence_of(?occ2,b) and
subactivity_occurrence(?occ1,?occ) and                  subactivity_occurrence(?occ2,?occ))
```

2. *In each path in the activity tree for S, there is an occurrence of a and either an occurrence of b or an occurrence of c.*

```
Activity2 {     OccActivity     occurrence ?occ1 a      occurrence ?occ2 b      occurrence ?occ3
c       ?occ1 & (?occ2 + ?occ3)}
```

```
forall ?occ     occurrence_of(?occ,S) ==>               (exists ?occ1                   occurrence_of(?
occ1,a) and                     subactivity_occurrence(?occ1,?occ) and                  (exists ?
occ2                            occurrence_of(?occ2,b) and
subactivity_occurrence(?occ2,?occ))                     or                      (exists ?
occ3                            occurrence_of(?occ3,c) and
subactivity_occurrence(?occ3,?occ)))
```

3. *In each path in the activity tree for S, there is no occurrence of b.*

```
Activity2.{     OccActivity     occurrence ?occ1 b      ~ ?occ1}
```

```
forall ?occ        occurrence_of(?occ,S) ==>                 (not (exists ?
occ1                        occurrence_of(?occ1,b) and
subactivity_occurrence(?occ1,?occ)))
```

4. *In each path in the activity tree for S, there is an occurrence of a, and after there is no occurrence of b.*

```
Activity5 {     OccActivity     occurrence ?occ1 a      occurrence ?occ2 b      ?occ1 & ~ ?occ2}
```

```
forall ?occ        occurrence_of(?occ,S) ==>                 (exists ?occ1
occurrence_of(?occ1,?a) and                                subactivity_occurrence(?occ1,?occ)
and                      (not (exists ?occ2                                (occurrence_of(?occ2,b)
and                                       subactivity_occurrence(?occ2,?occ)
and                                       min_precedes(?occ1,?occ2 S)))
```

3.5 State constraints

This extension provides an explicit mechanism for associating triggered activities with states (of an overall system) that satisfy a given condition. A particular use of State Constraints is in exception handling.

3.5.1 TriggeredActivity

A TriggeredActivity is an activity which occurs whenever a state condition is satisfied.

Link to Axiomatization

Link to Reference Grammar

Process descriptions for a TriggeredActivity specifies the state condition that is associated with occurrences of the activity.

Process Description Syntax

```
    < activity_name > {        TriggeredActivity         < state_formula > }
```

Examples

3.6 Exceptions

Exceptions are central elements of most process modeling languages. For example, BPEL [*BPEL 1.1*], provides mechanisms based on faults, fault handlers, and compensation, for specifying exceptions and their treatments. FLOWS offers a variety of approaches to model such exceptions, which are presented in the following. The approaches presented below are not prescriptive.

3.6.1 State-based Exception Handling

The simplest method for exception handling is to define an exception handler as an activity whose occurrence is triggered by satisfaction of a state constraint.

An exception handler is defined as an activity that gets triggered when a specified (exceptional) state occurs. Consequently, the triggered activity state constraint (see section 2.2.5.1 above) can be used.

Process Description Syntax

```
    < activity_name > {     ExceptionHandlerActivity       < exception_state_formula >}
```

3.6.2 Message-based Exception Handling

Most process modeling approaches and programming languages (such as Java [Gosling96]) require exceptions either to be raised or ("thrown") explicitly by a modeled element (i.e., a process) or by the execution environment. A raised exception is then sequentially passed to exception handler, which have registered for it. Essentially, this approach represents the raising of a "message" which describes the exception and the catching (or consuming) of this message by some type of handler.

We place the term 'message' in quotes because this can be embodied in two ways in the conceptual model presented here. In the first embodiment, these "messages" would be messages between Web services, i.e., messages in the sense described above. In the second embodiment, these "messages" would be internal to a Web service, and would involve the passing of information from one family of activities in the service (intuitively, the main body of the service) to another family of activities in the service (intuitively, the exception-handling or fault-handling portion of the service).

In either case, the creation and transmission of an exception-based "message" can be modeled as a refinement of the notion of state-based exception handling. In particular, if the system is in a state in which an exception condition is satisfied, then an activity that creates or in some other way embodies a "message" transmission is triggered.

Exceptions get raised explicitly as messages to the parent activity. The parent activity can explicitly register an exception handler to listen for the specific exception message to be raised. If no exception handler is registered, then the mechanism "executing" the specification will be informed of the exception and may halt the execution of the overall process.

Process Description Syntax

If the activity raising the exception is not an AtomicProcess, then the service description satisfies the following syntax:

```
            <  exception_raising_activity_name  >  {
                    Unordered | Split | Sequence | Choice | IfThenElse | Iterate | RepeatUntil
                    ...
                    occurrence ?occ RaiseException(< exception_name > )
                    ...
            }

            <  exception_raising_activity_name  >  {
                    Unordered | Split | Sequence | Choice | IfThenElse | Iterate | RepeatUntil
                    ...
                    occurrence <  subocc_varname  >  <  subactivity_name  >
                    ...
                    raises_exception < exception_name >
            }
```

If the activity raising the exception is an AtomicProcess, then the service description satisfies the following syntax:

```
            <  exception_raising_activity_name  >  {
                    Atomic
                    input <  input_fluent_name  >
                    output <  output_fluent_name  >  |  <  psuedo_state_formula  >
          <  output_fluent_name  >
                    precondition <  psuedo_state_formula  >
                    effect <  psuedo_state_formula  >  | <  psuedo_state_formula  >
  <  psuedo_state_formula  >
                    raises_exception < exception_name >
          }
```

If the activity catching the exception is not an AtomicProcess, then the service description satisfies the following syntax:

```
          <  exception_raising_activity_name  >  {
                 Unordered | Split | Sequence | Choice | IfThenElse | Iterate | RepeatUntil
                 ...
                 occurrence <  subocc_varname  >  <  subactivity_name  >
                 ...
                 catches_exception < exception_name >
          }
```

If the activity catching the exception is an AtomicProcess, then the service description satisfies the following syntax:

```
          <  exception_raising_activity_name  >  {
                    Atomic
                    input <  input_fluent_name  >
                    output <  output_fluent_name  >  |  <  psuedo_state_formula  >  <  output_fluent_name  >
                    precondition <  psuedo_state_formula  >
                    effect <  psuedo_state_formula  >  | <  psuedo_state_formula  >
  <  psuedo_state_formula  >
                    catches_exception < exception_name >
          }
```

3.6.3 Extended Exception Handling

Intended Semantics The intended semantics for the extended exception handling mechanism are described in great detail in [Klein 00a, Klein 00b]. Summarized it is the following (see also Figure 2.2):

- Any activity can have a number of possible exceptions that might occur during is execution.
- Any exception can be handled by a number of exception handlers.
- The exception handling either attempts to find or fix the exception.
- When finding the exception it can either be detected during execution using some description of the exceptional state or some it can be anticipated. Anticipation means that the exception handler detects that an exception will occur if no "evasive action" is taken, i.e., nothing is changed from the current planned course of activities.
- When fixing an exception, it can either be resolved (i.e, ex post to its occurrence) or avoided (ex ante to its occurrence).
- Resolution "fixes" (or handles) the exceptional state by transforming it into a "regular" (or non-exceptional) one.
- Avoidance runs a procedure that ensures that an exceptional state will not arise (e.g., purchasing health care avoids large financial loss in the case of a sickness). It is run ex ante to the (possible) occurrence of the exception.
- Each of the exception handling types (anticipation, detection, resolution, avoidance) relates the exception to an exception handler of the respective type.

This setup allows to develop hierarchy of exception types (similar to Java), where each exception is associated with a (potentially domain independent) collection different exception handlers (see [Klein 00a] Figure 7 for an example).

The various approaches to exception handling within SWSL are illustrated in the application scenarios below

Link to Axiomatization

Process Description Syntax

The process description for the exception itself satisfies the following syntax:

```
          < exception_name >  {
                 is_handled_by      < exception_handling_activity_name >
                 is_found_by        < exception_finding_activity_name >
                 is_fixed_by        < exception_fixing_activity_name >
                 is_detected_by     < exception_detection_activity_name >
                 is_anticipated_by  < exception_anticipation_activity_name >
                 is_avoided_by      < exception_avoidance_activity_name >
                 is_resolved_by     < exception_resolution_activity_name >
          }
```

If the activity raising the exception is not an AtomicProcess, then the service description satisfies the following syntax:

```
<  exception_raising_activity_name  >  {
        Unordered | Split | Sequence | Choice | IfThenElse | Iterate | RepeatUntil
        ...
        occurrence ?occ RaiseException(< exception_name > )
        ...
}

<  exception_raising_activity_name  >  {
        Unordered | Split | Sequence | Choice | IfThenElse | Iterate | RepeatUntil
        ...
        occurrence <  subocc_varname  >  <  subactivity_name  >
        ...
        raises_exception < exception_name >
}
```

If the activity raising the exception is an AtomicProcess, then the service description satisfies the following syntax:

```
            <  exception_raising_activity_name  >  {
                       Atomic
                       input <  input_fluent_name  >
                       output <  output_fluent_name  >  |  <  psuedo_state_formula  >  <  output_fluent_name  >
                       precondition <  psuedo_state_formula  >
                       effect <  psuedo_state_formula  >  | <  psuedo_state_formula  >
    <  psuedo_state_formula  >
                       raises_exception < exception_name >
            }
```

3.7 Additional Extensions to FLOWS-Core

This subsection briefly describes some other possible extensions to FLOWS-Core. The intention here is to briefly indicate how FLOWS-Core can indeed serve as a foundation for the formal study of several models and approaches to Web services found in standards and in the literature. This underscores the fact that FLOWS-Core can serve as a common foundation for a unified study of two or more currently disparate Web services models.

3.7.1 Meta-Server

This section describes a modeling construct above FLOWS-Core, that could be used as the basis for a PSL extension of FLOWS-Core.

In many models and standards for Web services, including BPEL, the Roman model, the Conversation model, and the Guarded Automata model, it is typical to consider both Web *services* and the Web *servers* that run them. In essence, a Web server is an executing process (i.e., an occurrence of some PSL complex activity) that has the ability to "launch" occurrences of one or more kinds of service activities (i.e., that includes atomic processes whose occurrences have the impact of creating service activity occurrences). The extension Meta-Server is intended to capture salient aspects of such frameworks.

In this possible extension, a *(Formal) Server* is itself a Formal Service. There could be a fluent `server_service(Server:activity, Service:activity)`, where `server_service`(*R*, *S*) has the intuitive meaning that Server *R* is managing Service *S*.

A Server *R* will include at least the following kind of service-specific atomic process:

- *Launch_Service_Occurrence*: An occurrence of this kind of atomic process has the effect of creating a new occurrence of some Service associated with the Server (by the fluent `server_service`). Occurrences of activities of this type will typically include the creation of a message, which is read as the first atomic process occurrence of the newly launched service occurrence.

A Formal Service may have other kinds of atomic processes. For example, it is typical that it would have Read_Message atomic processes. In typical application, a Server would only read messages that are intended to launch new occurrences of a Service that is managed by the Server. It is also natural to include Destroy_Message.

Axioms would enforce that a Server *R* can launch occurrences of Service *S* only if `server_service`(*R*, *S*) holds. Furthermore, each Service can be associated with at most one Server in `server_service`.

An illustration of a Server is provided in Section 2.2(c) (in the Application Scenarios document)

(In [*BPEL 1.1*] (and in BPEL 1.0), messages which are intended to launch new occurrences of a service activity, have the value of special-purpose parameter `createInstance` set to "yes"; messages intended for existing service activity occurrences have this parameter set to "no". In BPEL 1.1, it is required that the first step of the newly launched service activity occurrence is the receiving of the launching message. In the Meta-Service extension here we have the Server actually read (in essence, receive) the message. It is possible to build a theory that builds on Meta-Service to capture the semantics of Service launching as specified in BPEL 1.1.)

Although not explicitly included in the description of the potential extension Meta-Server presented here, it is possible in an application domain to include additional atomic processes with a server, which intuitively have the impact of terminating services associated with the server, or monitoring their execution. A server may also have atomic processes that embody administrative actions, such as creating/destroying channels. We note that in practice, it will be typical to associate a family of domain-specific fluents with a server, with a constraint that only services associated by `server_service` with that server are able to access or update those fluents.

3.7.2 Local Store

This possible extension provides constructs for explicitly modeling that a Web service has a variable-based local store with imperative commands for assigning and accessing values associated with the variables. The use of a local store is found in [*BPEL 1.1*], and in theoretical models such as Guarded Automata. These constructs are not included in the FLOWS-Core, to permit the exploration of other forms of information passing (e.g., in the style of functional programming or based on a style inspired by data flow).

3.7.3 Ordered Channels, Read & Destroy, FIFO queues

This section describes three possible extensions to FLOWS-Core. These may be particularly useful because they capture an approach for managing messages common to many Web service implementations.

The ***Ordered channels*** possible extension would include axioms that would enforce, intuitively speaking, that all or some of the channels include an ordering for the message objects that are held in the channel at a given time. This ordering might be total or partial.

The ***Read & destroy*** possible extension would include axioms which have the intuitive meaning that whenever a service occurrence reads a message, then in the very next step of execution that occurrence destroys that message. This will imply that each message is read at most once (because it is destroyed immediately thereafter). This restriction corresponds to an assumption made by many Web services models, including, e.g., [*BPEL 1.1*] and Guarded Automata.

The ***FIFO message queue*** possible extension is intended to captures the standard first-in-first-out behavior for channels. This extension would build on top of the Ordered channels and Read & destroy extensions. This would include axioms so that the (relevant) channels have a total ordering on contained message objects, that whenever a message is placed on the channel it has the least position in the ordering; that if a message is read (and destroyed) from the channel then it must have the greatest position in the channel; and that the relative ordering of messages on a channel and that the relative ordering of messages on a channel never changes during their existence.

3.7.2 Potential Extensions for Relation-valued and XML-valued Parameters

FLOWS-Core is very general with regards to the types of values that can be used in parameters, both as input or output of domain-specific atomic processes, and as the payload of messages. In some cases, it may be useful to create theories or extensions on top of PSL-Core, that can be used for "encoding" and "decoding" certain kinds of commonly arising payloads. We briefly consider here one way in which a uniform approach can be developed for "encoding" and "decoding" paramater values that are essentially relations, i.e., finite sets of records with uniform type. (Other encodings are possible.)

The basic approach to representing this is illustrated in Section 2.2(a) (in the Application Scenarios document) by fluents such as `Transport_content_lists`. In that example, an occurrence of the `Warehouse_order_fulfillment` service can perform an occurrence *o* of the activity `send_shipment_from_warehouse` using a parameter `transport_content_list_id`*R* as the value of this parameter for *o*. Under the intended operation, the actual contents of the shipment can be obtained from fluent `Transport_content_lists`. by selecting on "`transport_content_list_id` = *R*" and then projecting out the `transport_content_list_id` field. This will give a set of records with signature `item_id:int, quantity:int`. A similar encoding schema can be used for relation-valued parameters with different signatures, and an analogous encoding can be formulated for *XML-Valued Parameters*.

3.7.5 Relationships Relevant to BPEL

We briefly describe here several modeling constructs that lie above FLOWS-Core, in order to build up to a representation of the process model aspects of services described using [*BPEL 1.1*]. We also suggest a refinement of BPEL, called here BPEL++, which incorporates the use of meta-service, and specific design choices on how messages are passed between BPEL services.

- *Roles* can provide constructs for explicit modeling of the notion that "roles" can be associated with (web) services; these can provide some structure in specifying the macro-structure of the intended business logic to be supported by a family of interoperating services, and in particular provide guidance on substitutability of services.
- A *WSDL* possible extension might formally specify the basic building blocks of the WSDL standards (e.g., [*WSDL 1.1*]).
- *Enactment Naming* could provide a mechanism for having explicit names for the enactments (i.e., occurrences) of service activities; this might be used, for example, to provide a formal basis for studying the properties of "correlation sets" as found in BPEL [*BPEL 1.1*].
- *XML-Valued Parameters* could provide mechanisms to represent parameters, of both messages and input/output arguments for atomic "world-changing" activities.

All of these possible extensions, along with possible extensions for Local store, Control constructs and Exceptions, can be used in defining a possible extension *BPEL(process)*, which can serve as a formal, declarative specification of essential aspects of some version of BPEL (e.g., [*BPEL 1.1*]).

The BPEL(process) possible extension in turn might be combined with Meta-service and FIFO Message Queue to create a possible extension *BPEL++(process)*, which can incorporate often made assumptions concerning message passing semantics that are not explicitly mentioned in the BPEL 1.1

specification.

3.7.6 Potential Extensions for the Roman and Guarded Automata Models

In many approaches to Web services, including [*BPEL 1.1*], a flowchart-based process model is used to specify the internal process flow of Web services. This approach is also found in OWL-S, and is essentially embodied in the Control Constructs extension of FLOWS-Core. In this section we briefly consider an alternative basis for the internal process model of Web services, based on the use of automata. Of course, there are other process modeling paradigms that may also be considered, including e.g., Petri nets and models, such as some process algebras, with unbounded sub-process spawning (see also the example in Section 2.3 of the Application Scenarios document).

We mention two important automata-based approaches from the literature.

- *Roman model.* This model [*Berardi03*] provides an abstraction of human-machine Web services. The focus is primarily on (a) atomic processes that can be performed by a Web service, and (b) the use of an automata-based representation of a Web service process model. With this model an important theoretical result concerning automated discovery and composition of Web services has been obtained, underscoring the value of using an automata-based framework.
- *Guarded automata model.* An important study here [*Fu04*] develops a model of Guarded finite-state automata suitable for Web services, and shows how BPEL [*BPEL 1.1*] programs can be simulated using them. Interestingly, the model used can simulate BPEL programs. Furthermore, verification techniques have been developed for this model.

Section 2.2(b) (in the Application Scenarios document) gives a brief illustration of how a guarded automaton can be used to specify the process model of a service.

4 Grounding a Service Description

The SWSO concepts for service description (as described in Section 3 of this document), and the instantiations of these concepts that describe a particular service, are *abstract* specifications, in the sense that they do not specify the details of particular message formats, transport protocols, and network addresses by which a Web service is accessed. The role of the *grounding* is to provide these more *concrete* details.

The Web Services Description Language (WSDL), developed independently of SWSL and SWSO, provides a well developed means of specifying these kinds of details, and is already in widespread use within the commercial Web services community. Therefore, we can ground a SWSO service description by defining mappings from certain SWSO concepts to WSDL constructs that describe the concrete realizations of these concepts. (For example, SWSO's concept of message can be mapped onto WSDL's elements that describe messages.) These mappings are based upon the observation that SWSL's concept of grounding is generally consistent with WSDL's concept of *binding*. As a result, it is a straightforward task to ground a SWSL service description to a WSDL service description, and thus take advantage of WSDL features that allow for the lower-level specification of details related to interoperability between services and service users.

In this release of SWSF, we give a sketch of the essential elements of grounding to WSDL, rather than a complete specification. Our approach here is similar to the WSDL grounding specified in [*OWL-S 1.1*]. We rely upon the soon-to-be-finalized WSDL 2.0 specification [*WSDL 2.0*]), which is in last call as of this writing.

Note that SWSO groundings are deliberately decoupled from SWSO abstract service descriptions, so as to enable reusability. An abstract service specification (say, for a bookselling service) can be coupled with one grounding in one context (say, when deployed by one online bookseller) and coupled with a different grounding (when deployed by a second online bookseller). The two booksellers would have completely distinct groundings, which would of course specify different network addresses for contacting their services, but could also specify quite different message formats.

4.1 Relationships between SWSO and WSDL

The approach described here allows a service developer, who is going to provide service descriptions for use by potential clients, to take advantage of the complementary strengths of these two specification languages. On the one hand (the abstract side of a service specification), the developer benefits by making use of SWSO's process model, and the expressiveness of SWSO ontologies, relative to what XML Schema Definition (XSD) provides. On the other hand (the concrete side), the developer benefits from the opportunity to reuse the extensive work done in WSDL (and related languages such as SOAP), and software support for message exchanges based on WSDL declarations, as defined to date for various protocols and transport mechanisms.

Whereas a default WSDL specification refers only to XSD primitive data types, and composite data types defined using XSD, a SWSO/WSDL specification can refer to SWSO classes and other types defined in SWSO (in addition to the XSD primitive and defined types). These types can, if desired, be used directly by WSDL-enabled services, as supported by WSDL type extension mechanisms. In this case the SWSO types can either be defined within the WSDL spec, or defined in a separate document and referred to from within the WSDL spec. However, it is not necessary to construct WSDL-enabled services that use SWSO types directly in this manner. When they are *not* used directly, translation mechanisms such as those outlined below may be used to spell out the relationships between the SWSO types and the corresponding XML Schema types used in the default style of WSDL definition.

We emphasize that an SWSO/WSDL grounding involves a *complementary* use of the two languages, in a way that is in accord with the intentions of the authors of WSDL. Both languages are required for the full specification of a grounding. This is because the two languages do not cover the same conceptual space. The two languages *do* overlap, to a degree, in that they both provide for the specification of "types" associated with message contents. WSDL, by default, specifies the types of message contents using XML Schema, and is primarily concerned with defining valid, checkable syntax for message contents. SWSO, on the other hand, does not constrain syntax at all, but rather allows for the definition of abstract types that are associated with logical assertions (IO fluents) in a knowledge base. WSDL/XSD is unable to express the semantics associated with SWSO concepts. Similarly, SWSO has no means to define legal syntax or to declare the binding information that WSDL captures. Thus, it is natural that a SWSO/WSDL grounding uses SWSO types as the abstract types of messages declared in WSDL, and then relies on WSDL binding constructs to specify the syntax and formatting of the messages.

4.2 Mapping between SWSO and WSDL

Web services are inherently message-oriented, and messages are also the central concern in the grounding of a SWSO service. The essence of this grounding approach is to establish a mapping between selected message-oriented constructs in SWSO and their counterparts in WSDL service descriptions.

To elucidate the requirements for the grounding, and clarify the scope of the current effort, we consider a hypothetical service enactment environment, or execution engine, based on SWSO; that is, a software system that enacts processes described using SWSO's process model. We assume that this system incorporates a knowledge base that contains the relevant SWSO service descriptions, and manages the fluents associated with service execution, as described earlier in this document. We call a system of this type a Semantic Web Services Execution Environment (SWSEE).

The primary motivation for this grounding approach is to make it possible for a SWSEE to handle messages that conform to WSDL service specifications. We want to allow for scenarios in which a SWSEE plays the role of a service provider (that is, manages the communications associated with a service from the provider's point of view), a service user, or both. When acting as a service provider, a SWSEE will need to accept a WSDL input message, extract the needed pieces of information from its content, and assert them into the knowledge base in accordance with the SWSO declarations of input fluents. This step of handling an input message, as a service provider, would happen in correspondence with the execution of a Read_Message atomic process. In addition, a service-providing SWSEE will need to generate outputs in correspondence with Produce_Message atomics, by pulling information from its knowledge base and formatting it into appropriate WSDL-conformant output messages.

Conversely, when acting as a service *user* (invoker), a SWSEE will need to execute a Produce_Message atomic for the purpose of invoking a remote Web service. In correspondence with this atomic activity, the SWSEE will need to produce a WSDL-conformant message to send to the remote service. (This message will be an input from the WSDL perspective, but an output from the SWSEE perspective in this situation.) Following that, it may need to receive a return message from the WSDL service, and will this will be done by a Read_Message atomic.

In designing these basic grounding mechanisms, we are noncommittal about the architectural role of a SWSEE. So as to be general and avoid relying on any particular architectural commitments, our approach here is minimalist. For example, we do not explain the mechanisms by which a SWSEE might arrange to make a *run-time selection and binding* to a particular WSDL service. Rather, we are concerned only with essential representational requirements that will be needed by the builders of enactment environments: we seek to provide a common denominator of representation by which the relationships between elements of WSDL and SWSO service descriptions can be declared.

Thus, the role of the SWSO/WSDL grounding (or at least the most central role) may be conceived as follows: it provides a SWSEE with the information it needs at runtime to handle (both send and receive) messages associated with WSDL service descriptions. To do this, it must provide the following things:

1. mappings between SWSO and WSDL elements that specify message patterns
2. mappings between SWSO's (abstract) message types and the concrete message types declared in WSDL
3. specification of a method for translating from an abstract message type to the corresponding concrete message type (i.e., *serializing* the abstract message type into the concrete message type)
4. specification of a method for translating from a concrete message type to the corresponding abstract message type (i.e., *deserializing* the concrete message type into the abstract message type)

In addition to the above, it is also desirable to provide the following:

- specification of a default declarative style for providing (3)
- specification of a default declarative style for providing (4)

We briefly describe each of these aspects of the grounding in turn. The needed mappings will be provided by introducing several new SWSO relations here. These relations differ from the process model relations of Section 3 in that they are not axiomatized. This is because they provide mappings to external entities (declared in WSDL) that are outside the scope of SWSO axiomatization.

4.2.1 Message Patterns

In WSDL 2.0 an *operation* declares a set of messages that are used in accordance with a message pattern, as in this simple example from the [*WSDL 2.0 Primer*]:

```
<operation name="opCheckAvailability"
        pattern="http://www.w3.org/2004/03/wsdl/in-out"
        style="http://www.w3.org/2004/08/wsdl/style/uri"
        safe = "true">
    <input messageLabel="In"
          element="ghns:checkAvailability" />
    <output messageLabel="Out"
          element="ghns:checkAvailabilityResponse" />
    <outfault ref="tns:invalidDataFault" messageLabel="Out"/>
</operation>
```

Here the message pattern is `in-out` (declared elsewhere). Each input/output message declaration gives an XML Schema document type (e.g., `ghns:checkAvailability`) and a label (e.g., `In`) that shows where the message is used in the pattern.

The correspondence to SWSO is this:

- A WSDL operation corresponds to a SWSO complex activity with one or more Produce_Message and Read_Message atomics as subprocesses. (Note: once this correspondence is established, details of network address and protocol come along "for free", by means of WSDL's binding constructs.)
- A WSDL message pattern corresponds to the pattern of messages produced and read by the Produce_Message and Read_Message atomics included within the control structure of the complex activity.

To indicate this mapping, we introduce a single SWSO relation:

```
activity_grounding(activity, wsdl_operation)
```

where activity refers to a complex activity declared in a SWSO service description, and wsdl_operation is a URL for a WSDL operation declaration.

4.2.2 Message Type Realization

As discussed in Section 3.1.4, SWSO's process model is silent as to the concrete form (syntax, serialization) associated with a message type. The nature of a message type is characterized abstractly, by associating it with one or more IO fluents, using the `described_by` relation. One central job of the grounding, then, is to specify a concrete syntax for each SWSO message type. In WSDL these concrete syntaxes are normally given by "message types" declared using XML Schema declarations. To distinguish these message types from those of SWSO, we refer to those of WSDL as "concrete message types".

We note that WSDL allows for the use of type specifications using other mechanisms than XML Schema. In this document, however, for simplicity, we assume the use of XML Schema in WSDL service descriptions.

To specify the concrete syntax associated with a SWSO message type, we introduce the relation

```
message_type_grounding(msg_type, wsdl_operation, wsdl_message_label)
```

where msg_type is an identifier for a message type defined using SWSO, wsdl_operation is a URI for an operation defined in a WSDL specification, and wsdl_message_label is a label used within that operation definition. The meaning of an instance of this relation is simply that a message of the given SWSO message type corresponds to a WSDL message declared with the given label, within the given WSDL operation.

4.2.3 Mapping SWSO Messages to WSDL Messages

In addition to showing the correspondence of SWSO message types with WSDL concrete message types, a grounding should indicate a precise method for constructing WSDL message content from SWSO message content. For this purpose we provide the `message_serialization_method` relation:

```
message_serialization_method(msg_type, wsdl_operation, wsdl_message_label, method)
```

The meaning of an instance of this relation is that a message of the given SWSO message type will be serialized appropriately for the WSDL message having the given label within the given operation. The serialization will be performed by the specified method.

In general this allows for an arbitrary method, which could be external (e.g., it could be a Java method). Therefore, we do not specify what kind of thing `method` refers to; this should be defined by each particular SWSEE implementation.

The specified method retrieves the information associated with the given message type by querying the knowledge base (in particular, the IO fluents for that message type), and generates the appropriate WSDL message containing that information.

msg_type, wsdl_operation, and wsdl_message_label are as described for `message_type_grounding`.

In Section 4.2.5, we sketch a means by which this translation method can be specified in a more declarative style.

4.2.4 Mapping WSDL Messages to SWSO Messages

The grounding must also specify how the various parts of an individual WSDL message get mapped into the fluents associated with that message type. This is done using the `message_deserialization_method` relation:

```
message_deserialization_method(wsdl_operation, wsdl_msg_label, msg_type, method)
```

The meaning of an instance of this relation is that a WSDL message, having the given label within the given operation, will be deserialized into the given

SWSO message type, by the specified method.

As with `message_serialization_method`, `method` is regarded as external and implementation-defined.

The specified method takes an arbitrary message of the kind specified by the given message label (within the given WSDL operation), extracts the information content from that message, and asserts the appropriate IO fluents into the knowledge base.

`msg_type, wsdl_operation`, and `wsdl_message_label` are as described above for `message_type_grounding`.

In Section 4.2.6, we sketch a means by which this translation method can be specified in a more declarative style.

4.2.5 Declarative Specification of SWSO-to-WSDL Message Mapping

Here, we sketch a general declarative means for specifying the serialization of a message of a given SWSO message type into a concrete message type specified in WSDL, which could be used as an alternative to declaring a message serialization method. The idea here is to use a knowledge base query to extract information from the IO fluents within the SWSEE's knowledge base, and use a message generation script to generate a message containing that information. The information is passed from the query to the message generation script by means of variable bindings. This approach is supported by the `message_serialization` relation:

```
message_serialization(msg_type, wsdl_operation, wsdl_msg_label, kb_query, msg_generation_script)
```

`msg_type` refers to a SWSO message type. The second and third arguments are references to a WSDL message element, as used in `message_serialization_method` and `message_deserialization_method`, above.

`kb_query` is a query expression, containing variables, that extracts the needed information from IO fluents in the knowledge base and binds the variables to that information. We do not specify a particular query language in this document. A SWSL-based query language, yet to be developed, is a possibility.

`msg_generation_script` is a script in a language such as XSLT [*XSLT*] that is suitable for generating an XML document. Variables mentioned in the script would correspond to variables mentioned in the KB query, and would be bound to the query's return values before the script is executed.

The meaning of an instance of this relation is that a message of the given SWSO message type will be serialized appropriately into the WSDL message having the given label within the given operation. The serialization will result by running the message generation script with variables bound as described above.

4.2.6 Declarative Specification of WSDL-to-SWSO Message Mapping

Here, we sketch a general declarative means for specifying the deserialization of a message of a given concrete message type (as specified in WSDL). This approach may be regarded as the inverse functionality of that specified using `message_serialization`. The idea here is to use a document query language to extract information from the concrete message content, and use an update script to insert that information into a SWSEE's knowledge base. The information is passed from the document query to the update script by means of variable bindings. This approach is supported by the `message_deserialization` relation:

```
message_deserialization(wsdl_operation, wsdl_msg_label, msg_type, msg_query, kb_update_script)
```

The first two arguments, as above, refer to a WSDL message declaration (within an operation declaration), and the third refers to a SWSO message type.

`msg_query` is a document query expression, containing variables, that extracts the needed information from in the knowledge base and binds the variables to that information. We do not specify a particular document query language in this document, but XQuery [*XQuery 1.0*] would be a natural candidate.

`kb_update_script` is a script in a knowledge base update language. Variables bound in the update script would correspond to variables mentioned in the message query, and would be bound to the query's return values before the script is executed.

5 Rules Ontology for Web Services (ROWS)

The preceding material of this section describes FLOWS, the First-Order Logic Ontology for Web Services, which is expressed in SWSL-FOL. To enable implementations in reasoning and execution environments based on logic-programming, we provide, in Appendix C, the Rules Ontology for Web Services (ROWS). ROWS is a (partial) translation of FLOWS into SWSL-Rules. The intent of each ROWS axiom is identical to what is explained above for the corresponding axioms of FLOWS. However, because SWSL-Rules is inherently less expressive than SWSL-FOL, some axioms in ROWS are weakened with respect to the corresponding axiom in SWSL-FOL. The strategies by which the axioms have been translated into SWSL-Rules are summarized in Section 5 below.

ROWS also defines several top-level classes, which correspond to similar classes in FLOWS. These classes are `Service`, `Process`, `AtomicProcess`, `Message`, and `Channel`.

Services and service descriptors. The `Service` class is defined by its descriptors and the process associated with the service. Service descriptors are defined in Section 2 and processes in Section 3.1.

```
      prefix xsd = "http://www.w3.org/2001/XMLSchema".
      Service[
        name               *=> xsd#string,
        author             *=> xsd#string,
        contactInformation *=> xsd#string,
        contributor        *=> xsd#string,
        description        *=> xsd#string,
        url                *=> xsd#string,
        identifier         *=> xsd#string,
        version            *=> xsd#string,
        releaseDate        *=> xsd#date,
        language           *=> xsd#string,
        subject            *=> xsd#string,
        trust              *=> xsd#string,
        reliability        *=> xsd#string,
        cost               *=> xsd#string
    ].
```

Some of the above string-datatypes may be replaced with more appropriate data types later. For instance, the type of the `cost` descriptor may be replaced with a type that supports currencies and even compelex arrangements such as payment plans.

Processes and IOPEs. In addition to the descriptors, the `Service` class has the `process` attribute, which specifies the process associated with the service. It is defined as follows:

```
      Service[
        process  *=> Process
      ].
```

where the `Process` class has the following attributes:

```
     Process[
       precondition *=> Formula,
       effect       *=> Formula,
       input        *=> ProcessInput,
       output       *=> ProcessOutput
    ].
```

Here `Formula` is a built-in class that consists of all reifications of formulas in SWSL-Rules and `ProcessInput` and `ProcessOutput` are classes that determine the structure of inputs and outputs of processes. They will be defined in a future release.

In addition, the `AtomicProcess` class of Section 3.1 is defined as a subclass of `Process`:

```
      AtomicProcess :: Process.
```

Messages and channels. The `Message` and the `Channel` classes are declared as follows:

```
      Message[
        type     *=> MessageType,
        body     *=> MessageBody,
        producer *=> AtomicProcess
      ].
      Channel[
        contents *=> Message,
        source   *=> Service,
        target   *=> Service
      ].
```

where the classes `MessageType` and `MessageBody`, which are used to specify the ranges of the attributes `type` and `body` in class `Message`, will be defined in a future release.

6 Background Materials

6.1 Ontology of the Process Specification Language

The semantics of concepts in FLOWS is formally axiomatized using ontology of ISO 18629 (Process Specification Language) [*Gruninger03a*], [*Gruninger03b*]. The complete set of axioms for the PSL Ontology can be found at PSL Ontology .

FLOWS adopts the basic ontological commitments of ISO 18629-11 (PSL-Core):

1. There are four kinds of entities required for reasoning about processes -- activities, activity occurrences, timepoints, and objects.
2. Activities may have multiple occurrences, or there may exist activities that do not occur at all.
3. Timepoints are linearly ordered, forwards into the future, and backwards into the past.
4. Activity occurrences and objects are associated with unique timepoints that mark the begin and end of the occurrence or object.

Some key predicates in PSL-Core include `activity(?a)` which holds when the value of `?a` is an activity in a given interpretation, and `occurrence_of(?occ, ?a)` which holds when the value associated with `?occ` is an occurrence (intuitively, an execution) of the activity associated with `?a`. PSL-Core also provides terms and predicates for describing chronological time and relating the activity occurrences to time (e.g., term `beginof(?occ)` corresponds to the time point at which `?occ` begins, and predicate `before(?t1,?t2)` holds if the value of `?t1` is chronologically before the value of `?t2`).

The axiomatization of FLOWS also requires the extensions in ISO 18629-12 (PSL Outer-Core):

1. Subactivity Theory

   The PSL Ontology uses the `subactivity` relation to capture the basic intuitions for the composition of activities. This relation is a discrete partial ordering, in which primitive activities are the minimal elements. The primary predicate that is axiomatized in this extension is `subactivity(?a1,?a2)`, which holds when activity `?a1` is component activity of activity `?a2`.

2. Occurrence Trees Theory

   The occurrence trees that are axiomatized in this core theory are partially ordered sets of activity occurrences, such that for a given set of activities, all discrete sequences of their occurrences are branches of the tree. An occurrence tree contains all occurrences of *all* activities; it is not simply the set of occurrences of a particular (possibly complex) activity.

3. Discrete State Theory

   This core theory introduces the notion of state (fluents). Fluents are changed only by the occurrence of activities, and fluents do not change during the occurrence of primitive activities. In addition, activities have preconditions (fluents that must hold before an occurrence) and effects (fluents that always hold after an occurrence).

4. Atomic Activities Theory

   This core theory axiomatizes intuitions about the concurrent aggregation of primitive activities. This concurrent aggregation is represented by the occurrence of concurrent activities, rather than concurrent activity occurrences. In particular, occurrences of atomic activities are non-decomposable (e.g., in the sense of database concurrency) -- the occurrence is not interleaved with other activity occurrences, and all aspects of the occurrence are completed.
5. Complex Activity Theory

   This core theory characterizes the relationship between the occurrence of a complex activity and occurrences of its subactivities. Occurrences of complex activities correspond to sets of occurrences of subactivities; in particular, these sets are subtrees of the occurrence tree. An activity tree consists of all possible sequences of atomic subactivity occurrences beginning from a root subactivity occurrence. In a sense, activity trees are a microcosm of the occurrence tree, in which we consider all of the ways in which the world unfolds in the context of an occurrence of the complex activity.

   Different subactivities may occur on different branches of the activity tree, so that different occurrences of an activity may have different subactivity occurrences or different orderings on the same subactivity occurrences. In this sense, branches of the activity tree characterize the nondeterminism that arises from different ordering constraints or iteration.

   An activity will in general have multiple activity trees within an occurrence tree, and not all activity trees for an activity need be isomorphic. Different activity trees for the same activity can have different subactivity occurrences. Following this intuition, this core theory does not constrain which subactivities occur.

6. Complex Activity Occurrence Theory

   Within the Complex Activity Theory, complex activity occurrences correspond to activity trees, and consequently occurrences of complex activities are not elements of the legal occurrence tree. The axioms of the Activity Occurrences core theory ensure that complex activity occurrences correspond to branches of activity trees. Each complex activity occurrence has a unique atomic root occurrence and each finite complex activity occurrence has a unique atomic leaf occurrence. A subactivity occurrence corresponds to a sub-branch of the branch corresponding to the complex activity occurrence. The primary relation is `subactivity_occurrence(?occ1,?occ2)`, which holds in an interpretation when the occurrence `?occ1` (which can be viewed as a set of one or more atomic occurrences in one branch of the execution tree) is a subset of the occurrence `?occ2`.

6.2 Knowledge Preconditions and Knowledge Effects

In order to characterize the inputs and outputs of Web services as knowledge preconditions and knowledge effects, FLOWS augments PSL with a treatment of knowledge. To do so, it appeals to the work of Scherl and Levesque who provided a means of encoding the knowledge of an agent in the situation calculus by adapting Moore's possible-world semantics for knowledge and action [*Scherl03*]. Scherl and Levesque achieve this by adding a *K* fluent to the situation calculus. Intuitively, *K(s',s)* holds iff when the agent is in situation *s*, she considers it possible to be in *s'*. Thus, a first-order formula *f* is *known* in a situation *s* if *f* holds in every situation that is K-accessible from *s*. For notational convenience, the following abbreviations are adopted. **Knows***(f, s)* is defined to be *forall s' K(s',s) ==> f[s']*, and **Knowswhether***(f,s)* is equivalent to **Knows***(f,s)* or **Knows***(not f,s)*. To define properties of the knowledge of agents they define restrictions such as reflexivity over the *K* fluent. The FLOWS axiomatization of these epistemic concepts can be found here.

Note: We assume *f* is a *situation-suppressed* formula (i.e. a situation formula whose situation terms are suppressed). *f[s]* denotes the formula that restores situation arguments in *f* by *s*.

7 Glossary

Activity
Activity. In the formal PSL ontology, the notion of activity is a basic construct, which corresponds intuitively to a kind of (manufacturing or processing) activity. In PSL, an activity may have associated *occurrences*, which correspond intuitively to individual instances or executions of the activity. (We note that in PSL an activity is not a class or type with occurrences as members; rather, an activity is an object, and occurrences are related to this object by the binary predicate `occurrence_of`.) The occurrences of an activity may impact fluents (which provide an abstract representation of the "real world"). In FLOWS, with each service there is an associated activity (called the "service activity" of that service). The service activity may specify aspects of the internal process flow of the service, and also aspects of the messaging interface of that service to other services.

Channel
Channel. In FLOWS, a channel is a formal conceptual object, which corresponds intuitively to a repository and conduit for messages. The FLOWS notion of channel is quite primitive, and under various restrictions can be used to model the form of channel or message-passing as found in web services standards, including WSDL, BPEL, WS-Choreography, WSMO, and also as found in several research investigations, including process algrebras.

FLOWS
F*irst-order* ***L****ogic* ***O****ntology for* ***W****eb* ***S****ervices*. FLOWS, also known as SWSO-FOL, is the first-order logic version of the Semantic Web Services Ontology. FLOWS is an extension of the PSL-OuterCore ontology, to incorporate the fundamental aspects of (web and other electronic) services, including service descriptors, the service activity, and the service grounding.

Fluent
Fluent. In FLOWS, following PSL and the situation calculii, a fluent is a first-order logic term or predicate whose value may vary over time. In a first-order model of a FLOWS theory, this being a model of PSL-OuterCore, time is represented as a discrete linear sequence of *timees*, and fluents has a value for each time in this sequence.

Grounding
Grounding. The SWSO concepts for describing service activities, and the instantiations of these concepts that describe a particular service activity, are *abstract* specifications, in the sense that they do not specify the details of particular message formats, transport protocols, and network addresses by which a Web service is accessed. The role of the *grounding* is to provide these more concrete details. A substantial portion of the grounding can be acheived by mapping SWSO concepts into corresponding WSDL constructs. (Additional grounding, e.g., of some process-related aspects of SWSO, might be acheived using other standards, such as BPEL.)

Message
Message. In FLOWS, a message is a formal conceptual object, which corresponds intuitively to a single message that is created by a service occurrence, and read by zero or more service occurrences. The FLOWS notion of message is quite primitive, and under various restrictions can be used to model the form of messages as found in web services standards, including WSDL (1.0 and 2.0), BPEL, WS-Choreography, WSMO, and also as found in several research investigations. A message has a *payload*, which corresponds intuitively to the body or contents of the message. In FLOWS emphasis is placed on the knowledge that is gained by a service occurrence when reading a message with a given payload (and the knowledge needed to create that message.

Occurrence
Occurence (of a service). In FLOWS, a service *S* has an associated FLOWS activity *A* (which generalizes the notion of PSL activity). An *occurrence* of *S* is formally a PSL occurrence of the activity *A*. Intuitively, this occurrence corresponds to an instance or execution (from start to finish) of the activity *A*, i.e., of the process associated with service *S*. As in PSL, an occurrence has a starting time time and an ending time.

PSL
Process Specification Language. The Process Specification Language (PSL) is a formally axiomatized ontology [*Gruninger03a*, *Gruninger03b*] that has been standardized as ISO 18629. PSL provides a layered, extensible ontology for specifying properties of processes. The most basic PSL constructs are embodied in PSL-Core; and PSL-OuterCore incorporates several extensions of PSL-Core that includes several useful constructs. (An overview of concepts in PSL that are relevant to FLOWS is given in Section 6 of the Semantic Web Services Ontology document.)

QName
Qualified name. A pair (*URI*, *local-name*). The *URI* represents a namespace and *local-name* represents a name used in an XML document, such as a tag name or an attribute name. In XML, QNames are syntactically represented as *prefix*:*local-name*, where *prefix* is a macro that expands into a concrete URI. See Namespaces in XML for more details.

ROWS
R*ules* ***O****ntology for* ***W****eb* ***S****ervices*. ROWS, also known as SWSO-Rules, is the rules-based version of the Semantic Web Services Ontology. ROWS is created by a relatively straight-forward, almost faithful, transformation of FLOWS, the First-order Logic Ontology for Web Services. As with FLOWS, ROWS incorporates fundamental aspects of (web and other electronic) services, including service descriptors, the service activity, and the service grounding. ROWS enables a rules-based specification of a family of services, including both the underlying ontology and the domain-specific aspects.

Service

(Formal) Service. In FLOWS, a service is a conceptual object, that corresponds intuitively to a web service (or other electronically accessible service). Through binary predicates a service is associated with various service descriptors (a.k.a. non-functional properties) such as Service Name, Service Author, Service URL, etc.; an *activity* (in the sense of PSL) which specifies intuitively the process model associated with the service; and a *grounding*.

Service contract

Describes an agreement between the service requester and service provider, detailing requirements on a service occurrence or family of service occurrences.

Service descriptor

Service Descriptor. This is one of several non-functional properties associated with services. The Service Descriptors include Service Name, Service Author, Service Contract Information, Service Contributor, Service Description, Service URL, Service Identifier, Service Version, Service Release Date, Service Language, Service Trust, Service Subject, Service Reliability, and Service Cost.

Service offer description

Describes an abstract service (i.e. not a concrete instance of the service) provided by a service provider agent.

Service requirement description

Describes an abstract service required by a service requester agent, in the context of service discovery, service brokering, or negotiation.

sQName

Serialized QName. A serialized QName is a shorthand representation of a URI. It is a macro that expands into a full-blown URI. sQNames are *not* QNames: the former are URIs, while the latter are pairs (*URI*, *local-name*). Serialized QNames were originally introduced in RDF as a notation for shortening URI representation. Unfortunately, RDF introduced confusion by adopting the term QName for something that is different from QNames used in XML. To add to the confusion, RDF uses the syntax for sQNames that is identical to XML's syntax for QNames. SWSL distinguishes between QNames and sQNames, and uses the syntax *prefix#local-name* for the latter. Such an sQName expands into a full URI by concatenating the value of *prefix* with *local-name*.

URI

Universal Resource Identifier. A symbol used to locate resources on the Web. URIs are defined by IETF. See Uniform Resource Identifiers (URI): Generic Syntax for more details. Within the IETF standards the notion of URI is an extension and refinement of the notions of Uniform Resource Locator (URL) and Relative Uniform Resource Locators.

8 References

[Berardi03]

Automatic composition of e-services that export their behavior. D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. In *Proc. 1st Int. Conf. on Service Oriented Computing (ICSOC)*, volume 2910 of *LNCS*, pages 43--58, 2003.

[Bernstein2000]

How can cooperative work tools support dynamic group processes? Bridging the specificity frontier. A. Bernstein. In *Proc. Computer Supported Cooperative Work (CSCW'2000)*, 2000.

[Bernstein2002]

Towards High-Precision Service Retrieval. A. Bernstein, and M. Klein. In *Proc. of the first International Semantic Web Conference (ISWC'2002)*, 2002.

[Bernstein2003]

Beyond Monotonic Inheritance: Towards Semantic Web Process Ontologies. A. Bernstein and B.N. Grosof (NB: authorship sequence is alphabetic). Working Paper, Aug. 2003. Available at: http://ebusiness.mit.edu/bgrosof/#beyond-mon-inh-basic.

[Bonner93]

Database Programming in Transaction Logic. A.J. Bonner, M. Kifer, M. Consens. *Proceedings of the 4-th Intl.~Workshop on Database Programming Languages*, C. Beeri, A. Ohori and D.E. Shasha (eds.), 1993. In Springer-Verlag Workshops in Computing Series, Feb. 1994: 309-337.

[Bonner98]

A Logic for Programming Database Transactions. A.J. Bonner, M. Kifer. Logics for Databases and Information Systems, J. Chomicki and G. Saake (eds.). Kluwer Academic Publishers, 1998: 117-166.

[Bruijn05]

Web Service Modeling Ontology (WSMO). J. de Bruijn, C. Bussler, J. Domingue, D. Fensel, M. Hepp, M. Kifer, B. König-Ries, J. Kopecky, R. Lara, E. Oren, A. Polleres, J. Scicluna, M. Stollberg. *DERI Technical Report.*

[BPML 1.0]

A. Arkin. Business Process Modeling Language. BPMI.org, 2002

[BPEL 1.1]

Business Process Execution Language for Web Services, Version 1.1 . S. Thatte, editor. OASIS Standards Specification, May 5, 2003.

[Bultan03]

Conversation specification: A new approach to design and analysis of e-service composition. T. Bultan, X. Fu, R. Hull, and J. Su. In *Proc. Int. World Wide Web Conf. (WWW)*, May 2003.

[Chang73]
Symbolic Logic and Mechanical Theorem Proving. C.L. Chang and R.C.T. Lee. Academic Press, 1973.

[Chen93]
HiLog: A Foundation for Higher-Order Logic Programming. W. Chen, M. Kifer, D.S. Warren. Journal of Logic Programming, 15:3, February 1993, 187-230.

[Chimenti89]
Towards an Open Architecture for LDL. D. Chimeti, R. Gamboa, R. Krishnamurthy, VLDB Conference, 1989: 195-203.

[deGiacomo00]
ConGolog, A Concurrent Programming Language Based on the Situation Calculus. G. de Giacomo, Y. Lesperance, and H. Levesque. Artificial Intelligence, 121(1--2):109--169, 2000.

[Fu04]
WSAT: A Tool for Formal Analysis of Web Services. X. Fu, T. Bultan, and J. Su. *16th International Conference on Computer Aided Verification (CAV)*, July 2004.

[Frohn94]
Access to Objects by Path Expressions and Rules. J. Frohn, G. Lausen, H. Uphoff. Intl. Conference on Very Large Databases, 1994, pp. 273-284.

[Gosling96]
*The Java language specification.*Gosling, James, Bill Joy, and Guy L. Steele. 1996. Reading, Mass.: Addison-Wesley.

[Grosof99a]
A Courteous Compiler From Generalized Courteous Logic Programs To Ordinary Logic Programs. B.N. Grosof. IBM Report included as part of documentation in the IBM CommonRules 1.0 software toolkit and documentation, released on http://alphaworks.ibm.com. July 1999. Also available at: http://ebusiness.mit.edu/bgrosof/#gclp-rr-99k.

[Grosof99b]
A Declarative Approach to Business Rules in Contracts. B.N. Grosof, J.K. Labrou, and H.Y. Chan. Proceedings of the 1st ACM Conference on Electronic Commerce (EC-99). Also available at: http://ebusiness.mit.edu/bgrosof/#econtracts+rules-ec99.

[Grosof99c]
DIPLOMAT: Compiling Prioritized Default Rules Into Ordinary Logic Programs (Extended Abstract of Intelligent Systems Demonstration). B.N. Grosof. IBM Research Report RC 21473, May 1999. Extended version of 2-page refereed conference paper appearing in Proceedings of the National Conference on Artificial Intelligence (AAAI-99), 1999. Also available at: http://ebusiness.mit.edu/bgrosof/#cr-ec-demo-rr-99b.

[Grosof2003a]
Description Logic Programs: Combining Logic Programs with Description Logic. B.N. Grosof, I. Horrocks, R. Volz, and S. Decker. Proceedings of the 12th International Conference on the World Wide Web (WWW-2003). Also available at: http://ebusiness.mit.edu/bgrosof/#dlp-www2003.

[Grosof2004a]
Representing E-Commerce Rules Via Situated Courteous Logic Programs in RuleML. B.N. Grosof. Electronic Commerce Research and Applications, 3:1, 2004, 2-20. Preprint version is also available at: http://ebusiness.mit.edu/bgrosof/#.

[Grosof2004b]
SweetRules: Tools for Semantic Web Rules and Ontologies, including Translation, Inferencing, Analysis, and Authoring. B.N. Grosof, M. Dean, S. Ganjugunte, S. Tabet, C. Neogy, and D. Kolas. http://sweetrules.projects.semwebcentral.org. Software toolkit and documentation. Version 2.0, Dec. 2004.

[Grosof2004c]
Hypermonotonic Reasoning: Unifying Nonmonotonic Logic Programs with First Order Logic. B.N. Grosof. http://ebusiness.mit.edu/bgrosof/#HypermonFromPPSWR04InvitedTalk. Slides from Invited Talk at Workshop on Principles and Practice of Semantic Web Reasoning (PPWSR04), Sep. 2004; revised Oct. 2004. Paper in preparation.

[Grosof2004d]
SweetPH: Using the Process Handbook for Semantic Web Services. B.N. Grosof and A. Bernstein. http://ebusiness.mit.edu/bgrosof/#SweetPHSWSLF2F1204Talk. Slides from Presentation at SWSL Meeting, Dec. 9-10, 2004. *Note: Updates the design in the 2003 Working Paper "Beyond Monotonic Inheritance: Towards Semantic Web Process Ontologies" and describes implementation.*

[Grosof2004e]
SweetDeal: Representing Agent Contracts with Exceptions using Semantic Web Rules, Ontologies, and Process Descriptions. B.N. Grosof and T. C. Poon. International Journal of Electronic Commerce (IJEC), 8(4):61-98, Summer 2004 Also available at: http://ebusiness.mit.edu/bgrosof/#sweetdeal-exceptions-ijec.

[Grosof2004f]

Semantic Web Rules with Ontologies, and their E-Business Applications. B.N. Grosof and M. Dean. Slides of Conference Tutorial (3.5-hour) at the 3rd International Semantic Web Conference (ISWC-2004). Available at: http://ebusiness.mit.edu/bgrosof/#ISWC2004RulesTutorial.

[Gruninger03a]
A Guide to the Ontology of the Process Specification Language. M. Gruninger. *Handbook on Ontologies in Information Systems.* R. Studer and S. Staab (eds.). Springer Verlag, 2003.

[Gruninger03b]
Process Specification Language: Principles and Applications. M. Gruninger and C. Menzel. *AI Magazine,* 24:63-74, 2003.

[Gruninger03c]
Applications of PSL to Semantic Web Services. M. Gruninger. *Workshop on Semantic Web and Databases. Very Large Databases Conference, Berlin.*

[Hayes04]
RDF Model Theory. Hayes, P. W3C, February 2004.

[Helland05]
Data on the Outside Versus Data on the Inside. P. Helland. *Proc. 2005 Conf. on Innovative Database Research (CIDR)*, January, 2005.

[Hull03]
E-Services: A Look Behind the Curtain. R. Hull, M. Benedikt, V. Christophides, J. Su. *Proc. of the ACM Symp. on Principles of Database Systems (PODS)*, San Diego, June, 2003.

[Kifer95]
Logical Foundations of Object-Oriented and Frame-Based Languages, M. Kifer, G. Lausen, J. Wu. Journal of ACM, 1995, 42, 741-843.

[Kifer04]
A Logical Framework for Web Service Discovery, M. Kifer, R. Lara, A. Polleres, C. Zhao. Semantic Web Services Workshop, November 2004, Hiroshima, Japan.

[Klein00a]
Towards a Systematic Repository of Knowledge About Managing Collaborative Design Conflicts. Klein, Mark. 2000. Proceedings of the Conference on Artificial Intelligence in Design. Boston, MA, USA.

[Klein00b]
A Knowledge-Based Approach to Handling Exceptions in Workflow Systems. Klein, Mark, and C. Dellarocas. 2000. Computer Supported Cooperative Work: The Journal of Collaborative Computing 9:399-412.

[Lloyd87]
Foundations of logic programming (second, extended edition). J. W. Lloyd. Springer series in symbolic computation. Springer-Verlag, New York, 1987.

[Lindenstrauss97]
Automatic Termination Analysis of Logic Programs. N. Lindenstrauss and Y. Sagiv. International Conference on Logic Programming (ICLP), 1997.

[Maier81]
Incorporating Computed Relations in Relational Databases. D. Maier, D.S. Warren. SIGMOD Conference, 1981: 176-187.

[Malone99]
Tools for inventing organizations: Toward a handbook of organizational processes. T. W. Malone, K. Crowston, J. Lee, B. Pentland, C. Dellarocas, G. Wyner, J. Quimby, C. Osborne, A. Bernstein, G. Herman, M. Klein, E. O'Donnell. *Management Science*, 45(3), pages 425--443, 1999.

[McIlraith01]
Semantic Web Services. *IEEE Intelligent Systems*, Special Issue on the Semantic Web, S. McIlraith, T.Son and H. Zeng. 16(2):46--53, March/April, 2001.

[Milner99]
Communicating and Mobile Systems: The π-Calculus. R. Milner. Cambridge University Press, 1999.

[Narayanan02]
Simulation, Verification and Automated Composition of Web Services. S. Narayanan and S. McIlraith. In *Proceedings of the Eleventh International World Wide Web Conference (WWW-11)*, May, 2002.

[Ontobroker]
Ontobroker 3.8. Ontoprise, GmbH.

[OWL Reference]
OWL Web Ontology Language 1.0 Reference. Mike Dean, Dan Connolly, Frank van Harmelen, James Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. W3C Working Draft 12 November 2002. Latest version is available at http://www.w3.org/TR/owl-ref/.

[OWL-S 1.1]
OWL-S: Semantic Markup for Web Services. David Martin, editor. Technical Overview (associated with OWL-S Release 1.1).

[Papazoglou03]
Service-Oriented Computing: Concepts, Characteristics and Directions. M.P. Papazoglou. Keynote for the 4th International Conference on Web Information Systems Engineering (WISE 2003), December 10-12, 2003.

[Perlis85]
Languages with Self-Reference I: Foundations. D. Perlis. Artificial Intelligence, 25, 1985, 301-322.

[Preist04]
A Conceptual Architecture for Semantic Web Services, C. Preist, 1993. In Proceedings of Third International Semantic Web Conference, Nov. 2004: 395-409.

[Reiter01]
Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems. Raymond Reiter. MIT Press. 2001

[Scherl03]
Knowledge, Action, and the Frame Problem. R. B. Scherl and H. J. Levesque. *Artificial Intelligence*, Vol. 144, 2003, pp. 1-39.

[Singh04]
Protocols for Processes: Programming in the Large for Open Systems. M. P. Singh, A. K. Chopra, N. V. Desai, and A. U. Mallya. *Proc. of the 19th Annual ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Vancouver, October 2004.

[SWSL Requirements]
Semantic Web Services Language Requirements. B. Grosof, M. Gruninger, et al, editors. White paper of the Semantic Web Services Language Committee.

[UDDI v3.02]
Universal Description, Discovery and Integration (UDDI) protocol. S. Thatte, editor. OASIS Standards Specification, February 2005.

[VanGelder91]
The Well-Founded Semantics for General Logic Programs. A. Van Gelder, K.A. Ross, J.S. Schlipf. Journal of ACM, 38:3, 1991, 620-650.

[WSCL 1.0]
Web Services Conversation Language (WSCL) 1.0. A. Banerji et al. W3C Note, March 14, 2002.

[WSDL 1.1]
Web Services Description Language (WSDL) 1.1. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. W3C Note, March 15, 2001.

[WSDL 2.0]
Web Services Description Language (WSDL) 2.0 -- Part 1: Core Language. R. Chinnici, M. Gudgin, J.-J. Moreau, J. Schlimmer, and S. Weerawarana. W3C Working Draft, August 3, 2004.

[WSDL 2.0 Primer]
Web Services Description Language (WSDL) Version 2.0 -- Part 0: Primer. D. Booth, C. Liu, editors. W3C Working Draft, 21 December 2004.

[WS-Choreography]
Web Services Choreography Description Language Version 1.0. N. Kavantzas, D. Burdett, et. al., editors. W3C Working Draft, December 17, 2004.

[XSLT]
XSL Transformations (XSLT) Version 1.0. J. Clark, editor. W3C Recommendation, 16 November 1999.

[XQuery 1.0]
XQuery 1.0: An XML Query Language. S. Boag, D. Chamberlin, et al, editors. W3C Working Draft 04 April 2005.

[Yang02]
Well-Founded Optimism: Inheritance in Frame-Based Knowledge Bases. G. Yang, M. Kifer. Intl. Conference on Ontologies, DataBases, and Applications of Semantics for Large Scale Information Systems (ODBASE), October 2002.

[Yang03]
Reasoning about Anonymous Resources and Meta Statements on the Semantic Web. G. Yang, M. Kifer. Journal on Data Semantics, Lecture Notes in Computer Science 2800, Springer Verlag, September 2003, 69-98.

[Yang04]
FLORA-2 User's Manual. G. Yang, M. Kifer, C. Zhao, V. Chowdhary. 2004.

Appendices

SWSF Application Scenarios

Version 1.0

Authors:
Steve Battle (Hewlett Packard) Abraham Bernstein (University of Zurich) Harold Boley (National Research Council of Canada) Benjamin Grosof (Massachusetts Institute of Technology) Michael Gruninger (NIST) Richard Hull (Bell Labs Research, Lucent Technologies) Michael Kifer (State University of New York at Stony Brook) David Martin (SRI International) Sheila McIlraith (University of Toronto) Deborah McGuinness (Stanford University) Jianwen Su (University of California, Santa Barbara) Said Tabet (The RuleML Initiative)

Abstract

This document presents several use cases that illustrate SWSO -- the Semantic Web Services Ontology, and SWSL -- the Semantic Web Services Language developed by the committee.

Status of this document

This is one of four documents that make up the initial report of the Semantic Web Services Language Committee of the Semantic Web Services Initiative. The report defines the Semantic Web Services Framework (SWSF).

History of publication at http://www.daml.org/services/swsl/report/applications/:

- v. 0.9: April 6, 2005
- v. 0.91: April 13, 2005
- v. 0.92: April 25, 2005
- v. 0.93: May 5, 2005

History of publication at http://www.daml.org/services/swsf/applications/:

- v. 1.0: May 9, 2005

Table of contents

1 Introduction

This document is part of the technical report of the Semantic Web Services Language (SWSL) Committee of the Semantic Web Services Initiative (SWSI). The overall structure of the report is described in the document titled Semantic Web Services Framework Overview. The present document discusses a number of use cases that illustrate the Semantic Web Services Ontology (SWSO), which is defined in a separate document, and the Semantic Web Services Language (SWSL), which is also described separately.

Section 2 presents use cases that illustrate various aspects of SWSO with emphasis on the process model. Section 3 considers a particular use case, the Amazon.com service, and shows how SWSO and SWSL can be used to describe this service. Section 4 focuses on the problem of Web service discovery and shows how services, user goals, mediators, and the discovery engine itself can be defined in SWSL-Rules, which is a sublanguage of SWSL. Section 5 presents use cases for policy specification in e-commerce and shows how these cases can be specified in SWSL-Rules. Finally, Section 6 uses SWSL-Rules to illustrate the need for non-monotonic inheritance and overriding in domain-specific service ontologies.

2 Use Cases Illustrating the Conceptual Model

We introduce here several examples, which are useful in illustrating selected aspects of the Process Model portion of the Conceptual Model underlying the FLOWS Ontology. Section 2.1 focuses on human-machine interactions, and describes a hypothetical on-line bookseller. The example illustrates aspects of atomic processes, fluents, and messages. Section 2.2 focuses on machine-machine interaction, and illustrates the use of channels, and the possible FLOWS-Core extensions for Guarded Automata and Meta-Server. Finally, Section 2.3, illustrates the flexibility of the conceptual model, by showing briefly how it can be applied to specify services coordination in a telecommunications context.

2.1 Illustration of Atomic Processes and Messages

This example focuses on a web service Acme_Book_Sales, that provides support for the external web presence of the hypothetical Acme book selling business. We focus primarily on the occurrences of the Acme_Book_Sales service that interact with end-users (modeled abstractly as service occurrences), and mention only at a high level some aspects of the interaction of occurrences of this service with other services (e.g., to arrange for credit card charges and shipping). In the following we describe (i) three of the domain-specific fluents manipulated by occurrences of Acme_Book_Sales, (ii) four atomic processes used by Acme_Book_Sales, (iii) four of the types of message that Acme_Book_Sales can send or receive, and (iv) finally a typical process flow in occurrences of Acme_Book_Sales.

Note that the domain-specific fluents described here might be accessed by services operated by Acme other than the service Acme_Book_Sales, and perhaps accessed by services operated by enterprises other than Acme. (E.g., a shipper might have read or read/write access on some of these fluents.) Likewise, the atomic processes used in the service Acme_Book_Sales might be used in the specification of other services (supported by Acme or other enterprises).

The description here is intended to illustrate certain aspects of the process model of the SWSL conceptual model, and is not intended to provide a complete specification of the Acme_Book_Sales service. (In any event, recall that SWSL can support both complete and incomplete specifications of services.) We focus on three domain-specific fluents:

```
          Book_info[
              ISBN => xsd#string,
              Title => xsd#string,
              Author => xsd#string
          ].
```

For each book that Acme is selling, this fluent holds a record (i,t,a) for each author of the book, where i is the ISBN number, t is the title, and a is the author name.

(For simplicity of exposition, this is not, in the parlance of relational databases, in Third Normal Form.) Note that some books listed in `Book_info` may not be in Acme's stock at a given time.

```
    Book_inventory[
        ISBN => xsd#string
        warehouse_id => xsd#string,
        quantity_on_hand => xsd#integer
    ].
```

We assume that the Acme company operates multiple warehouses, and that copies of a given book might be available from one or more of these warehouses.

```
    Book_reservation[
        ISBN => xsd#string,
        user_id => xsd#string,
        warehouse_id => xsd#string
    ].
```

When a user puts a book in their shopping cart, this corresponds to a commitment by Acme that it will actually ship the book if the user commits to the purchase (and the payment details are successful). To this end, Acme has to reserve a copy of the book while the user is deciding whether to commit. The fluent `Book_reservations` is intended to hold triples of the form (u, i,w), that will indicate that a copy of the book with ISBN number i is being held at warehouse w for user u.

We now overview four atomic processes used by Acme_Book_Sales. The description here is somewhat informal and is intended to provide an intuitive understanding, not a formal specification.

```
    book_search[
        input => book_descriptor[ISBN => xsd#string,
                                 title => xsd#string,
                                 authors => Person,
                                 keywords => xsd#string],
        output => book_record[ISBN => xsd#string,
                              title => xsd#string,
                              authors => xsd#string],
        effect => Formula  // focused on output only
    ].
```

This process supports searches against the full catalog of books sold by the bookseller. The input argument for this may include precise information such as ISBN, title, author, and/or may include more open-ended information such as key-words (in the title), partial author names, etc. This atomic process returns a (possibly empty) list of books that satisfy the search criteria. There are no side-effects.

```
    book_reserve[
        input => reservation_input[user_id => xsd#integer,
                                   ISBN => xsd#string],
        output => reservation_output[warehouse => xsd#string,
                                     expected_ship_date => xsd#date],
        effect => Formula
           // the effect is supposed to be: if the book is available, then
           // decrement quantity of book for this warehouse,
           // insert record into the "reserved" relation
           // if not, then no effect
    ].
```

Intuitively, this process corresponds to a user putting a book into her shopping cart. In this case, Acme commits that it can in fact ship the book. So it decrements the quantity of copies available from one of the warehouses, and puts a record into the `Book_reservations` fluent.

```
    reservation_cancel[
        input => cancellation_result[user_ID => xsd#integer, ISBN => xsd#string],
        output => success_flag_type,
        effect => Formula
           // Intended effect:
           //    If the book is reserved for this user
           //    remove appropriate records from "reserved" relation,
           //    increment quantity of book for appropriate warehouse,
           //    set success_flag to true;
           //    if not set success_flag to false;
    ].


    book_shipment[
        input => ship_type[user_ID => xsb#integer,
                           ISBN => xsd#string],   // list of ISBNs
        output => xsb#integer,                      // confirmation_#;
        effect => Formula
           // Intended effect: Assuming no exceptions,
           //    remove appropriate records from "reserved" relation,
           //    (possibly, move a copy of book to loading dock);
    ].
```

These atomic processes deal with "clean up" concerning reserved books. `reservation_cancel` effectively cancels out the reservation of the book, and `book_shipment` records the fact that the book should really be shipped (and might include physically moving a copy of the book to a loading dock).

So far we have focused on how occurrences of Acme_Book_Sales "interacts" with domain-specific fluents. What about interaction with other services, and with humans (which are modeled as services)? Some of the messages that Acme_Book_Sales can send and receive include the following. As before, the description of argument types is for intuitive purposes only.

```
    book_search_request(User_ID: string, search_criteria: search_criteria_type)
```

A user can send a message of this type to an occurrence of Acme_Book_Sales. The single argument is an amalgam of whatever search criteria the user has specified. In practice we would expect this argument to be assembled by the user interface code running on the user's web browser and Acme's web server, based on the inputs of the user into her browser.

```
    book_search_response(ISBN: string, title: string, list_of [author: string])
```

This message can be sent by an occurrence of Acme_Book_Sales to a user, indicating information about zero or more books that Acme is selling.

```
    shopping_cart_request(user_ID: string, ISBN: string)
```

This message can be sent by a user to an occurrence of Acme_Book_Sales, indicating that the user wants the specified book to be put into her shopping cart.

```
    book_availability_response(ISBN: string, title: string,
                               list_of [author: string], ship_date: date)
```

This message can be sent by an occurrence of Acme_Book_Sales to a user, indicating that the indicated book is available, and can be shipped to the user on ship_date.

Acme_Book_Sales is capable of sending and receiving several other types of message. This includes messages from the user to commit to a purchase, and to put in credit card information, and messages back to the user to confirm various operations. It includes messages between occurrences of Acme_Book_Sales and banking services, and between Acme_Book_Sales and shipping services.

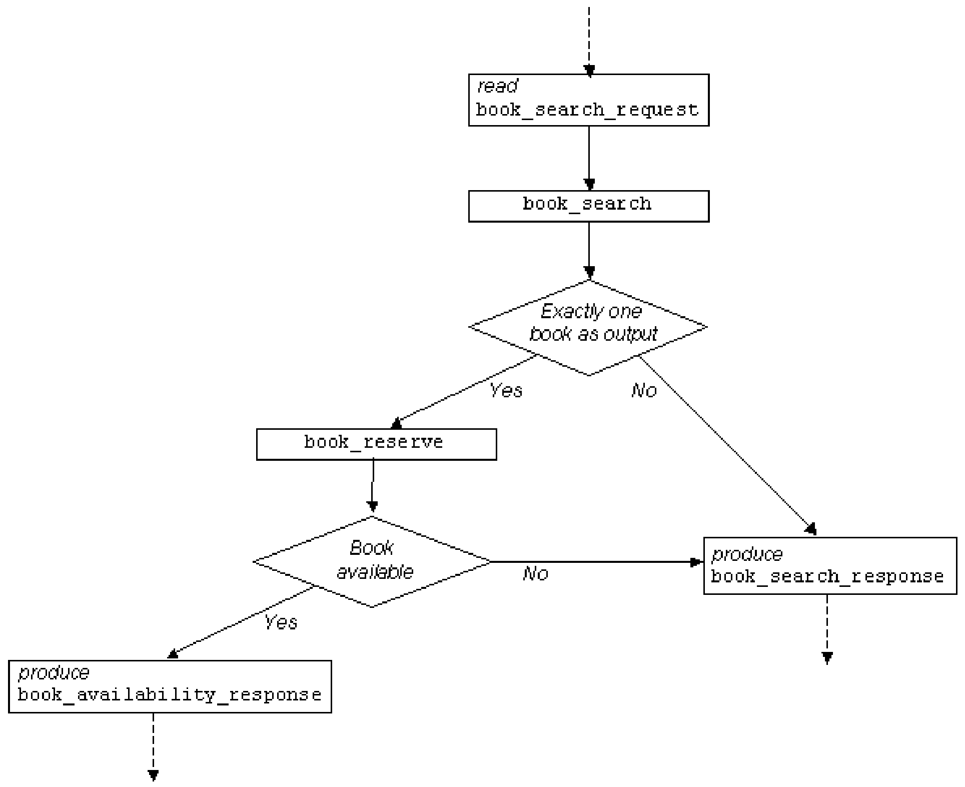

Figure 2.1: Part of process flow for Acme_Book_Sales service activity

Example 2.1(a): Atomic Processes and Internal Process Model. We now briefly discuss a partial specification of the permitted process flow of atomic process occurrences in interpretations of the theory for this application domain. Figure 2.1 illustrates a portion of this process flow, using a flow-chart paradigm. In terms of a formal domain theory, this (or something equivalent) could be specified using the Control Constructs extension of FLOWS-Core, or by using more primitive PSL-OuterCore predicates such as `soo_precedes`.

A notable aspect of the process flow indicated in Figure 2.1 is that there is not necessarily a one-to-one correspondence between messages being sent by a user and invocations of atomic processes in the occurrence of Acme_Book_Sales. Specifically, after an Read_Message occurrence of kind `book_search_request`, there will be an occurrence in Acme_Book_Sales of the atomic process `book_search`, with the appropriate input argument. If zero or more than one books is output from this atomic process occurrence, then there is a Produce_Message occurrence that creates a message of type `book_search_response`. (Presumably, this is followed by an appropriate Read_Message occurrence in the end-user.) However, if exactly one book was returned by the `book_search` occurrence, then in the occurrence of Acme_Book_Sales there is an occurrence of `book_reserve`. Intuitively, this is a pro-active step by Acme_Book_Sales, in order to provide for the user information about availability and possible shipping date of this book. If the book is available, then in Acme_Book_Sales there is a Produce_Message occurrence that produces a message of type `book_availability_response`. (If the book is not available, then in Acme_Book_Sales there would be a Produce_Message occurrence that creates a message of type `book_search_response`, or perhaps of a different type, that includes the fact that the book is currently unavailable.)

(We note that if the end-user has a Produce_Message occurrence that creates a message of type `book_search_request` then the response message may have one of two kinds. We assume that the Acme web server and the user's web browser software will present these to the user in an appropriate manner.)

It is possible, using the knowledge pre-conditions and conditional effects portion of the FLOWS-Core ontology, to precisely characterize the flow of information between atomic process occurrences in an occurrence of the Acme_Book_Sales service. One application of this capability would be to support a form of de-bugging, in which one could verify that a domain theory for a service such as Acme_Book_Sales provides for the flow of information needed to support a desired objective. Another application of this capability is presented in the next example.

Example 2.1(b): Inferring Message Production and Reads We now consider a variation, in which the domain theory *T* for Acme_Book_Sales includes two parts:

- ***T1:*** An explicit description of the domain-specific atomic processes that Acme_Book_Sales can perform along with constraints on their sequencing, relationship to domain-specific fluents, etc.
- ***T2:*** A family of generic constraints that state properties such as that if an atomic process (e.g., `book_search`) requires a certain form of input (e.g., ISBN# and/or book title and/or ...), and there is an occurrence of that process, then a message holding the required information in its payload must have been produced (e.g., the the user) and read by some Read_Message occurrence in the occurrence of Acme_Book_Sales.

Here, the family *T2* of constraints might be used in a variety different contexts involving services that embody many different variations of Acme_Book_Sales.

Imagine now that we want to study the possible occurrences satisfying the theory *T1* union *T2*, in which a user's goal is to buy a book with a given ISBN#. Suppose further that *o* is an occurrence of the overall system in which the user's goal is achieved. From the constraints in *T1* one can infer properties of *o* concerning the actual ordering of the domain-specific atomic process occurrences, the input and output values associated to those occurrences, and the values of the domain-specific fluents. Furthermore, one can use *T2*, along with axioms from the FLOWS-Core ontology, to infer that certain messages must have been produced and read in *o*, and infer some of the characteristics of the payloads of those messages. Thus, assuming that an appropriate family *T2* of constraints is specified, one can specify the full behavior of a web service by specifying only the domain-specific atomic processes involved that service. In particular, the message-handling atomic processes of the service can be inferred, and do not have to be specified explicitly.

2.2 Illustration of Channels, Meta-Service, Guarded Automata

We now describe the *Store-Warehouse-Bank* example, which is based loosely on an example in [*Bultan03*] (see also [*Hull03*]). As suggested in Figure 2.2 below, we assume the existence of three entities, namely a (bricks-and-mortar) Store, a Warehouse, and a Bank. (Unlike the example of Subsection 2.1, the store here is assumed to maintain a physical inventory on premises.) For each entity we focus on one of the several web services that it might perform; these have the following names:

`Store_inventory_maintenance`
: *A "back-office" service, that attempts to ensure that the for each item sold by the store, the available stock at the store remains above some threshold.*

`Warehouse_order_fulfillment`
: *This service running at the Warehouse responds to orders from the Store. As part of its operation it charges an account of the Store held at the Bank.*

`Bank_account_maintenance`
: *This service running at the Warehouse responds to orders from the Store. As part of its operation it charges an account of the Store held at the Bank.*

In the following we describe, for the three services, (a) the domain-specific fluents they can access and manipulate, (b) the atomic domain-specific atomic processes in them, and (c) the kinds of messages that they can create, read, and destroy.

(FLOWS-Core does not consider parameter typing, but we include types here as an intuitive convenience. The use of structured values in message parameters, and atomic process outputs, can in principle be provided in extensions of FLOWS-Core, such as Relation-valued Parameters and XML-valued Parameters; see Section 3.7.2.)

For this example, we assume the following service-specific fluents and relations:

`Bank_accounts(account_id:int, owner_id:int, balance:int)`
: *This fluent holds data on bank accounts. It is accessed and maintained exclusively by some services associated with the Bank (including `Bank_account_maintenance`).*

`Store_inventory(item_id:int, quantity:int)`
: *This fluent holds data on inventory held by the store. It is accessed and maintained exclusively by some services associated with the Store (including `Store_inventory_maintenance`).*

`Warehouse_inventory(item_id:int, quantity:int)`
: *This fluent holds data on inventory held by the warehouse. It is accessed and maintained exclusively by some services associated with the Warehouse (including `Warehouse_order_fulfillment`. (For simplicity, we assume that the Store and Warehouse use the same numbering system.)*

`Goods_in_transit(shipment_id:int, shipping_date:date, expected_delivery_date:date, transport_content_list_id:int)`
: *This fluent holds data on goods that are being shipped (from the Warehouse to the Store). For this example, we assume that it can be viewed by some services associated with the Warehouse and the Store. (In practice, this fluent might be maintained by yet another service, e.g., operated by an entity Shipper. In that case, services associated with the Store and Warehouse might "read" information in `Goods_in_transit` by exchanging messages with a service operated by Shipper.)*

`Transport_content_lists(transport_content_list_id:int, item_id:int, quantity:int)`
: *This relation holds the contents of shipments. (The approach to modeling taken by this fluent is suggestive of how the extension Relation-Valued Parameters on FLOWS-Core might be constructed.) It can be accessed by the same services that can access `Good_in_transit`.*

`Wholesale_order_content_lists(wholesale_order_id:int, item_id:int, quantity:int)`
: *Similar to `Transport_content_lists`, this relation holds the contents of orders (made by the Store against the Warehouse). This fluent can be read by some services associated with the Store and the Warehouse. (The reader may wonder why this is not a fluent. The answer is that the association between `wholesale_order_id`'s and the `item_id`,`quantity` pairs need not vary over time -- it can be viewed as fixed for the duration of the execution of the overall system being modeled. In particular, a given `wholesale_order_id` will be used in exactly one message payload, and so there is no need to override the `item_id`,`quantity` pairs associated to it.)*

At least the following kinds of domain-specific atomic processes can be performed by the various services.

`Create_account(owner_id:int, account_id:int, initial_balance:int)`
`Modify_account_balance(account_id:int, amount:int)`
: *These have the expected semantics, and can be performed by occurrences of `Bank_account_maintenance` (and perhaps other services associated with the Bank).*

`Transfer_funds(source_account_id:int, target_bank_id:int, target_account_id:int, amount)`
: *This atomic activity is performed by occurrences of `Bank_account_maintenance` (and perhaps other services associated with the Bank). The source account must be maintained by the Bank service (the target might be another bank), and the source account must have enough funds to cover the transfer.*

`Send_shipment_from_warehouse(shipment_id:int, transport_content_list_id:int, ship_date:date)`
: *This atomic activity is performed by occurrences of the `Warehouse_order_fulfillment` service. It corresponds intuitively to the situation where a physical shipment is created by the warehouse (for the store). The primary pre-condition is that there is enough inventory in `Warehouse_inventory`; the conditional effect is to update `Warehouse_inventory`, `Goods_in_transit` and `Transport_contents_lists` appropriately. In practice, occurrences of this atomic activity will occur after a message is received from the Store, requesting a shipment of goods.*

`Receive_shipment_to_store(shipment_id:int, transport_content_list_id:int, receive_date:date)`
: *This atomic activity is performed by occurrences of the `Store_inventory_maintenance` service. It corresponds intuitively to the situation where a physical shipment is received by the store (from the warehouse). One pre-condition is that there is a record in `Goods_in_transit` with this `shipment_id`. (Additional pre-conditions might be based on dates.) The primary conditional effect is that the `Store_inventory` and `Goods_in_transit` are updated appropriately.*

We mention two additional atomic domain-specific activities that are associated with the Store and Bank, respectively, but which do not arise in the services `Store_inventory_maintenance` or `Warehouse_order_fulfillment`.

`Sell_to_buyer(buyer_id:int, retail_order_id:int, price:int)`
: *This atomic activity corresponds intuitively to the situation where an individual makes a purchase at the store. As with orders against the Warehouse, an additional fluent `Retail_order_content_lists(retail_order_id:int, item_id:int, quantity:int)` can be maintained to hold the list of goods involved in a retail order. The pre-condition*

on this atomic activity is that there is sufficient inventory at the Store, and the conditional effect is that the `Store_inventory` is updated appropriately.

`Receive_shipment_to_warehouse(shipment_id:int, transport_content_list_id:int, ship_date:date)`
: *This atomic activity corresponds intuitively to the situation where a physical shipment is received by the warehouse (from a service not discussed in the example). The conditional effect is to update `Warehouse_inventory` appropriately.*

Finally, we turn to the kinds of messages that the three services can exchange.

`request_account_creation(store_id:int; account_amount:int)`
: *Intuitively, this message requests establishment of an account by the Bank for the Store in the given amount.*

`acknowledge_account(account_id:int, account_balance:int)`
: *Intuitively, this message acknowledges the establishment of an account by the Bank for the Store, and provides the (newly created) account_id, along with current balance.*

`account_balance_inquiry(store_id:int; account_id:int)`
: *Intuitively, this message requests the current balance of a bank account.*

`account_balance_report(account_id:int, account_balance:int)`
: *Intuitively, this message gives the current balance in an account.*

`place_order(order_id:int, store_id:int, wholesale_order_id:int, bank_account_id:int, stop_attempt_date:date)`
: *Intuitively, this message places an order by the Store against the Warehouse. It will fail if there is not sufficient inventory in the Warehouse to fill the order. In that case, the Store wants the Warehouse to keep trying to fill the order until `stop_attempt_date`. (The list of things ordered is found in the fluent `Wholesale_order_contents_lists`, associated with the value of `wholesale_order_id`; see Example 2.2(a) below.)*

`fulfill_order_commitment(order_id:int; shipment_id:int)`
: *Intuitively, this message indicates that an order against the Warehouse will be filled, and provides the shipment_id. (Further information on the shipment can be obtained from the fluents `Goods_in_transit` and `Transport_contents_lists`.)*

`reject_order(order_id:int)`
: *Intuitively, this message indicates that an order against the Warehouse cannot be filled.*

`bill_for_goods(account_id:int; order_id:int, bill_amount:int, target_bank_id:int, target_account_id:int)`
: *Intuitively, this message indicates that the Warehouse is sending a bill to the Bank for payment from the specified account. Here the value of `target_account_id` is intended to be a bank account of the Warehouse; it might be maintained by the Bank highlighted in this example or by some other bank (as indicated by the value of `target_bank_id`).*

`payment_for_goods(account_id:int; order_id:int, payment_amount:int target_account_id:int)`
: *Intuitively, this message indicates that the Bank is making a payment of the specified amount to the Warehouse.*

Example 2.2(a): Relation-valued parameters. Using the above we can illustrate concretely the approach described in Subsection 3.7.2 for supporting in FLOWS-Core parameters having the form of relations (rather than having form essentially equivalent to single objects or scalars). Consider the message type `place_order`, whose third argument is `wholesale_order_id`. Speaking intuitively, to understand the information conveyed by the third argument, one must look at the relation `Wholesale_order_content_lists`(`wholesale_order_id`, `item_id`, `quantity`). In particular, a given value *W* for `wholesale_order_id` will be associated by `Wholesale_order_content_lists` with a set of `item_id`,`quantity` pairs; these will correspond to the set of items (with quantities) that the Store is ordering.

In a similar manner, the output parameter `transport_content_list_id` of atomic process `Send_shipment_from_warehouse` identifies a set of item-quantity pairs, according to the relation `Transport_content_lists`.

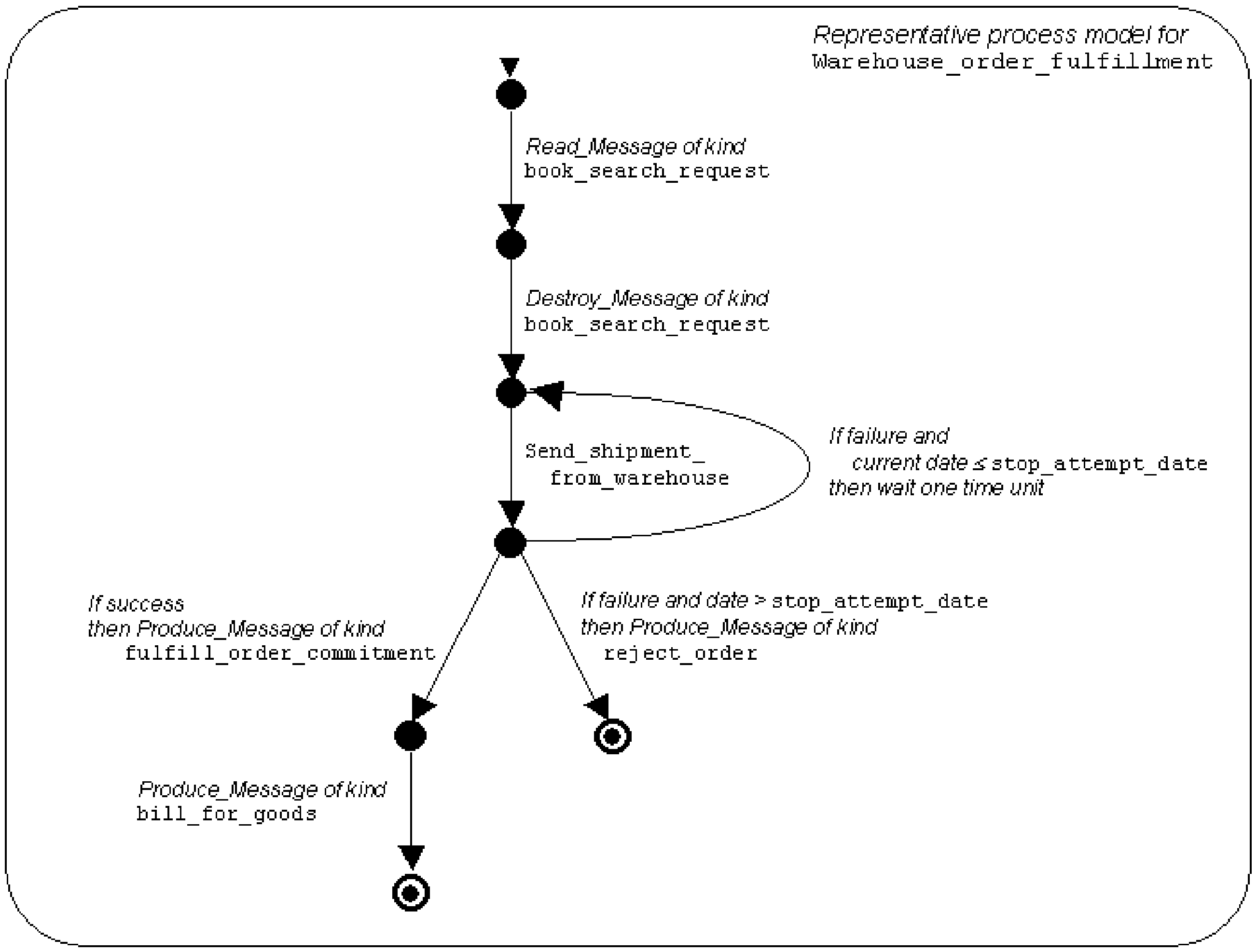

Figure 2.2: Informal specification of representative process model for `Warehouse_order_fulfillment`, using paradigm of Guarded Automaton

Example 2.2(b): Guarded Automata. We now consider one approach to formally specifying a representative process flow for the activity associated with service

`Warehouse_order_fulfillment`. This is illustrated in Figure 2.2, which shows informally a Guarded Automaton that specifies a possible behavior. This follows the spirit of the potential PSL extension of FLOWS-Core described in Subsection 3.7.6.

Intuitively, a typical flow of activity for (occurrences of) this service could be as follows:

1. At the beginning, there would be a Read_Message atomic process occurrence of kind `place_order`, followed by a Destroy_Message occurrence (which destroys the message just read).
2. An occurrence of atomic process `Send_shipment_from_warehouse` is next. Note that the automaton state reached after this occurrence has three out-edges. These are guarded by conditions, which in this case are disjoint. (In general, the conditions of edges from a given state of a guarded automaton may be overlapping, which models a form of non-determinism.)
3. If this atomic process occurrence fails (e.g., because the warehouse does not have sufficient inventory) then the occurrence of `Warehouse_order_fulfillment` will include additional occurrences of atomic process `Send_shipment_from_warehouse`, if the current time precedes the `stop_attempt_date` specified as an argument of the `place_order` message that was initially read.
4. If there is a successful occurrence of `Send_shipment_from_warehouse`, then there is a Produce_Message occurrence that creates a message of type `fulfill_order_commitment` (intended for the Store), and there is a Produce_Message occurrence that creates a message of type `bill_for_goods` (intended for the Bank).
5. If there is no successful occurrence of `Send_shipment_from_warehouse`, then there is a Produce_Message occurrence that creates a message of type `reject_order`.

Can a single occurrence of `Warehouse_order_fulfillment` accommodate many orders, or in other words, can it include many occurrences of the Read_Message atomic process of kind `book_search_request`? As specified in Figure 2.2, this is not possible; rather, a new occurrence of the service `Warehouse_order_fulfillment` will be needed for each message of type `place_order` coming from the Store. An alternative design for `Warehouse_order_fulfillment` would be to add transitions from the two final states shown in Figure 2.2, which point back to the start state. This would enable a form of sequential handling of order requests. A problem with this design is that it may take a long period of time to process a single order (in the case where there is looping on attempts to have successful execution of `Send_shipment_from_warehouse`). The next example describes how the use of a meta-service can enable a better design.

Example 2.2(c): Meta-Service In Subsection 3.7.1 above a potential extension for FLOWS-Core is described, in which "Servers" are used to support the orderly creation of occurrences of a given service. We illustrate that notion here, in the context of the Store-Warehouse-Bank example. In particular, we assume now that are Formal Servers *Store_server*, *Warehouse_server*, and *Bank_server* associated with the Store, Warehouse, and Bank.

We assume that `server_service`(*Warehouse_server*, *Warehouse_order_fulfillment*) holds. The fluent `server_service` might also associate other services with *Warehouse_server*, e.g., for making orders against manufacturing enterprises, for managing its own bank account, etc.

Suppose now that the service `Warehouse_order_fulfillment` has been specified so that an occurrence of this service is focused on fulfilling a single order placed by the Store (or other entities), basically as in Figure 2.2. If we use *Warehouse_server*, then the messages of type `place_order` will be read and destroyed by `Warehouse_server`. When such a message is read, then an occurrence of service `Warehouse_order_fulfillment` will come into existence. It would be typical for this new occurrence to start with a Read_Message occurrence, which reads a message produced by `Warehouse_server` (which might be essentially a copied version of the `place_order` message.) The new occurrence of `Warehouse_order_fulfillment` could then operate as specified in Figure 2.2.

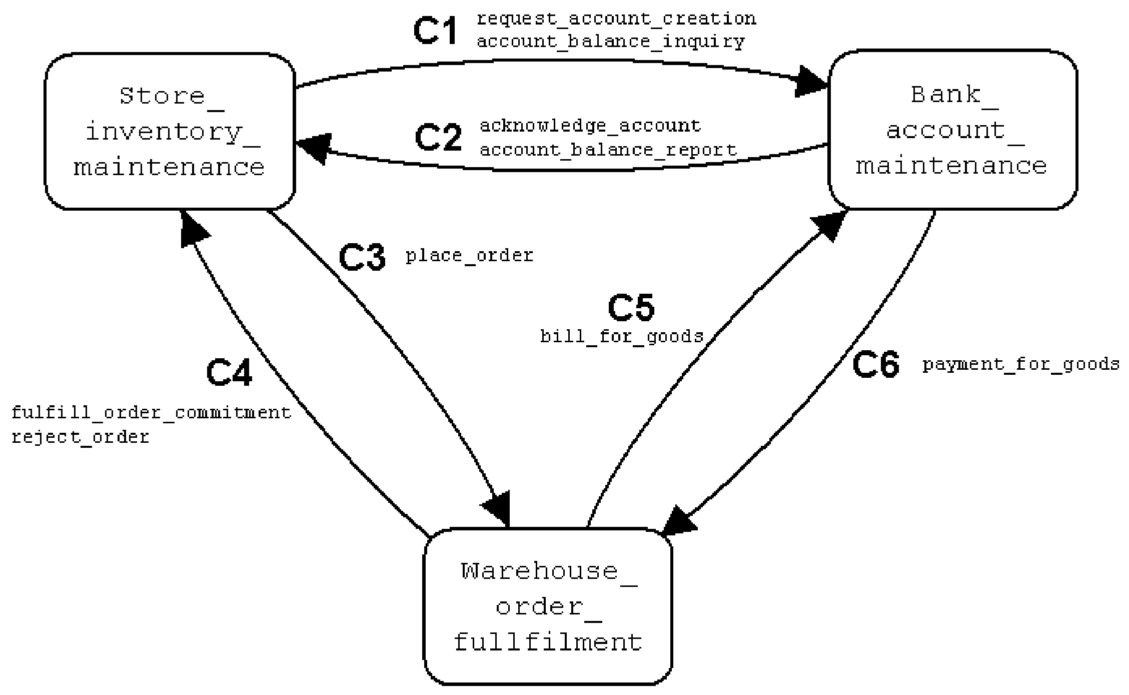

Figure 2.3: Channels for the Store-Warehouse-Bank example

Example 2.2(d): Static and Dynamic Channels. We now consider how channels might be used in the Store-Warehouse-Bank example. Figure 2.3, illustrates how six pre-defined channels might be established between the three services (see Subsection 3.1.5). The figure also shows how the various message types are assigned to the channels. In a domain theory over FLOWS-Core, this would be achieved by asserting constraints such as `channel_source`(*C5*, `Warehouse_order_fulfillment`), `channel_target`(*C5*, `Bank_account_maintenance`), and `channel_mtype` (*C5*, `bill_for_goods`).

Suppose that constraints of this sort are specified in a domain theory, and also that a constraint is specified stating that all messages must be placed on a channel. Then specification of service behavior can be simplified. To illustrate, consider a Produce_Message atomic process that is a subactivity of `Warehouse_order_fulfillment`, and that creates messages of type `bill_for_goods`. For any occurrence of this atomic process, the message created will necessarily be placed on channel *C5*. Furthermore, only occurrences of the service `Bank_account_maintenance` will be able to read this message.

Suppose now that new Warehouses can be added to the overall system at arbitrary times. For example, suppose that a new service `Alternative_warehouse_order_fulfillment` becomes available at some time *t*. It might be natural in this case to create new channels *C7* and *C8*, analogous to *C3* and *C4*, but connecting `Store_inventory_maintenance` to `Alternative_warehouse_order_fulfillment` rather than to `Warehouse_order_fulfillment`. After this, occurrences of `Store_inventory_maintenance` could make orders against either warehouse.

A different approach to using the new warehouse service `Alternative_warehouse_order_fulfillment` would be to modify channel *C3* by adding `Alternative_warehouse_order_fulfillment` as a new target, and to modify channel *C4* by adding `Alternative_warehouse_order_fulfillment` as a new source. In an associated application domain theory, the destination of `place_order` messages might be non-deterministic, or might be determined by properties of the message payload and/or relevant fluents (e.g., the inventory at the two warehouses).

2.3 Illustration of Rich Internal Process Model

This brief example is intended to illustrate the flexibility of the FLOWS-Core ontology. Specifically, we illustrate how the framework can be applied in the context of services coordination for telecommunications, a discipline which has characteristics somewhat different than traditional e-commerce oriented, transaction-based web services. A basic building block in the deployment of telecommunications services is the notion of "session" (e.g., in which several people are in communication, possibly by multiple media), and a basic concern in telecommunications is dealing with asynchronous events (such as wireless connections dropping or pre-pay accounts running out) that might occur at essentially any time. The IETF Session Initiation Protocol (SIP) provides an extensible family of standardized protocols for initiating and maintaining communication sessions; this operates at a fairly low level in the network, tends to focus on a single media at a time, and addresses a variety of details concerning both signaling and real-time packet streams, the selection of codecs, etc. The 3GPP IP Multimedia Subsystem (IMS) reference architecture provides an architectural framework and a broad family of protocols extending core SIP, with a focus on enabling rich, flexible, and scalable communication services. We focus here on the use of the FLOWS-Core ontology for describing session management and orchestration at an even higher level, which can be realized on top of the SIP and IMS layers.

To illustrate, we describe a simple environment which includes 3 core services, namely, `audio_bridge`, `video_server`, `teleconf_manager`; and includes in essence a service corresponding to each kind of end-user device, say, `device_1`, ..., `device_n`. (E.g., perhaps we could have a service `device_i` for each style of cell phone currently available.)

The `audio_bridge` would support messages of the sort "initiate_audio_bridge" (in-going), "audio_bridge_initiated(bridge_id:)" (out-going), "add_end_point(bridge_id, end_point_id)" (in-going), "drop_end_point(bridge_id, end_point_id)" (in-going), and "cancel_audio_bridge(bridge_id)" (in-going). Atomic processes internal to the `audio_bridge` would include things like "add_to_bridge (bridge_id, end_point_id)", which would have the real-world effect of establishing a real-time packet (RTP) stream between the physical bridge server and the physical end-point. To represent this real-world situation a domain-specific fluent `active_audio_sessions(bridge_id, end_point_id)`, can be used, where a pair (*b*, *e*) in this fluent would indicate that end-point *e* is currently connected to bridge *b*. Note that a failure in the network could lead to breaking the RTP stream, and in turn the removal of pairs from the fluent, at essentially any (non-deterministic) time during processing.

The `video_server` would support similar messages for enabling one or more end-points to connect to the server, support atomic processes for establishing RTP streams to end-points, and rely on a fluent `active_video_sessions(session_id, end_point_id)` analogous to `active_audio_sessions`. In addition, for a given video session *s*, there will be the state of the video server, which is whether it is currently streaming a video clip, or whether an end-user has requested fast-forward, fast-reverse, or scene-based look-up request. Thus, the `video_server` can support incoming messages of the sort "play_video(session_id, video_id)", "fast_forward(session_id)", "fast_reverse(session_id)", "indexed_scene_selection(session_id,index)". Additional atomic processes in `video_server` would include capabilities for manipulating the content of those streams. A corresponding domain-specific fluent `video_session_status(session_id, current_video_id, play_back_state, video_location)` is used to track the status of each video session. (PSL is based on discrete time, so this fluent is assumed to work with discrete time as well.)

The `conference_manager` can serve as a single point of access, so that end-points can enter into multi-media multi-party teleconferences, and take advantage of additional features not directly available from the `audio_bridge` and `video_server` services. This might include mute capabilities, the establishment of "whisper" sessions (e.g., in which a subset of the teleconference participants temporarily drop out of the primary audio session and create a separate, private audio session just for themselves), the possibility that only a subset of the participants are watching the video, an awareness of who is speaking at a given moment (so that it can be delivered through a web portal to end-users), etc. The `conference_manager` might also take care of billing considerations, using messages to and from a billing server. In practice, the `conference_manager` will need to maintain the state of all of the conferences it is managing. One natural way to represent this would be using one or more domain-specific fluents, which are accessible only by occurrences of `conference_manager`.

We note that the process model used inside the `conference_manager` might be based on explicit support for sessions and subsessions, rather than being based on flowchart-based constructs as found in [*BPEL 1.1*] or [*OWL-S 1.1*]. In particular, it would be natural for the internal process model to support in essence spawning of an unbounded number of sub-processes (sub-activity occurrences), where each one corresponds to the addition of one more participant into a teleconference. This is reminiscent of the ability of many process algebras to support process spawning, but is not supported in BPEL 1.1. (Such unbounded spawning is supported in BPML [*BPML*].) Furthermore, it is important that the internal model for `conference_manager` be able to accommodate specifications for what should happen if asynchronous events of various types should arise at essentially any time. There is evidence that it is cumbersome to specify the treatment of such asynchronous events in BPEL, because the BPEL constructs related to "waiting" for asynchronous events are generally tied to scopes. In a telecom context, the natural treatments for several kinds of asynchronous events cut across the scoping that is convenient to establish in a BPEL specification. In any case, the needed internals of `conference_manager` can be modeled accurately using FLOWS-Core.

Finally, there are services associated with the end-points. These will include the capability for sending messages to the `conference_manager` to request a teleconference and perhaps to request information about an ongoing teleconference (e.g., who is speaking now?). Atomic processes would include the ability to start receiving or sending an RTP stream, the initiation or cancellation of presentation/display of audio/video output to the end-user, and maintenance of information in connection with the RTP streams.

3 The Amazon E-Commerce Service

The Amazon e-commerce service (ECS)exposes a range of capabilities including the full Amazon product catalogue. A freely downloadable development kit enables developers to create value-added web-sites and services. In addition to exposing product information through comprehensive search capabilities, Amazon offer remote shopping cart resources that can be managed online and submitted for check-out processing.

We use the Amazon scenario to explore the benefits that semantic modeling of web-services can provide. We focus on the requirements for advertising & matchmaking; contracting & negotiation; process modeling & enactment, and express these in terms of a concrete and straightforward challenge for semantic modeling: *to assist the customer in buying a copy of "The Description Logic Handbook" at the cheapest price.*

The terminology of the scenario is drawn from the Web Services Architecture document (plus extensions from [Preist04]).

Advertising and Matchmaking

We don't assume from the outset that the customer will buy the book from Amazon. They may in fact consult many different book vendors and select the cheapest offer. We are interested in how the customer finds the Amazon service, and how detailed the service description needs to be. Let us be clear, we are not at this stage trying to describe the operational web-service interface for which one may continue to use WSDL. Rather we aim to describe, and advertise, the service in terms of its value proposition.

Advertising is therefore concerned with the description of this *service offer* and its flip-side the *service requirement*. It is unreasonable to expect Amazon to advertise their service in terms of each and every product available; they may instead publish a generalized service offer. For example, Amazon divide their product range into a number of offerings including Books, Music, DVDs, etc. The service offer may simply describe Amazon as a book vendor, assuming we have a readily available domain ontology that declares the concept of 'Books' together with a suitable 'offering' relationship. Another resource we have available to describe the type of service offered by Amazon is the MIT Process handbook that classifies the customer interaction with Amazon.com as an example of the 'Sell via electronic store with posted prices' business process.

The service offer is published in some form so that the Amazon service can be discovered either directly by the customer, or by using some third-party discovery service. The service requirements mirror the service offer, though we will assume the request is more general in the way it classifies the service but is more specific in the product detail. The requirements will define only a service that will 'Sell' a Book with the name 'The Description Logic Handbook'. To 'Sell' is more generic than to 'Sell via electronic store...', however the class of Books with this specific product name is more specific than Books in general. Therefore, even in this simple example there is no strict taxonomical relationship between the service offer and requirements.

The function of matchmaking is to identify the relationship (if any) between the service offer and the service request. We are primarily interested in matches in the intersection of the service offer and request. An additional conundrum appears when we consider the service offer, described above, in relation to other service offers. For example, consider an alternative offer that describes Amazon as a Music vendor. If Music is not explicitly disjoint from Books then there are also solutions in the intersection where Music is deemed synonymous with Books. What is the logical justification for ranking the Books offer over the Music offer? SWSL rules may be used to define this matchmaking relationship; the relationship between matching service offers and requirements. These rules may be interpreted by an agent to perform service discovery. Furthermore, the non-monotonic features of SWSL rules enable us to select the *best* matches available - selecting Amazon Books over Amazon Music.

Contracting and Negotiation

The abstract nature of the service offer means that we can't immediately see if the required book is actually for sale and at what price. The requester must enter into direct negotiation with the service provider agent to fill in the gaps - to come up with the *service contract*. The contract represents an agreement between the customer and the service provider, defining exactly the service (instance) to be provided. The contract must be specific about the required product, for the title may not be sufficient: for books the Amazon Standard Item Number (ASIN) provides a unique product code equivalent to the book's ISBN.

As with service descriptions and matchmaking, the service contract defines the content of a negotiation process. With Amazon ECS, negotiation involves a catalogue search that returns product info including price and availability. Only at this point do we have the raw material for a service contract - a tangible offer. To flesh-out and and shake-hands on the contract, the agent adds the book to the cart and proceeds to the checkout. We assume the final go-ahead to make the purchase is referred back to the customer. This style of negotiation follows a very simple *take-it-or-leave-it* pattern, but requires a technically non-trivial choreography that differs between services. For this reason, we require a method of process modeling so that our agent can negotiate on the customer's behalf.

Process Modeling and Enactment

A key feature of the Service Oriented Architecture is its characterization of the tasks and actions concomitant with the negotiation and delivery of the service. Tasks are modeled as processes, which break down by way of *composite* processes into more primitive *atomic* processes that represent these basic actions. Additionally, every process is modeled in terms of its Inputs, Outputs, Preconditions and Effects (IOPEs) which provide an epistemic account of the impact of these actions on the knowledge-base of the agent. Through process modeling we aim to describe the overall negotiation task which involves actions that relate to catalogue search; remote cart management; and checkout processing (there are also tasks that pertain to service delivery, such as order tracking, that are not covered in the immediate scenario). The SWSL process model expresses declarative processes constraints rather than directly executable programs, providing additional flexibility to agents that may use them to govern, rather than determine, their behaviour. The process model is further underpinned by a theory of the semantics of these tasks which captures the meaning and purpose, over and above the mechanics, of the interaction.

For brevity we only model a subset of the Amazon ECS capabilities required to describe the book-buying scenario. One of the basic operations available to the customer agent is to search the product catalogue for the desired item. One of the actions available to the requester agent is to request an ItemSearch. This *atomic process* is defined below. The search index defines the Amazon store to be searched which for the scenario will be 'Books'. The 'title' of the book is a string as defined above which represents all or part of the required product name. The 'items' parameter represents the search output that will include potentially more than one matching item with pricing and availability details. The example is described using the presentation syntax which is a convenience notation that translates into SWSL-FOL axioms.

```
aws#ItemSearch(?searchIndex,?title,?items) {
        Atomic
        input aws#SubscriptionId xs#string
        input aws#Operation xs#string 'ItemSearch'
        input aws#SearchIndex xs#string ?searchIndex
        input aws#Title xs#string ?title
        input aws#ResponseGroup xs#string 'Offers'
        output aws#Items aws#ItemsOffered ?items
}
```

These examples are supported by the creation of a domain ontology that defines the structured objects that appear in the scenario, specifically definitions of 'ItemsOffered', using SWSL. 'ItemsOffered' is simply a collection of 'ItemOffered' (a 'Item') each of which may have a number of actual offers. The only property that is necessary for an item is the ASIN. This is subclassed in ItemSearch to include offers detailing pricing and availability information. It is also subclassed in CartCreate to include the required quantity for each item.

```
prefix aws = _"http://webservices.amazon.com#".
prefix xs = _"http://www.w3.org/2001/XMLSchema#".

aws#Items [ aws#item => aws#Item ].
aws#Item [ aws#ASIN => xs#string ].

aws#ItemsOffered [ aws#item => aws#ItemOffered ] :: aws#Items.
aws#ItemOffered [ aws#offers => aws#Offers ] :: aws#Item.

aws#Offers [ aws#offer => aws#Offer ].
aws#Offer [
        aws#price => aws#Price,
        aws#availability => xs#string
].

aws#Price [
        aws#amount => xs#positiveInteger,
        aws#currencyCode => xs#string
].
```

The advantage of using a domain ontology is that we can work with the data indepedently of any particular message syntax. For example, it may allow us to compare the catalogue content with that of other online stores. We also have a level of abstraction that is resilient to minor version changes in the interface specification (unless there is a semantic difference). The service grounding should identify the appropriate domain ontology and provide enough information for a client to lift message content into this abstract ontology. Amazon web-services are interesting in this respect because they provide two alternative modes of service invocation with equivalent semantics. A Service Oriented Model utilizing SOAP over HTTP is supported, together with a REST-compliant Resource Oriented Model.

In the last example, nothing really changes other than the agent's *knowledge* of the product catalogue. Contrast this with the effect of adding the book to the cart. Amazon provides a 'remote shopping cart' resource that is managed by the Amazon server. Operations on the cart allow an agent to add items to a new or existing cart; to clear and modify a cart; and to view the current contents of a cart. The real-world effect is the state-change of a remote shopping cart (a new cart resource is created). Any stateful resource is described by a *fluent* that reflects the way it changes in response to events. *Effects* encode the changes that can be directly attributed to an action. *Preconditions* describe the state of the world (i.e. not just the knowledge of our agent) that must pertain prior to performing an action.

We look first at adding an item to a new shopping cart with CartCreate. We can add more than one item at a time so the input is again a collection of items. The response includes a URL from which the contents of the cart may be purchased. The existence of this new shopping cart is a statement of fact about the world so this is expressed as an effect; an assertion of the existence of the new cart with a specific cart identifier.

```
aws#CartCreate(?items,?url) {
        Atomic
        input aws#SubscriptionId xs#string
        input aws#Operation xs#string 'CartCreate'
        input aws#Items aws#ItemsRequested ?items
        output aws#CartId xs#string ?cart
        output aws#purchaseURL xs#anyURI ?url
        effect aws#cart(?cart)
}
```

This example require additional domain classes that define 'ItemsRequested'. This is again a subclass of 'Items' with the addition in 'ItemRequested' of the required quantity required.

```
aws#ItemsRequested [ aws#item => aws#ItemRequested ] :: aws#Items.
aws#ItemRequested [ aws#quantity => xs#positiveInteger ] :: aws#Item.
```

The 'items' are an *output* of item search and a required *input* to any cart operation. We can describe this linkage of inputs to outputs as a constraint on the relative ordering of an occurrence of a search and cart operation. In this case we wish to describe a *sequential* composition where we put the search before the cart. We describe this composite process as a shopping activity, and just like window shopping there is no implication that anything is actually purchased.

```
aws#shopping(?searchIndex,?title,?qty,?cart,?url) {
        Sequence
        occurrence ?o1 aws#ItemSearch(?searchIndex,?title,?items)
        occurrence ?o2 aws#CartCreate(?items,?cart,?url)
        ?o1 soo_precedes ?o2
}
```

Currently, the Amazon API does not allow an agent to complete a purchase automatically, but each shopping cart has a purchase URL that the customer can visit to finalize the purchase. This creates an interesting challenge for process modeling. How do we represent these out-of-band actions that are not directly performed by the agent and are not grounded in web-services? The effect of order submission is also significantly different from the simple state-changes we see in shopping carts. Adding items to a cart does not *commit* the buyer to anything. However, when the buyer places a confirmed order, the seller is then under an obligation to deliver the product (and the customer is under obligation to pay). This is understood to be central to the buyer/seller contract and may be described as an obligation between the requester and provider. It represents a commitment to honour an agreement at all future times (until it is discharged). In reality, these obligations are conditional. The buyer has the right to cancel the order at any time up until the order is actually dispatched. Conversely, a permission proffers the right at some future time to perform an action (until it is withdrawn). Both obligations and permissions are described in the web-service policy model.

```
aws#purchase(?cart,?url) {
        Atomic
        input aws#purchaseURL xs#anyURI ?url
        effect aws#commit(?cart)
}
```

4 Service Discovery with SWSL-Rules

This example illustrates the use of SWSL-Rules for Web service discovery. The particular features of the language that this example relies on include frame-based representation, reification, and nonmonotonic Lloyd-Topor extensions. In addition, logical updates à la Transaction logic [*Bonner98*] are used in the discovery queries. Transaction Logic was mentioned in Section 3.15 as a possible extension for SWSL-Rules.

To make the example manageable, services are described only by their names and conditional effects. To discover a service, users must represent their goals using the goal ontology described below. These goals are described in terms of *user requests*, which represent formulas that the user wants to be true in the after-state of the service (i.e., the state that would result after the execution of the service).

User goals and services may be expressed in different ontologies and so *mediators* are needed to translate between those ontologies. This type of mediators is known as *wgMediators* [*Bruijn05*]. In this example, we assume that each service advertises the mediators that can be used to talk to this service though the attribute `mediators`.

Ontologies

Geographical ontology. To begin, we assume the following simple geographic taxonomy, which is shared by user goals and services. It defines several regions and subregions, such as `America`, `USA`, `Europe`, `Tyrol`. Each region is viewed as a class of cities. For instance, `Innsbruck` is a city in `Tyrol` and `'Stony Brook'` is a town in the New York State (`NYState`).

```
USA::America.
Germany::Europe.
Austria::Europe.
France::Europe.
Tyrol::Austria.
NewYorkState::USA.
StonyBrook:NewYorkState.
NewYork:NewYorkState.
Innsbruck:Tyrol.
Lienz:Tyrol.
Vienna:Austria.
Bonn:Germany.
Frankfurt:Germany.
Paris:France.
Nancy:France.

Europe:Region.
America:Region.

?Reg:Region :- ?Reg1:Region and ?Reg::?Reg1.
?Loc:Location :- ?Reg:Region and ?Loc:?Reg.
```

To make it easier to specify what is a region and what is not, we use a rule (the penultimate statement above) to say that a subclasses of a region are also regions and, therefore, such subclasses do not need to be explicitly declared as regions. The last rule simply says that any object that is a member of a geographical region is a location.

Goal ontology. Services write their descriptions to conform to specific ontologies. Likewise, clients describe their goals in terms of goal ontologies. Here we will not describe these ontologies, but rather the forms of the inputs and outputs that the services expect to produce and the structure of the user goals. Furthermore, since users and service designers are unlikely to be skilled knowledge engineers, we assume that the inputs, the outputs, and the goals are fairly simple and that most of the intelligence lies in the mediators.

We assume that there is one ontology for goals and two for services. Consequently, there are two mediators: one translating between the goal ontology and the first service ontology, and the other between the goal ontology and the second service ontology.

The goal ontology looks as follows:

```
Goal[requestId *=> Request,
     request   *=> TravelSearchQuery,
     result    *=> Service
].
```

The classes `Request` and `Service` will be specified explicitly by placing specific object Ids in them. The class `TravelSearchQuery` consists of the following search queries:

```
searchTrip(?From,?To):TravelSearchQuery :-
              ?From:(Region or Location) and ?To:(Region or Location).
searchCitipass(?Loc):TravelSearchQuery :- ?Loc:(Region or Location).
```

The meaning of the query `searchTrip(?X,?Y)` depends on whether the parameters are regions or just locations. For location-parameters, the query is assumed to fetch the services that serve those locations. For region-parameters, the query is assumed to find services that service *every* location in the region that is known to the knowledge base. For instance, `searchTrip(Paris, Germany)` is a request for travel services that can sell a ticket from Paris to *any* city in Germany. Similarly, `searchCitipass(NewYork)` is interpreted as a search for travel services that can sell city passes for New York and the request `searchCitipass(USA)` is looking for services that can sell city passes for every location in USA. The `result` attribute is provided by the ontology as a place where the discovery mechanism is supposed to put the results.

Domain-specific service ontologies. A service ontology is intended to represent the inputs and outputs of the service as well as the effects of the service. Since the inputs are not generally provided in the user goal (since the user is not expected to know anything about such inputs), the job of translating goal queries into the inputs to the services lies with the mediator.

Service ontology #1 is defined as follows:

```
// Service input
search(?requestId,?fromLocation,?toLocation):ProcessInput :-
                 ?requestId:Request and
                 ?fromLocation:Location and ?toLocation:Location.
search(?requestId,?city):ProcessInput  :-
                 ?requestId:Request and ?city:Location.
// Service output
ItineraryInfo::ServiceOutput.
PassInfo::ServiceOutput.
ItineraryInfo[from*=>Location, to*=>Location].
PassInfo[city*=>Location].
itinerary(?reqNumber):ItineraryInfo :- ?reqNumber:Request.
pass(?reqNumber):PassInfo :- ?reqNumber:Request.
```

Note that services expect locations as part of their input and they know nothing about regions. In contrast, as we have seen, user goals can have region-wide requests. It is one of the responsibilities of the mediators to bridge this mismatch.

Service ontology #2 is similar to ontology #1 except that it understands only requests for citipasses and the formats for the input and the output are slightly different.

```
// Service input
discover(?requestId,?city):ProcessInput  :-
                 ?requestId:Request and ?city:Location.
// Service output
ServiceOutput[location*=>Location].
?reqNumber:ServiceOutput :- ?reqNumber:Request.
```

Shared core ontology for services. In addition, we need a core ontology that is shared by everyone in order to provide a common ground for the service infrastructure. In this example, the core ontology is represented by a single class `Service`, which is declared as follows:

```
prefix xsd = "http://www.w3.org/2001/XMLSchema".
Service[
   name        *=> xsd#string,
   process     *=> Process[effect(ProcessInput) *=> Formula],
   mediators   *=> Mediator
].
```

Note that the definition of the class `Service` belongs to the core ontology and therefore it is shared by everybody. The method `effect` represents the conditional effect of the service. It takes an input to the service as a parameter and returns a set of rules that specify the effects of the service for that input. `Formula` is a predefined class. The attribute `mediators` indicates the mediators that the service advertises for anywho would want to talk to that service.

Note that the class `ProcessInput` belongs to the core ontology, but it's extension (the set of objects that are members of that class) is defined by domain-specific ontologies.

Examples of Concrete Services

We now present instances of concrete services.

```
// This service uses ontology #1, and mediator med1 bridges it to the goal ontology
serv1:Service[
   name -> "Schwartz Travel, Inc.",

   // Input must be a request for ticket from somewhere in Germany to somewhere
   // in Austria  OR  a request for a city pass for a city in Tyrol
   // Depending on the input, output is either an itinerary object with Id
   // itinerary(requestId) or a citipass object with Id
   // pass(requestId).
   process ->
       _#[effect(?Input) -> ${
               (itinerary(?Req)[from->?From,to->?To] :-
                           Input = search(?Req, ?From:Germany, ?To:Austria))
```

```
                  and
                  (pass(?Req)[city->?City] :- ?Input=search(?Req,?City:Tyrol))
                                 }],
        mediators -> med1
    ].

    // Another ontology #1 service
    serv2:Service[
        name -> "Mueller Travel, Inc.",
        process ->
            _#[effect(?Input)-> ${
                    itinerary(?Req)[from->?From, to->?To] :-
                         ?Input = search(?Req,?From:(France or Germany),?To:Austria)
                                  }],
        mediators -> med1
    ].

    // An ontology #2 service
    serv3:Service[
        name -> "France Citeseeing, Inc.",
        process ->
            _#[effect(?Input)-> ${
                     ?Req[location->?City] :- ?Input=discover(?Req,?City:France)
                                   }],
        mediators -> med2
    ].

    // Another ontology #2 service
    serv4:Service[
        name -> "Province Travel",
        process ->
            _#[effect(?Input)-> ${
                  ?Req[location->?City] :-
                           ?Input = discover(?Req,?City:France) and ?City != Paris
                                  }],
        mediators -> med2
    ].
```

User Goals

Next we show examples of user goals. Note that the value of the attribute `result` is initially the empty set. When the goal is posed to the discovery engine, this value will be changed to contain the result of the discovery.

```
    goal1:Goal[
       requestId -> _#:Request,
       request   -> searchTrip(Bonn,Innsbruck),
       result    -> {}
    ].

    // search for services that serve all cities in France and Austria
    goal2:Goal[
       requestId -> _#:Request,
       request   -> searchTrip(France,Austria),
       result    -> {}
    ].

    goal3:Goal[
       requestId -> _#:Request,
       request   -> searchCitipass(Frankfurt),
       result    -> {}
    ].

    goal4:Goal[
       requestId -> _#:Request,
       request   -> searchCitipass(Innsbruck),
       result    -> {}
    ].

    // services that can sell citipasses for every city in France
    goal5:Goal[
       requestId -> _#:Request,
       request   -> searchCitipass(France),
       result    -> {}
    ].
```

Mediators

Each of the two mediators, `med1` and `med2`, consists of several main clauses. The first clause in each mediator takes a user goal and translates it into input (to services) that is appropriate for the corresponding domain-specific service ontology.

The remaining clauses define the mediator's method `getResult`. This method is supposed to be invoked in the after-state of the service execution. It takes as parameters the user goal and the service (in whose after-state the method is invoked). Depending on the form of the goal's request, `getResult` poses a query that is appropriate for that request and the service ontology of the service. For instance, if the request is `searchCitipass(?City:Location)`, i.e., finding services that can sell citipasses for a specific location, then the query appropriate for services that use ontology #1 is `pass(?)[city->?City]` and the query for ontology #2 is `?[location->?City]`. Finally, if the query yields results, the mediator constructs output that can be used to return results to the user and this output is compliant with our goal ontology.

Each form of the input has two cases: one assumes that the parameters are locations (e.g., `searchCitipass(?City:Location)`) and the other that they are regions (e.g., `searchCitipass(?City:Region)`). Therefore, for each form of the input our mediators have two clauses. Since ontology #2 understands only one input, `med2` uses only two clauses to define `getResult`. The mediator for ontology #1, `med1`, needs four clauses to cover both forms of the input.

Finally, we remark that the clauses that deal with region-based requests have to construct more sophisticated queries to be asked in the after-state of the services. In our example, we use nonmonotonic Lloyd-Topor extensions to simplify such queries.

```
// mediator for ontology #1
med1:Mediator.
med1[constructInput(?Goal)->?Input] :-
        ?Goal[requestId->?ReqId, request->?Query] and
        if ?Query = searchTrip(?From,?To)
        then ?Input = search(?ReqId,?From1,?To1)
        else if ?Query = searchCitipass(?City)
        then ?Input = search(?ReqId,?City1).

med1[getResult(?Goal,?Serv) -> ${?Goal[result->?Serv]}] :-
        ?Goal[request->searchCitipass(?City:Location)] and
        pass(?)[city->?City].
med1[getResult(?Goal,?Serv) -> ${?Goal[result->?Serv]}] :-
        ?Goal[request->searchCitipass(?Region:Region)] and
        forall ?City (?City:?Region ==> pass(?)[city->?City]).

med1[getResult(?Goal,?Serv) -> ${?Goal[result->?Serv]}] :-
        ?Goal[request->searchTrip(?From:Location,?To:Location)] and
        itinerary(?)[from->?From, to->?To].
med1[getResult(?Goal,?Serv) -> and ?Result = ${?Goal[result->?Serv]}] :-
        ?Goal[request->searchTrip(?From:Region,?To:Region)] and
        forall ?From,?To (?City1:?FromReg and ?City2:?ToReg
                          ==> itinerary(?)[from->?City1, to->?City2]).

// mediator for ontology #2
med2:Mediator.
med2[constructInput(?Goal)->?Input] :-
        ?Goal[requestId->?ReqId, request->?Query] and
        if ?Query = searchCitipass(?City)
        then ?Input = discover(?ReqId,?City1).

med2[getResult(?Goal,?Serv) -> ${?Goal[result->?Serv]}] :-
        ?Goal[request->searchCitipass(?City:Location)] and
        ?[location->?City].
med2[getResult(?Goal,?Serv) -> ${?Goal[result->?Serv]}] :-
        ?Goal[request->searchCitipass(?Region:Region)] and
        forall ?City (?City:?Region ==> ?[location->?City]).
```

The Discovery Engine

The final piece of the puzzle is the actual engine that performs service discovery. It relies on the features, borrowed from Transaction Logic [*Bonner98*], which are currently not in SWSL-Language, but are considered for future extensions. These features include modifications to the current state of the knowledge base and hypothetical execution of such modifications.

```
findService(?Goal) :-
        ?Serv[mediators -> ?Mediator] and
        ?Mediator[constructInput(?Goal) -> ?Input] and
        ?Serv.process[effect(?Input) -> ?Effects] and
        hypothetically(
            insert{?Effects} and
            ?Mediator[getResult(?Goal,?Serv) -> ?Result]
        ) and
        insert{?Result}.
```

The `findService` transaction performs the following tasks:

1. For each service instance, *s*, it finds the mediator that the service advertises.
2. It then uses the mediator to construct the input for that service based on the user goal.
3. Using the input, it computes the effects that the service guarantees to be true in the after-state of the execution.
4. It then hypothetically does the following:
    1. It inserts the effects (which are rules in this case) into the knowledge base. This simulates the execution of the service *s* and temporarily creates the after-state of the service execution.
    2. Using the mediator, it checks whether the user goal is true in the after-state of the service *s* and then returns the result.
5. If the above hypothetical execution fails for a particular service, no result is returned and the subsequent `insert` operation is not executed. If the hypothetical execution succeeds, it means that the service *s* matches the goal. After the hypothetical execution, the state of the knowledge base returns to what was before the execution of the service, but the variable `?Result` is now bound to a result, which is a formula of the form

    goal`[result->`*s*`]`

    This is then inserted into the knowledge base. In this way, the set of answers to the goal is built as the value of the `result` attribute of the goal object.

For instance, if

```
?- findService(goal1).
```

is executed then the following will become true:

```
goal1[result -> {serv1,serv2}]
```

Similarly, executing

```
?- findService(goal2).
```

yields `goal2[result -> serv2]`. The third goal, `goal3`, matches none of the services listed above, so only `goal3[result->{}]` can be derived. Executing

```
?- findService(goal4).
```

yields `goal4[result -> serv1]`.

A more interesting goal is `goal5`, because it requests citipasses for an entire region (France). Given the information available in our knowledge base, only `serv3` should match. Note that `serv4` does not match because it does not serve Paris, while the goal specifies only those services that can sell citipasses for *every* location in France.

5 Policy Rules for E-Commerce

The overall SWSL language and ontologies support many kinds of application scenarios. In this subsection, we discuss in detail how SWSL-Rules can be used to represent several (fictional) examples of policies for e-commerce. Some of these examples are given directly in the remainder of this section, in full detail. These include:

price discounting example; refunds example; supply chain ordering lead time example; creditworthiness example; and credit card transaction authorization example.

Taken together these examples include both B2C/retailing and B2B/supply-chain aspects, in several industry domains (books, appliances, and computer equipment manufacturing). Each of these policies is useful for not just one but several different kinds of tasks within an application realm. For example, price discounting rules are useful in advertising and in service contract specification. Most of these detailed examples are rather brief, for the sake of expository simplicity. However, the example dealing with authorization of credit card transactions is significantly longer and more realistic.

Additional examples of policy rules are available that use the same fundamental knowledge representation as SWSL-Rules. [*Grosof2004e*] gives a long example of e-contracting policy rules that combines rules with ontologies and deals with exception handling / monitoring. The [*Grosof2004f*] tutorial gives a collection of use cases and examples, including those for semantic mediation. The SweetRules [*Grosof2004f*] downloadable includes a collection of examples.

Overall, the SWSL-Rules is especially well suited to represent available knowledge and desired patterns of reasoning for several of the SWS tasks, including:

- authorization policies (for security, access control, confidentiality, privacy, and other kinds of trust);
- contracts (partial or complete, proposed or finalized), e.g., terms and conditions, service provisions, or surrounding business process policies;
- monitoring of processes to recognize and handle exceptions or other dynamic conditions (e.g., monitoring of performance of contracts to detect and respond to violations of contract provisions such as late delivery or non-payment);
- advertising, discovery, and matchmaking (e.g., advertisements and requests for quotation or requests for proposals can be regarded as partial contract proposals);
- semantic mediation, especially translation mappings that mediate between different ontologies or contexts and thus between knowledge expressed in those different ontologies or contexts (e.g., to translate from the output of one service to the input expected by another service); and
- object-oriented ontologies that use default inheritance with priorities and/or cancellation (e.g., in the manner of the Process Handbook [*Process Handbook*, *Bernstein2003*, *Grosof2004d*]).

In particular, the capabilities of SWSL-Rules for logical nonmonotonicity (negation-as-failure and/or Courteous prioritized conflict handling) is used heavily in many use case scenarios for each of the above tasks and its associated kinds of knowledge.

Personalized Price Discounting in an E-Bookstore

Next, we give an example of a set of policy rules that specify personalized price discounting in an e-bookstore. These rules are useful in advertising, and also as part of a contract (proposed or final).

The policy rules specify that by default, a shopper gets no discount (i.e., gets a zero percent discount). However, there are some particular circumstances which do warrant a discount. Loyal purchasers get a five percent discount. Members of the Platinum Club get a ten percent discount. However, customers who have been late in making payments during the last year get no discount. Also there is a mutex (integrity constraint) rule which specifies that it is a contradiction to conclude two different values of the discount percentage for the same customer, i.e., that the discount percentage should be unique.

```
/* price discounting policy rules */ {ordinary} giveDiscount(percent00,?Cust) :- shopper(?Cust). {loyal} giveDiscount(percent05,?Cust) :- shopper
(?Cust) and loyalPurchaser(?Cust). {platinum} giveDiscount(percent10,?Cust) :- shopper(?Cust) and member(?Cust,platinumClub). {slowPayer}
giveDiscount(percent00,?Cust) :- slowToPay(?Cust,last1year). overrides(loyal,ordinary). overrides(platinum,loyal). overrides(platinum,ordinary).
overrides(slowPayer,loyal). overrides(slowPayer,platinum). !- giveDiscount(?X,?Cust) and giveDiscount(?Y,?Cust) | notEquals(?X,?Y). /* some
"case" facts about particular customers ann, cal, kim, and peg. */ shopper(ann). shopper(cal). loyalPurchaser(cal). shopper(kim). member(kim,
platinumClub). shopper(peg). loyalPurchaser(peg). slowToPay(peg,last1year).
```

The above premises (policy rules and case facts) together entail the following conclusions about the discount percentages for the particular customers Ann, Cal, Kim, and Peg.

```
/* the entailed discount percentages for the particular customers */ giveDiscount(percent00,ann). giveDiscount(percent05,cal). giveDiscount
(percent10,kim). giveDiscount(percent00,peg).
```

Refunds in an E-Retailer

Next, we give a typical example of a seller's refund policy, as a set of policy rules that specify refunds in an e-retailer (here, of small consumer appliances) These rules are useful in advertising, and as part of a contract (proposed or final), and as part of contract monitoring / exception handling (which is in turn part of contract execution).

The unconditional guarantee rule says that if the buyer returns the purchased good for any reason, within 30 days, then the purchase amount, minus a 10 percent restocking fee, will be refunded. The defective guarantee rule says that if the buyer returns the purchased good because it is defective, within 1 year, then the full purchase amount will be refunded. A priority rule says that if both of the previous two rules apply, then the defective guarantee rule "wins", i.e., has higher priority. A mutex says that the refund percentage is unique per customer return.

```
/* refund policy rules */ {unconditionalGuarantee} refund(?Return,percent90) :- return(?Return) and delay(?Return,?D) and lessThanOrEqual(?D,
days30). {defectiveGuarantee} refund(?Return,percent100) :-         return(?Return) and reason(?Return,defective) and delay(?Return,?D) and
lessThanOrEqual(?D,years1). overrides(defectiveGuarantee,unconditionalGuarantee). !- refund(?Refund,percent90) and refund(?Refund,percent100). /*
some background facts (typically provided by lessThanOrEqual being a built-in predicate */ lessThanOrEqual(days12,days30). lessThanOrEqual(days44,
years1). lessThanOrEqual(days22,years1). lessThanOrEqual(days22,days30). /* some "case" facts about particular customer returns of items */ return
(toaster02). delay(toaster02,days12). return(blender08). delay(blender08,days44). reason(blender08,defective). return(radio04). delay(radio04,
days22). reason(radio04,defective).
```

The above premises (policy rules, background facts, and case facts) together entail the following conclusions about the discount refund percentages for the particular customer returns of the toaster, blender, and radio.

```
/* the entailed refund percentages for the particular customer returns */ refund(toaster02,percent90). refund(blender08,percent100). refund
(radio04,percent100).
```

Ordering Lead Time in Supply Chain (B2B)

In B2B commerce, e.g., in supply chains (especially in manufacturing), sellers often specify how much lead time, i.e., minimum advance notice, is required when a buyer places or modifies a purchase order. Next, we give an example of a part supplier vendor's (here, Samsung supplying computer equipment) lead time policies as a set of rules. These rules are useful in advertising, and as part of a contract (proposed or final), and as part of contract monitoring / exception handling (which is in turn part of contract execution).

The first policy rule says "14 days lead time if the buyer is a preferred customer". This might be authored by the marketing part of the seller's organization. The second policy rule says "30 days lead time if the ordered item is a minor part". This might be authored by the financial accounting part of the seller's organization. The third policy rule says "2 days lead time if: the ordered item is backlogged at the vendor (i.e., the seller is having trouble fulfilling its overall set of existing orders), and the order is a modification to reduce the quantity of the item, and the buyer is a preferred customer". This might be authored by the operations part of the seller's organization. The third rule is given higher priority than the first rule, say, because operations' authority (about lead time) is greater than that of marketing. A mutex says that the lead time is unique per purchase order.

```
/* ordering lead time policy rules */ {leadTimeRule1} orderModificationNotice(?Order,days14) :-        preferredCustomerOf(?Buyer,?Seller) and
purchaseOrder(?Order,?Buyer,?Seller). {leadTimeRule2} orderModificationNotice(?Order,days30) :-        minorPart(?Order) and purchaseOrder(?
Order,?Buyer,?Seller). {leadTimeRule3} orderModificationNotice(?Order,days2) :-        preferredCustomerOf(?Buyer,?Seller) and
orderModificationType(?Order,reduce) and        orderItemIsInBacklog(?Order) and purchaseOrder(?Order,?Buyer,?Seller). overrides(leadTimeRule3,
leadTimeRule1). !- orderModificationNotice(?Order,?X) and orderModificationNotice(?Order,?Y) | notEquals(?X,?Y). /* some "case" facts about
particular purchase orders */ purchaseOrder(po1234567,compUSA,samsung). preferredCustomerOf(compUSA,samsung). purchaseOrder(po5678901,microCenter,
samsung). preferredCustomerOf(microCenter,samsung). orderModificationType(po5678901,reduce). orderItemIsInBacklog(po5678901).
```

The above premises (policy rules and case facts) together entail the following conclusions about the ordering lead time for the particular purchase orders po1234567 and po5678901.

```
/* the entailed lead times for the particular purchase orders */ orderModificationNotice(po1234567,days14). orderModificationNotice(po5678901,
days2).
```

Creditworthiness using Credit Report and Fraud Alert Services

Policies about authorization, including creditworthiness and other kinds of trustworthiness to access information or perform transactions, are often naturally expressed in terms of necessary and sufficient conditions. Such conditions include: credentials, e.g., credit ratings or professional certifications; third-party recommendations; properties of a transaction in question, e.g., its size or mode of payment; and historical experience between the agents, e.g., familiarity or satisfaction.

Next, we give a simple example of policy rules of a merchant about creditworthiness (of customers) using a credit report service and a fraud alert service. These rules are useful for representing authorization policies, as part of contract negotiation, and in contract monitoring and exception handling (e.g., if the fraud alert arrives after contractual agreement has been reached).

The first policy rule says that, by default, the merchant deems a requester customer to be creditworthy if the requester has a good rating from a particular credit report service (CreditReportsRUs, a source trusted by the merchant). The second policy rule says that the merchant deems a requester to be not creditworthy if the requester has a bad rating from any fraud alert service that is recommended as expert by a particular security consultant (recommenderServiceD, again, a source trusted by the merchant). The second rule is given higher priority than the first rule. The merchant is called "self" (as is conventional in the security policy literature for the granting authority institution).

```
/* creditworthiness policy rules */ {credSelf} honest(self,?Requester) :- creditRating(creditReportsRUs,?Requester,good). {frauSelf} neg honest
(self,?Requester) :-        creditRating(?BlackballService,?Requester,bad) and        fraudExpert(recommenderServiceD,?BlackballService).
overrides(frauSelf,credSelf). fraudExpert(recommenderServiceD,studentLoanAgency). /* some "case" facts about particular requester customers */
creditRating(creditReportsRUs,sue,good). creditRating(creditReportsRUs,joe,good). creditRating(studentLoanAgency,joe,bad).
```

The above premises (policy rules and case facts) together entail the following conclusions about the creditworthiness of the particular requester customers Sue and Joe.

```
/* the entailed creditworthiness of the particular requester customers */ honest(self,sue). neg honest(self,joe).
```

Credit Card Transaction Authorization by Merchant and Bank

Next, we give a longer and more realistic example of authorization in e-commerce: authorization by a merchant of credit card transactions requested by customers. In this example, the merchant ("eSellWow") merges the authorization policies of credit card issuer (called the "bank") with the merchant's own additional authorization policies. These rules are useful for specifying authorization policies, contracts, and exception handling.

```
/* Some terminological abbreviations: CVC: Card Verification Code (the three or so numbers found on the back of a credit card) "Bank": the credit
card company that is the issuer of the credit card (e.g., Mastercard) */ /* Predicates' meaning: transactionRequest: a credit card transaction
requested by a customer merchant: a merchant who is established to do credit card transactions with the credit card company ccInGoodStanding:
credit card is in good standing with the credit card company that is the issuer of the credit card ccInfo: credit card info about the account and
its status, that is on file with the bank authorize: the credit card transaction is authorized transactionExpirationDateOf: customer-supplied
expiration date that is part of the transaction transactionCardholderNameOf: customer-supplied cardholder name that is part of the transaction
transactionCVCOf: customer-supplied CVC that is part of the transaction transactionCardholderAddressOf: customer-supplied cardholder billing
address that is part of the transaction ccFraudRating: rating of a credit card by a fraud alerting/tracking service fraudExpert: a service is a
legitimate expert in fraud alerting/tracking fraudRecommenderFor: trusted recommender of fraud experts customerRating: the rating of customer
based on the merchant's own/other experience Built-in predicates (used): notEquals lessThan */ /* The following group of rules are policies of
the bank, then adopted/imported as a group/module by the merchant, in this case eSellWow. */ /* bankGoodStanding: Bank says by default the
transaction is authorized if the card is in good standing */ /* expiredCard: Bank says the transaction is disallowed if the card is expired. */ /
* overLimit: Bank says the transaction is disallowed if the card is above its account limit. */ /* mismatchExpirationDate: Bank says the
transaction is disallowed if the expiration date from the customer or card in the transaction does not match what's on file for the card.
However, the expiration date may not be available as part of the transaction. */ /* mismatchCVC: Bank says the transaction is disallowed if the
Card Verification Code does not match what's on file for the card. However, note that the CVC may not be available as part of the transaction.
*/ /* mismatchAddress: Bank says the transaction is disallowed if customer-supplied cardholder billing address does not match what's on file for
the card. However, the customer-supplied cardholder billing address may not be available. */ /* mismatchName: Bank says the transaction is
disallowed if customer-supplied cardholder name does not match what's on file for the card. However, the customer-supplied cardholder billing
address may not be available. */ /* The expiredCard, overLimit, mismatchExpirationDate, mismatchCVC, mismatchAddress, and mismatchName rules all
have higher priority than bankGoodStanding. */ /* The following group of rules are additional policies of the merchant eSellWow, which it adopted/
imported from a vendor and consultant when setting up its e-store website */ /* merchantRespectBank: Merchant say a transaction is disallowed if
the bank does. */ /* merchantTrustBank: Merchant says, by default, that a transaction is allowed if the bank does. */ /* fraudAlert: Merchant
says transaction is disallowed if a trusted fraud tracking service rates the fraud risk as high for the card. */ /* trustTRW: Merchant relies on
recommenderServiceTRW for establishing such trust. */ /* badCustomer: Merchant says transaction is disallowed if its own/other experience
indicates that the customer is a bad customer to deal with. */ /* The fraudAlert and badCustomer rules both have higher priority than
merchantTrustBank. */ /* The following are additional background fact rules, known to the merchant eSellWow and the bank. */ /* eSellWow is an
established merchant for mastercard and visa. */ /* The following are additional background fact rules, known to the merchant eSellWow. */ /*
recommenderServiceTRW recommends fraudscreen */ {bankGoodStanding} authorize(?Bank,?TransactionID) :-        transactionRequest(?TransactionID,?
Merchant,?CreditCardNumber,?Amount) and        issuer(?CreditCardNumber,?Bank) and merchant(?Merchant,?Bank) and        ccInGoodStanding(?Bank,?
CreditCardNumber). {expiredCard} neg authorize(?Bank,?TransactionID) :-        transactionRequest(?TransactionID,?Merchant,?CreditCardNumber,?
Amount) and        issuer(?CreditCardNumber,?Bank) and merchant(?Merchant,?Bank) and        ccInfo(?CreditCardNumber,?Bank,?CardholderName,?
AccountLimit,                ?ExpiredFlag,?ExpirationDate,?CardholderAddress,?CVC) and        notEquals(?ExpiredFlag,"false"). {overLimit} neg
authorize(?Bank,?TransactionID) :-        transactionRequest(?TransactionID,?Merchant,?CreditCardNumber,?Amount) and        issuer(?
CreditCardNumber,?Bank) and merchant(?Merchant,?Bank) and        ccInfo(?CreditCardNumber,?Bank,?CardholderName,?AccountLimit,                ?
ExpiredFlag,?ExpirationDate,?CardholderAddress,?CVC) and        lessThan(?AccountLimit,?Amount). {mismatchExpirationDate} neg authorize(?Bank,?
TransactionID) :-        transactionRequest(?TransactionID,?Merchant,?CreditCardNumber,?Amount) and        issuer(?CreditCardNumber,?Bank) and
merchant(?Merchant,?Bank) and        ccInfo(?CreditCardNumber,?Bank,?CardholderName,?AccountLimit,                ?ExpiredFlag,?ExpirationDate,?
CardholderAddress,?CVC) and         transactionExpirationDateOf(?TransactionID,?TransactionExpirationDate) and         notEquals(?
TransactionExpirationDate,?ExpirationDate). {mismatchName} neg authorize(?Bank,?TransactionID) :-         transactionRequest(?TransactionID,?
Merchant,?CreditCardNumber,?Amount) and         issuer(?CreditCardNumber,?Bank) and merchant(?Merchant,?Bank) and         ccInfo(?
CreditCardNumber,?Bank,?CardholderName,?AccountLimit,                 ?ExpiredFlag,?ExpirationDate,?CardholderAddress,?CVC) and
transactionCardholderNameOf(?TransactionID,?TransactionCardholderName) and         notEquals(?TransactionCardholderName,?CardholderName).
{mismatchCVC} neg authorize(?Bank,?TransactionID) :-         transactionRequest(?TransactionID,?Merchant,?CreditCardNumber,?Amount) and
issuer(?CreditCardNumber,?Bank) and merchant(?Merchant,?Bank) and         ccInfo(?CreditCardNumber,?Bank,?CardholderName,?AccountLimit,
      ?ExpiredFlag,?ExpirationDate,?CardholderAddress,?CVC) and         transactionCVCOf(?TransactionID,?TransactionCVC) and notEquals(?
TransactionCVC,?CVC). {mismatchAddress} neg authorize(?Bank,?TransactionID) :-         transactionRequest(?TransactionID,?Merchant,?
CreditCardNumber,?Amount) and         issuer(?CreditCardNumber,?Bank) and merchant(?Merchant,?Bank) and         ccInfo(?CreditCardNumber,?Bank,?
CardholderName,?AccountLimit,                 ?ExpiredFlag,?ExpirationDate,?CardholderAddress,?CVC) and         transactionCardholderAddressOf(?
TransactionID,?TransactionCardholderAddress),         notEquals(?TransactionCardholderAddress,?CardholderAddress). overrides(expiredCard,
bankGoodStanding). overrides(overLimit,bankGoodStanding). overrides(mismatchExpirationDate,bankGoodStanding). overrides(mismatchName,
bankGoodStanding). overrides(mismatchCVC,bankGoodStanding). overrides(mismatchAddress,bankGoodStanding). {merchantTrustBank} authorize(?Merchant,?
TransactionID) :-         transactionRequest(?TransactionID,?Merchant,?CreditCardNumber,?Amount) and         issuer(?CreditCardNumber,?Bank) and
merchant(?Merchant,?Bank) and         authorize(?Bank,?TransactionID). {merchantRespectBank} neg authorize(?Merchant,?TransactionID) :-
transactionRequest(?TransactionID,?Merchant,?CreditCardNumber,?Amount) and         issuer(?CreditCardNumber,?Bank) and merchant(?Merchant,?Bank)
and         neg authorize(?Bank,?TransactionID). {fraudAlert} neg authorize(?Merchant,?TransactionID) :-         transactionRequest(?
TransactionID,?Merchant,?CreditCardNumber,?Amount) and         issuer(?CreditCardNumber,?Bank) and merchant(?Merchant,?Bank) and         ccInfo(?
CreditCardNumber,?Bank,?CardholderName,?AccountLimit,                 ?ExpiredFlag,?ExpirationDate,?CardholderAddress,?CVC) and
fraudRecommenderFor(?Merchant,?recommenderService) and         fraudExpert(?recommenderService,?FraudFirm) and         ccFraudRiskRating(?
FraudFirm,?CardholderName,high). {badCustomer} neg authorize(?Merchant,?TransactionID) :-         transactionRequest(?TransactionID,?Merchant,?
CreditCardNumber,?Amount) and         issuer(?CreditCardNumber,?Bank) and merchant(?Merchant,?Bank) and         ccInfo(?CreditCardNumber,?Bank,?
CardholderName,?AccountLimit,                 ?ExpiredFlag,?ExpirationDate,?CardholderAddress,?CVC) and         customerRating(?Merchant,?
CardholderName,bad). overrides(fraudAlert,merchantTrustBank). overrides(badCustomer,merchantTrustBank). fraudRecommenderFor(eSellWow,
recommenderServiceTRW). merchant(eSellWow,mastercard). merchant(eSellWow,visa). fraudExpert(recommenderServiceTRW,fraudscreen). /* The following
groups of "case" facts each specify a particular case scenario of a requested customer transaction. */ /* Joe Goya has a card in good standing,
unexpired, and the transaction amount is below the account limit. His customer rating is good. Transaction expiration date, address, and CVC are
unavailable, as is fraud alert rating. The policies thus imply that his transaction ought to be authorized by the merchant as well as by the
bank. */ /* Mary Freund has a card in good standing, unexpired, and the transaction amount is below the account limit. Her address matches, and
her fraud report and customer rating are fine. But the transaction CVC and address do not match the ones on file. Thus the policies imply that
the transaction on her card ought to be disallowed by the merchant as well as the bank. */ /* Andy Lee has a card in good standing, unexpired,
and the transaction amount is under the account limit. But his customer rating is bad. Thus the policies imply that his transaction ought to be
disallowed by the merchant (regardless of whether the bank allows it). */ transactionRequest(trans1014,eSellWow,"999912345678",70). issuer
("999912345678",mastercard). ccInfo("999912345678",mastercard,joeGoya,1100,"false","2007_08","43 Garden Drive, Cincinnati OH",702).
ccInGoodStanding(mastercard,"999912345678"). customerRating(eSellWow,joeGoya,good). transactionRequest(trans2023,eSellWow,"999987654321",410).
issuer("999987654321",visa). ccInfo("999987654321",visa,maryFreund,2400,"false","2008_02","325 Haskell Street, Seattle, WA",684). ccInGoodStanding
(visa,"999987654321"). ccFraudRiskRating(fraudscreen,maryFreund,low). customerRating(eSellWow,maryFreund,excellent). transactionCVCOf
(trans2023,524). transactionTransactionAddressOf(trans2023,"1493 Belair Place, Los Angeles, CA"). transactionRequest(trans3067,
eSellWow,"999956781234",120). issuer("999956781234",mastercard). ccInfo("999956781234",mastercard,andyLee,900,"false","2006_05","1500 Seaview
Boulevard, Daytona Beach, FL",837). ccInGoodStanding(mastercard,"999956781234"). customerRating(eSellWow,andyLee,bad).
```

The above premises (policy rules and case facts) together entail the following conclusions about the authorization of the particular requested transactions by customers Joe Goya, Mary Freund, and Andy Lee. Notice that in the case of Andy Lee's, the merchant denies authorization even though the bank approves it, because of the merchant's own customer rating info and policies.

```
/* the entailed approval vs. denial by the bank, and by the merchant, of authorization of the particular requested credit card transactions. */
authorize(mastercard, trans1014). authorize(eSellWow, trans1014). neg authorize(visa, trans2023). neg authorize(eSellWow, trans2023). authorize
(mastercard, trans3067). neg authorize(eSellWow, trans3067).
```

6 Using Defaults in Domain-Specific Service Ontologies

SWSO is a *core* service ontology and is domain-independent. However, it is often necessary to represent *domain-specific* ontologies. Such ontologies might be represented in SWSL-Rules or in SWSL-FOL. An important and frequently-used flavor of domain-specific service ontologies uses the object-oriented framework of the kind found in object-oriented programming languages and in AI frame-based systems. In such frameworks, the values of a property P of a superclass are inherited by each of its subclasses. These subclasses are known as the *specializations* of the *parent* superclass. For example, the property P might be a data attribute or a method definition. This inheritance has *default* flavor: explicitly specified information about the property for the subclass overrides -- or cancels -- the inheritance. Such default inheritance (and, by extension, such an ontology) is logically nonmonotonic and cannot be represented in SWSL-FOL. However, it can be represented in SWSL-Rules, which is object-oriented and includes inheritance as a basic feature.

To illustrate this point, we give an example of a service, *Sell Product*, which relies on such a domain-specific service ontology. The ontology describes a Sell Product business process and a some of its specializations. The form and domain of this ontology are similar to those found in the Process Handbook [*Process Handbook*, *Malone99*]. This kind of ontology is useful for business process modeling and design, and thus is useful -- along with other knowledge -- for a number of SWS tasks, including:

- service discovery [*Bernstein2002*];
- service execution [*Bernstein2000*];
- service composition;
- contract formation and negotiation [*Grosof2004e*]; and
- contract monitoring and exception handling [*Grosof2004e*].

In [*Bernstein2003*] we introduced an example about Sell Product. The example below is adapted from that. There we give an approach to representing default inheritance about process ontologies in terms of Courteous LP. Another approach to representing default inheritance in a similar spirit is shown in [*Yang02*] who use LP with NAF but without Courteous, instead defining special constructs that extend the LP KR.
Below, we give an approach similar in spirit to both of the above, that is relatively simple -- simple enough for a self contained brief presentation here -- although lacking some of the advantages and subtleties of the above two approaches.

Modeling the parent service "Sell Product"

Suppose we need to design a sales process that is used in an organization in two ways. One version is used in a mail-order business and the other in a retail store. First, we need to model a generic sales process for which one can find a template in a process repository (e.g., the Process Handbook). The "Sell product" service consists of five subtasks: "Identify potential customers," "Inform potential customers," "Obtain Order," "Deliver product ," and "Receive payment." In this example, we do not model any sequencing dependencies between the subtasks.

One way to represent this situation is to treat the main service as a class and its subtasks as attributes:

```
SellProduct[
    identifyCustomers *-> genericFindCust,
    informCustomers *-> genericInformCust,
    obtainOrder *-> genericGetOrder,
    deliverProduct *-> genericDeliver,
    receivePay *-> genericGetPay
].
```

Here the attribute names represent the names of the subtasks and the values of these attributes are the names of the actual procedures to be used to perform these tasks. These procedures can be written in a procedural programming language, such as Java, or even in SWSL-Rules. For instance,

```
 ?Product[genericInformCust] :-
         ?Product[genericFindCust -> ?Cust] and
         generateFlyer(?Cust) and
         informMailRoom.
```

Some of the formulas in the body of the above rule could be purely declarative and be defined by other rules in the knowledge base (e.g., `genericFindCust`) while others could have side effects in the physical world (such as `generateFlyer` and `informMailRoom`). Such side-effectful statements are not currently part of SWSL-Rules, but they are planned for future extensions.

Note that the method `genericFindCust` returns a *set* of customers (based on some marketing criteria), and the method `genericFindCust` is executed for *each* such customer.

Overriding and Canceling Inherited Features

Continuing the example, we specify the "Sell by mail order" and "Sell in retail store" services as subclasses of the "Sell Product" process, as shown in Figure 6.1.

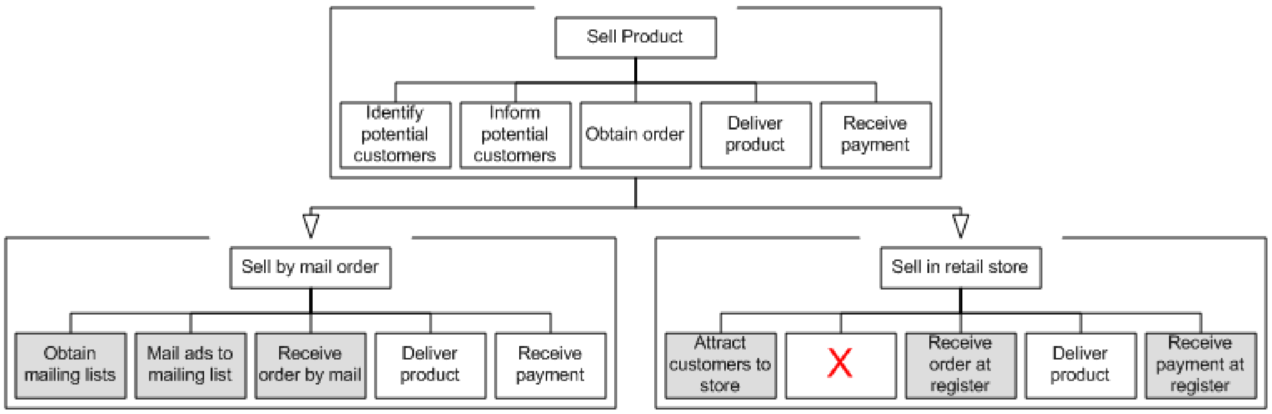

Figure 6.1: The "Sell Product" service with two subclasses. The grayed out subtasks are overwritten with more specific alternatives; the subtask with the red cross is deleted ("canceled").

This subclassing approach has several advantages. First, it provides a simple way of reusing the elements already defined in an object-oriented manner. Second, taxonomic organization of processes has been found useful for human browsing [*Malone99*].

For "Sell by mail order," inheritance of three of the subtasks of the parent service "Sell product" is overwritten by other subtasks, and two subtasks are inherited. For "Sell in retail store", one subtask is inherited, inheritance of three others is overwritten, and one subtask is "canceled" (i.e., is no longer defined for the subsprocess).

Since nonmonotonic inheritance is one of the basic concepts of SWSL-Rules, modeling of the service "Sell by mail order" is straightforward. First, we need to specify it as a subclass of `SellProduct`. Then we need to explicitly define the three overwritten subtasks and provide new values (new procedures) for them. We do not need to mention the two tasks that are inherited:

```
SellByMailOrder::SellProduct[
    identifyCustomers *-> obtainMailLists,
    informCustomers *-> junkmailAds,
    obtainOrder *-> getOrderByMail
].
```

Because of the overriding in the `SellByMailOrder` subclass, the subtask `informCustomers` is no longer performed using the previously defined method `genericInformCust`. Instead, the method `junkmailAds` is used; it could be defined in SWSL-Rules as follows:

```
?Product[junkmailAds] :-
        ?Product[obtainMailLists -> ?List] and
        ?List[address -> ?Addr] and
        affixLabelToAd(?Addr) and
        informMailRoom.
```

To model the "Sell in retain store" we first specify a new subclass of the "Sell Product" service. According to the figure, this subclass inherits the `deliverProduct` attribute, while three other attributes, `identifyCustomers`, `obtainOrder`, and `receivePay`, are overwritten. This is modeled similarly to the `SellByMailOrder` subclass:

```
SellInRetail::SellProduct[
         identifyCustomers *-> attractToBrickAndMortar,
         obtainOrder *-> getOrderAtRegister,
         receivePay *-> getPayAtRegister
].
```

The more interesting case is the cancellation of the `informCustomers` attribute. One way to achieve this is to introduce a special `null` subtask and use it to override inheritance of the `genericInformCust` procedure. A more interesting way it to take advantage of the semantics of `multiple inheritance` in SWSL-Rules according to which conflicting multiple inheritance for the same attribute makes the value undefined. To achieve this, we can introduce a family of classes parameterized by the features that need to be canceled:

```
FeatureTerminator(?Feature)[?Feature *-> null].
```

For each concrete attribute, the above statement says that the value of that attribute in the corresponding class is `null`. As a special case (when `?Feature = informCustomers`), the value of the attribute `informCustomers` in class `FeatureTerminator(informCustomers)` is null. To cancel the inheritance of `informCustomers` we now need to add the following fact:

```
SellInRetail::FeatureTerminator(informCustomers).
```

With this statement, `SellInRetail` becomes a subclass of two classes, `SellProduct` and `FeatureTerminator(informCustomers)`. Each of these classes has an explicit definition of the attribute `informCustomers`, but those definitions are in conflict. According to the semantics of inheritance in SWSL-Rules, this makes the value of `informCustomers` in class `SellInRetail` *undefined* and thus the inheritance of that attribute is "canceled."

Adding Exception Handling

Next, suppose that we need to extend the repertoire of selling services with a third process, "Sell electronically," which can be executed electronically.

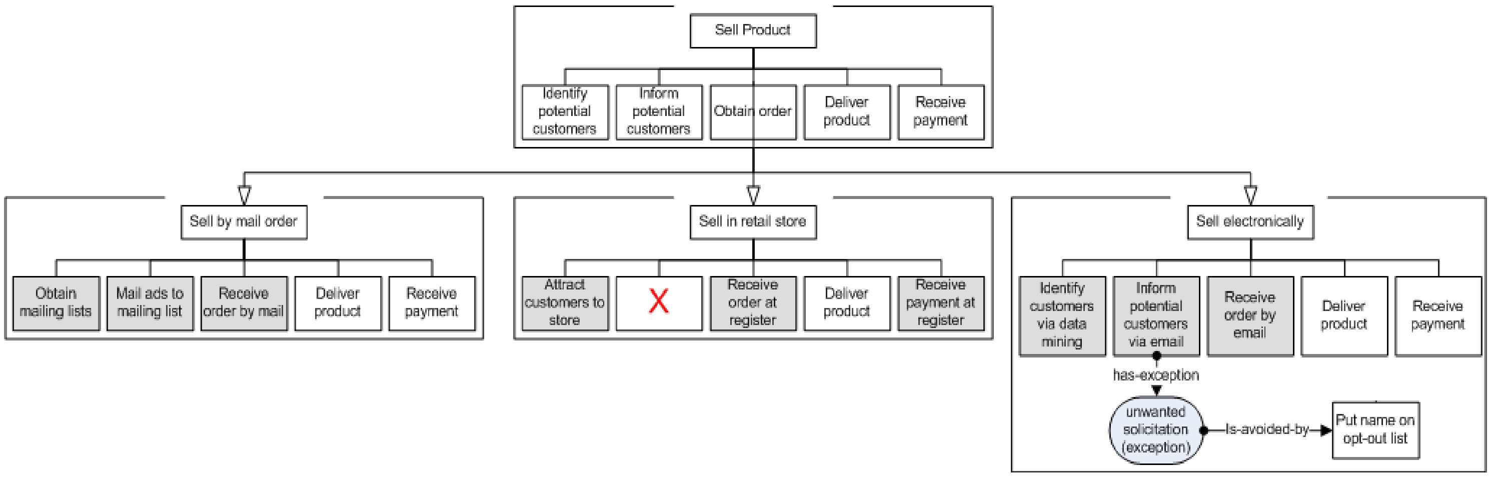

Figure 6.2: The "Sell Product" service with three subclasses; the third subclass has an exception handler.

Figure 6.2 shows that the new process has an exception attached to the second subtask "Inform potential customers by email" in order to addresses the issue of unwanted email solicitations. The exception permits people to remove themselves from the mailing list by putting their addresses on the opt-out list.

Inheritance and overriding of the attributes from the parent class is modeled as before:

```
SellElectronically::SellProduct[
        identifyCustomers *-> obtainByDataMining,
        informCustomers *-> informByEmail,
        obtainOrder *-> getOrderByEMail
].
```

The method `informByEmail` could be defined as follows:

```
?Product[informByEmail] :-
        ?Product[obtainByDataMining -> ?Email] and
        sendEmail(?Email).
```

One way to incorporate the opt-out exception to the general policy is by adding the premise `naf optOutList(?Email)` to the body of the above rule. However, this approach is not modular, since it requires modification of the existing rules (the method `informByEmail` may have already been defined). Even if it were acceptable to make this change now, further changes to the opt-out policy might require additional changes to the existing rules. A more scalable approach is to express the above opt-out exception using constraints, and this is where the *mutex* primitive of the Courteous layer of SWSL-Rules comes in:

```
!- sendEmail(?Email) and optOut(?Email).
```

This constraint says that `sendEmail(?Email)` and `optOut(?Email)` cannot be true together for the same individual. For people who put their names on the opt-out list, `optOut(?Email)` will be true and, therefore, `sendEmail(?Email)` will be false. On the other hand, for an email address, e, that is not found on the opt-out list, the corresponding predicate predicate `optOut(e)` will not be provable and, by negation-as-failure, `optOut(e)` will be assumed false. Therefore, sending email to that address will not be blocked.

7 Glossary

Activity
: *Activity*. In the formal PSL ontology, the notion of activity is a basic construct, which corresponds intuitively to a kind of (manufacturing or processing) activity. In PSL, an activity may have associated *occurrences*, which correspond intuitively to individual instances or executions of the activity. (We note that in PSL an activity is not a class or type with occurrences as members; rather, an activity is an object, and occurrences are related to this object by the binary predicate `occurrence_of`.) The occurrences of an activity may impact fluents (which provide an abstract representation of the "real world"). In FLOWS, with each service there is an associated activity (called the "service activity" of that service). The service activity may specify aspects of the internal process flow of the service, and also aspects of the messaging interface of that service to other services.

Channel
: *Channel*. In FLOWS, a channel is a formal conceptual object, which corresponds intuitively to a repository and conduit for messages. The FLOWS notion of channel is quite primitive, and under various restrictions can be used to model the form of channel or message-passing as found in web services standards, including WSDL, BPEL, WS-Choreography, WSMO, and also as found in several research investigations, including process algrebras.

FLOWS
: ***F****irst-order* ***L****ogic* ***O****ntology for* ***W****eb* ***S****ervices*. FLOWS, also known as SWSO-FOL, is the first-order logic version of the Semantic Web Services Ontology. FLOWS is an extension of the PSL-OuterCore ontology, to incorporate the fundamental aspects of (web and other electronic) services, including service descriptors, the service activity, and the service grounding.

Fluent
: *Fluent*. In FLOWS, following PSL and the situation calculii, a fluent is a first-order logic term or predicate whose value may vary over time. In a first-order model of a FLOWS theory, this being a model of PSL-OuterCore, time is represented as a discrete linear sequence of *timees*, and fluents has a value for each time in this sequence.

Grounding
: *Grounding*. The SWSO concepts for describing service activities, and the instantiations of these concepts that describe a particular service activity, are *abstract* specifications, in the sense that they do not specify the details of particular message formats, transport protocols, and network addresses by which a Web service is accessed. The role of the *grounding* is to provide these more concrete details. A substantial portion of the grounding can be acheived by mapping SWSO concepts into corresponding WSDL constructs. (Additional grounding, e.g., of some process-related aspects of SWSO, might be acheived using other standards, such as BPEL.)

Message
: *Message*. In FLOWS, a message is a formal conceptual object, which corresponds intuitively to a single message that is created by a service occurrence, and read by zero or more service occurrences. The FLOWS notion of message is quite primitive, and under various restrictions can be used to model the form of messages as found in web services standards, including

WSDL (1.0 and 2.0), BPEL, WS-Choreography, WSMO, and also as found in several research investigations. A message has a *payload*, which corresponds intuitively to the body or contents of the message. In FLOWS emphasis is placed on the knowledge that is gained by a service occurrence when reading a message with a given payload (and the knowledge needed to create that message.

Occurrence

Occurence (of a service). In FLOWS, a service *S* has an associated FLOWS activity *A* (which generalizes the notion of PSL activity). An *occurrence* of *S* is formally a PSL occurrence of the activity *A*. Intuitively, this occurrence corresponds to an instance or execution (from start to finish) of the activity *A*, i.e., of the process associated with service *S*. As in PSL, an occurrence has a starting time time and an ending time.

PSL

Process Specification Language. The Process Specification Language (PSL) is a formally axiomatized ontology [*Gruninger03a*, *Gruninger03b*] that has been standardized as ISO 18629. PSL provides a layered, extensible ontology for specifying properties of processes. The most basic PSL constructs are embodied in PSL-Core; and PSL-OuterCore incorporates several extensions of PSL-Core that includes several useful constructs. (An overview of concepts in PSL that are relevant to FLOWS is given in Section 6 of the Semantic Web Services Ontology document.)

QName

Qualified name. A pair (*URI*, *local-name*). The *URI* represents a namespace and *local-name* represents a name used in an XML document, such as a tag name or an attribute name. In XML, QNames are syntactically represented as *prefix*:*local-name*, where *prefix* is a macro that expands into a concrete URI. See Namespaces in XML for more details.

ROWS

***R**ules **O**ntology for **W**eb **S**ervices*. ROWS, also known as SWSO-Rules, is the rules-based version of the Semantic Web Services Ontology. ROWS is created by a relatively straight-forward, almost faithful, transformation of FLOWS, the First-order Logic Ontology for Web Services. As with FLOWS, ROWS incorporates fundamental aspects of (web and other electronic) services, including service descriptors, the service activity, and the service grounding. ROWS enables a rules-based specification of a family of services, including both the underlying ontology and the domain-specific aspects.

Service

(Formal) Service. In FLOWS, a service is a conceptual object, that corresponds intuitively to a web service (or other electronically accessible service). Through binary predicates a service is associated with various service descriptors (a.k.a. non-functional properties) such as Service Name, Service Author, Service URL, etc.; an *activity* (in the sense of PSL) which specifies intuitively the process model associated with the service; and a *grounding*.

Service contract

Describes an agreement between the service requester and service provider, detailing requirements on a service occurrence or family of service occurrences.

Service descriptor

Service Descriptor. This is one of several non-functional properties associated with services. The Service Descriptors include Service Name, Service Author, Service Contract Information, Service Contributor, Service Description, Service URL, Service Identifier, Service Version, Service Release Date, Service Language, Service Trust, Service Subject, Service Reliability, and Service Cost.

Service offer description

Describes an abstract service (i.e. not a concrete instance of the service) provided by a service provider agent.

Service requirement description

Describes an abstract service required by a service requester agent, in the context of service discovery, service brokering, or negotiation.

sQName

Serialized QName. A serialized QName is a shorthand representation of a URI. It is a macro that expands into a full-blown URI. sQNames are *not* QNames: the former are URIs, while the latter are pairs (*URI*, *local-name*). Serialized QNames were originally introduced in RDF as a notation for shortening URI representation. Unfortunately, RDF introduced confusion by adopting the term QName for something that is different from QNames used in XML. To add to the confusion, RDF uses the syntax for sQNames that is identical to XML's syntax for QNames. SWSL distinguishes between QNames and sQNames, and uses the syntax *prefix#local-name* for the latter. Such an sQName expands into a full URI by concatenating the value of *prefix* with *local-name*.

URI

Universal Resource Identifier. A symbol used to locate resources on the Web. URIs are defined by IETF. See Uniform Resource Identifiers (URI): Generic Syntax for more details. Within the IETF standards the notion of URI is an extension and refinement of the notions of Uniform Resource Locator (URL) and Relative Uniform Resource Locators.

8 References

[Berardi03]

Automatic composition of e-services that export their behavior. D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. In *Proc. 1st Int. Conf. on Service Oriented Computing (ICSOC)*, volume 2910 of *LNCS*, pages 43--58, 2003.

[Bernstein2000]

How can cooperative work tools support dynamic group processes? Bridging the specificity frontier. A. Bernstein. In *Proc. Computer Supported Cooperative Work (CSCW'2000)*, 2000.

[Bernstein2002]

Towards High-Precision Service Retrieval. A. Bernstein, and M. Klein. In *Proc. of the first International Semantic Web Conference (ISWC'2002)*, 2002.

[Bernstein2003]

Beyond Monotonic Inheritance: Towards Semantic Web Process Ontologies. A. Bernstein and B.N. Grosof (NB: authorship sequence is alphabetic). Working Paper, Aug. 2003. Available at: http://ebusiness.mit.edu/bgrosof/#beyond-mon-inh-basic.

[Bonner93]

Database Programming in Transaction Logic. A.J. Bonner, M. Kifer, M. Consens. *Proceedings of the 4-th Intl.~Workshop on Database Programming Languages*, C. Beeri, A. Ohori and D.E. Shasha (eds.), 1993. In Springer-Verlag Workshops in Computing Series, Feb. 1994: 309-337.

[Bonner98]

A Logic for Programming Database Transactions. A.J. Bonner, M. Kifer. Logics for Databases and Information Systems, J. Chomicki and G. Saake (eds.). Kluwer Academic Publishers, 1998: 117-166.

[Bruijn05]

Web Service Modeling Ontology (WSMO). J. de Bruijn, C. Bussler, J. Domingue, D. Fensel, M. Hepp, M. Kifer, B. König-Ries, J. Kopecky, R. Lara, E. Oren, A. Polleres, J. Scicluna, M. Stollberg. *DERI Technical Report.*

[BPML 1.0]

A. Arkin. Business Process Modeling Language. BPMI.org, 2002

[BPEL 1.1]

Business Process Execution Language for Web Services, Version 1.1 . S. Thatte, editor. OASIS Standards Specification, May 5, 2003.

[Bultan03]

Conversation specification: A new approach to design and analysis of e-service composition. T. Bultan, X. Fu, R. Hull, and J. Su. In *Proc. Int. World Wide Web Conf. (WWW)*, May 2003.

[Chang73]

Symbolic Logic and Mechanical Theorem Proving. C.L. Chang and R.C.T. Lee. Academic Press, 1973.

[Chen93]

HiLog: A Foundation for Higher-Order Logic Programming. W. Chen, M. Kifer, D.S. Warren. Journal of Logic Programming, 15:3, February 1993, 187-230.

[Chimenti89]

Towards an Open Architecture for LDL. D. Chimeti, R. Gamboa, R. Krishnamurthy, VLDB Conference, 1989: 195-203.

[deGiacomo00]
ConGolog, A Concurrent Programming Language Based on the Situation Calculus. G. de Giacomo, Y. Lesperance, and H. Levesque. Artificial Intelligence, 121(1--2):109--169, 2000.

[Fu04]
WSAT: A Tool for Formal Analysis of Web Services. X. Fu, T. Bultan, and J. Su. *16th International Conference on Computer Aided Verification (CAV)*, July 2004.

[Frohn94]
Access to Objects by Path Expressions and Rules. J. Frohn, G. Lausen, H. Uphoff. Intl. Conference on Very Large Databases, 1994, pp. 273-284.

[Gosling96]
*The Java language specification.*Gosling, James, Bill Joy, and Guy L. Steele. 1996. Reading, Mass.: Addison-Wesley.

[Grosof99a]
A Courteous Compiler From Generalized Courteous Logic Programs To Ordinary Logic Programs. B.N. Grosof. IBM Report included as part of documentation in the IBM CommonRules 1.0 software toolkit and documentation, released on http://alphaworks.ibm.com. July 1999. Also available at: http://ebusiness.mit.edu/bgrosof/#gclp-rr-99k.

[Grosof99b]
A Declarative Approach to Business Rules in Contracts. B.N. Grosof, J.K. Labrou, and H.Y. Chan. Proceedings of the 1st ACM Conference on Electronic Commerce (EC-99). Also available at: http://ebusiness.mit.edu/bgrosof/#econtracts+rules-ec99.

[Grosof99c]
DIPLOMAT: Compiling Prioritized Default Rules Into Ordinary Logic Programs (Extended Abstract of Intelligent Systems Demonstration). B.N. Grosof. IBM Research Report RC 21473, May 1999. Extended version of 2-page refereed conference paper appearing in Proceedings of the National Conference on Artificial Intelligence (AAAI-99), 1999. Also available at: http://ebusiness.mit.edu/bgrosof/#cr-ec-demo-rr-99b.

[Grosof2003a]
Description Logic Programs: Combining Logic Programs with Description Logic. B.N. Grosof, I. Horrocks, R. Volz, and S. Decker. Proceedings of the 12th International Conference on the World Wide Web (WWW-2003). Also available at: http://ebusiness.mit.edu/bgrosof/#dlp-www2003.

[Grosof2004a]
Representing E-Commerce Rules Via Situated Courteous Logic Programs in RuleML. B.N. Grosof. Electronic Commerce Research and Applications, 3:1, 2004, 2-20. Preprint version is also available at: http://ebusiness.mit.edu/bgrosof/#.

[Grosof2004b]
SweetRules: Tools for Semantic Web Rules and Ontologies, including Translation, Inferencing, Analysis, and Authoring. B.N. Grosof, M. Dean, S. Ganjugunte, S. Tabet, C. Neogy, and D. Kolas. http://sweetrules.projects.semwebcentral.org. Software toolkit and documentation. Version 2.0, Dec. 2004.

[Grosof2004c]
Hypermonotonic Reasoning: Unifying Nonmonotonic Logic Programs with First Order Logic. B.N. Grosof. http://ebusiness.mit.edu/bgrosof/#HypermonFromPPSWR04InvitedTalk. Slides from Invited Talk at Workshop on Principles and Practice of Semantic Web Reasoning (PPWSR04), Sep. 2004; revised Oct. 2004. Paper in preparation.

[Grosof2004d]
SweetPH: Using the Process Handbook for Semantic Web Services. B.N. Grosof and A. Bernstein. http://ebusiness.mit.edu/bgrosof/#SweetPHSWSLF2F1204Talk. Slides from Presentation at SWSL Meeting, Dec. 9-10, 2004. *Note: Updates the design in the 2003 Working Paper "Beyond Monotonic Inheritance: Towards Semantic Web Process Ontologies" and describes implementation.*

[Grosof2004e]
SweetDeal: Representing Agent Contracts with Exceptions using Semantic Web Rules, Ontologies, and Process Descriptions. B.N. Grosof and T.C. Poon. International Journal of Electronic Commerce (IJEC), 8(4):61-98, Summer 2004 Also available at: http://ebusiness.mit.edu/bgrosof/#sweetdeal-exceptions-ijec.

[Grosof2004f]
Semantic Web Rules with Ontologies, and their E-Business Applications. B.N. Grosof and M. Dean. Slides of Conference Tutorial (3.5-hour) at the 3rd International Semantic Web Conference (ISWC-2004). Available at: http://ebusiness.mit.edu/bgrosof/#ISWC2004RulesTutorial.

[Gruninger03a]
A Guide to the Ontology of the Process Specification Language. M. Gruninger. *Handbook on Ontologies in Information Systems.* R. Studer and S. Staab (eds.). Springer Verlag, 2003.

[Gruninger03b]
Process Specification Language: Principles and Applications. M. Gruninger and C. Menzel. *AI Magazine,* 24:63-74, 2003.

[Gruninger03c]
Applications of PSL to Semantic Web Services. M. Gruninger. *Workshop on Semantic Web and Databases. Very Large Databases Conference, Berlin.*

[Hayes04]
RDF Model Theory. Hayes, P. W3C, February 2004.

[Helland05]
Data on the Outside Versus Data on the Inside. P. Helland. *Proc. 2005 Conf. on Innovative Database Research (CIDR)*, January, 2005.

[Hull03]
E-Services: A Look Behind the Curtain. R. Hull, M. Benedikt, V. Christophides, J. Su. *Proc. of the ACM Symp. on Principles of Database Systems (PODS)*, San Diego, June, 2003.

[Kifer95]
Logical Foundations of Object-Oriented and Frame-Based Languages, M. Kifer, G. Lausen, J. Wu. Journal of ACM, 1995, 42, 741-843.

[Kifer04]
A Logical Framework for Web Service Discovery, M. Kifer, R. Lara, A. Polleres, C. Zhao. Semantic Web Services Workshop, November 2004, Hiroshima, Japan.

[Klein00a]
Towards a Systematic Repository of Knowledge About Managing Collaborative Design Conflicts. Klein, Mark. 2000. Proceedings of the Conference on Artificial Intelligence in Design. Boston, MA, USA.

[Klein00b]
A Knowledge-Based Approach to Handling Exceptions in Workflow Systems. Klein, Mark, and C. Dellarocas. 2000. Computer Supported Cooperative Work: The Journal of Collaborative Computing 9:399-412.

[Lloyd87]
Foundations of logic programming (second, extended edition). J. W. Lloyd. Springer series in symbolic computation. Springer-Verlag, New York, 1987.

[Lindenstrauss97]
Automatic Termination Analysis of Logic Programs. N. Lindenstrauss and Y. Sagiv. International Conference on Logic Programming (ICLP), 1997.

[Maier81]
Incorporating Computed Relations in Relational Databases. D. Maier, D.S. Warren. SIGMOD Conference, 1981: 176-187.

[Malone99]
Tools for inventing organizations: Toward a handbook of organizational processes. T. W. Malone, K. Crowston, J. Lee, B. Pentland, C. Dellarocas, G. Wyner, J. Quimby, C. Osborne, A. Bernstein, G. Herman, M. Klein, E. O'Donnell. *Management Science*, 45(3), pages 425--443, 1999.

[McIlraith01]
Semantic Web Services. *IEEE Intelligent Systems*, Special Issue on the Semantic Web, S. McIlraith, T.Son and H. Zeng. 16(2):46--53, March/April, 2001.

[Milner99]
Communicating and Mobile Systems: The π-Calculus. R. Milner. Cambridge University Press, 1999.

[Narayanan02]
Simulation, Verification and Automated Composition of Web Services. S. Narayanan and S. McIlraith. In *Proceedings of the Eleventh International World Wide Web Conference (WWW-11)*, May, 2002.

[Ontobroker]
Ontobroker 3.8. Ontoprise, GmbH.

[OWL Reference]
OWL Web Ontology Language 1.0 Reference. Mike Dean, Dan Connolly, Frank van Harmelen, James Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. W3C Working Draft 12 November 2002. Latest version is available at http://www.w3.org/TR/owl-ref/.

[OWL-S 1.1]
OWL-S: Semantic Markup for Web Services. David Martin, editor. Technical Overview (associated with OWL-S Release 1.1).

[Papazoglou03]
Service-Oriented Computing: Concepts, Characteristics and Directions. M.P. Papazoglou. Keynote for the 4th International Conference on Web Information Systems Engineering (WISE 2003), December 10-12, 2003.

[Perlis85]
Languages with Self-Reference I: Foundations. D. Perlis. Artificial Intelligence, 25, 1985, 301-322.

[Preist04]
A Conceptual Architecture for Semantic Web Services, C. Preist, 1993. In Proceedings of Third International Semantic Web Conference, Nov. 2004: 395-409.

[Reiter01]
Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems. Raymond Reiter. MIT Press. 2001

[Scherl03]
Knowledge, Action, and the Frame Problem. R. B. Scherl and H. J. Levesque. *Artificial Intelligence*, Vol. 144, 2003, pp. 1-39.

[Singh04]
Protocols for Processes: Programming in the Large for Open Systems. M. P. Singh, A. K. Chopra, N. V. Desai, and A. U. Mallya. *Proc. of the 19th Annual ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Vancouver, October 2004.

[SWSL Requirements]
Semantic Web Services Language Requirements. B. Grosof, M. Gruninger, et al, editors. White paper of the Semantic Web Services Language Committee.

[UDDI v3.02]
Universal Description, Discovery and Integration (UDDI) protocol. S. Thatte, editor. OASIS Standards Specification, February 2005.

[VanGelder91]
The Well-Founded Semantics for General Logic Programs. A. Van Gelder, K.A. Ross, J.S. Schlipf. Journal of ACM, 38:3, 1991, 620-650.

[WSCL 1.0]
Web Services Conversation Language (WSCL) 1.0. A. Banerji et al. W3C Note, March 14, 2002.

[WSDL 1.1]
Web Services Description Language (WSDL) 1.1. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. W3C Note, March 15, 2001.

[WSDL 2.0]
Web Services Description Language (WSDL) 2.0 -- Part 1: Core Language. R. Chinnici, M. Gudgin, J.-J. Moreau, J. Schlimmer, and S. Weerawarana. W3C Working Draft, August 3, 2004.

[WSDL 2.0 Primer]
Web Services Description Language (WSDL) Version 2.0 -- Part 0: Primer. D. Booth, C. Liu, editors. W3C Working Draft, 21 December 2004.

[WS-Choreography]
Web Services Choreography Description Language Version 1.0. N. Kavantzas, D. Burdett, et. al., editors. W3C Working Draft, December 17, 2004.

[XSLT]
XSL Transformations (XSLT) Version 1.0. J. Clark, editor. W3C Recommendation, 16 November 1999.

[XQuery 1.0]
XQuery 1.0: An XML Query Language. S. Boag, D. Chamberlin, et al, editors. W3C Working Draft 04 April 2005.

[Yang02]
Well-Founded Optimism: Inheritance in Frame-Based Knowledge Bases. G. Yang, M. Kifer. Intl. Conference on Ontologies, DataBases, and Applications of Semantics for Large Scale Information Systems (ODBASE), October 2002.

[Yang03]
Reasoning about Anonymous Resources and Meta Statements on the Semantic Web. G. Yang, M. Kifer. Journal on Data Semantics, Lecture Notes in Computer Science 2800, Springer Verlag, September 2003, 69-98.

[Yang04]
FLORA-2 User's Manual. G. Yang, M. Kifer, C. Zhao, V. Chowdhary. 2004.

Appendix A: PSL in SWSL

This is Appendix A of the Semantic Web Services Ontology (SWSO) document.

Here we give the SWSL-FOL and SWSL-Rules formulations of selected theories of the Process Specification Language (PSL), including PSL core and five theories from the outer core. (These are the theories upon which SWSO builds.) Each of these theories is given in its entirety.

The original PSL declarations and axioms, expressed in KIF, are retained here, precisely as given on the PSL Web site. For each axiom of PSL, we give an equivalent axiom expressed in SWSL-FOL, and a (possibly weakened) set of rules expressed in SWSL-Rules. The SWSL-Rules axiomatizations were derived according to the translation approach described in Section 3 of the SWSL document.

A.1 PSL Core

Extension Name: psl_core.th
Theories Required by this Extension: None
Definitional Extensions Required by this Extension: None

Primitive Lexicon

Relations:

KIF

- `(object ?x)`
- `(activity ?a)`
- `(activity_occurrence ?occ)`
- `(timepoint ?t)`
- `(before ?t1 ?t2)`
- `(occurrence_of ?occ ?a)`
- `(participates_in ?x ?occ ?t)`

SWSL-FOL and SWSL-Rules

- `object(?x)`
- `activity(?a)`
- `activity_occurrence(?occ)`
- `timepoint(?t)`
- `before(?t1, ?t2)`
- `occurrence_of(?occ, ?a)`
- `participates_in(?x, ?occ, ?t)`

Functions:

KIF

- `(beginof ?occ)`
- `(endof ?occ)`

SWSL-FOL and SWSL-Rules

- `beginof(?occ)`
- `endof(?occ)`

Constants:

KIF

- `inf+`
- `inf-`

SWSL-FOL and SWSL-Rules

- `plusInf`
- `minusInf`

Defined Lexicon

Relations:

KIF

- `(between ?t1 ?t2 ?t3)`
- `(beforeEq ?t1 ?t2)`
- `(betweenEq ?t1 ?t2 ?t3)`
- `(exists_at ?x ?t)`
- `(is_occurring_at ?occ ?t)`

SWSL-FOL and SWSL-Rules

- `between(?t1, ?t2, ?t3)`
- `beforeEq(?t1, ?t2)`
- `betweenEq(?t1, ?t2, ?t3)`
- `exists_at(?x, ?t)`
- `is_occurring_at(?occ, ?t)`

Axioms

Axiom 1 *The before relation only holds between timepoints.*

KIF

```
(forall (?t1 ?t2)
        (implies  (before ?t1 ?t2)
                  (and  (timepoint ?t1)
                        (timepoint ?t2))))
```

SWSL-FOL

```
forall ?t1 ?t2
  ( before(?t1, ?t2) ==> timepoint(?t1) and timepoint(?t2) ).
```

SWSL-Rules

```
neg before(?t1, ?t2) :- neg timepoint(?t1).
timepoint(?t1) :- before(?t1, ?t2).

neg before(?t1, ?t2) :- neg timepoint(?t2).
timepoint(?t2) :- before(?t1, ?t2).
```

Axiom 2 *The before relation is a total ordering.*

KIF

```
(forall (?t1 ?t2)
        (implies  (and  (timepoint ?t1)
                        (timepoint ?t2))
                  (or   (= ?t1 ?t2)
                        (before ?t1 ?t2)
                        (before ?t2 ?t1))))
```

SWSL-FOL

```
forall ?t1 ?t2
  ( timepoint(?t1) and timepoint(?t2) ==>
    ?t1 :=: ?t2 or before(?t1, ?t2) or before(?t2, ?t1) ).
```

SWSL-Rules

```
neg timepoint(?t1) :- timepoint(?t2) and neg?t1 :=: ?t2 and neg before(?t1,?t2) and neg before(?
t2,?t1).
neg timepoint(?t2) :- timepoint(?t1) and neg ?t1 :=: ?t2 and neg before(?t1,?t2) and neg before(?
t2,?t1).
?t1 :=: ?t2 :- timepoint(?t1) and timepoint(?t2) and neg before(?t1,?t2) and neg before(?t2,?t1).
before(?t1,?t2) :- timepoint(?t1) and timepoint(?t2) and neg ?t1 :=: ?t2 and neg before(?t2,?t1).
before(?t2,?t1) :- timepoint(?t1) and timepoint(?t2) and neg ?t1 :=: ?t2 and neg before(?t1,?t2).
```

Note: The following could cause problems, because time is assumed to be some built-in theory, such as the integers. For integers it doesn't make sense to assert additional equalities, since they could contradict the built-in theory.

```
  ?t1 :=: ?t2 :- timepoint(?t1), timepoint(?t2),
                 neg before(?t1,?t2), neg before(?t2, ?t1).
```

Axiom 3 *The before relation is irreflexive.*

KIF

```
(forall (?t1)
        (not (before ?t1 ?t1)))
```

SWSL-FOL

```
forall ?t1
  ( neg before(?t1, ?t1) ).
```

SWSL-Rules

```
neg before(?t1, ?t1) :- timepoint(?t1).
```

Notes:

1. Another possibility is to use an empty head:

   ```
   :- before(?t1, ?t1).
   ```

2. `neg before(?t1,?t1)`

   would not be a "safe" rule, i.e., the bindings cannot be supplied by the body, which is inefficient for exhaustive forward reasoning and also not operational for backward reasoning when the goal contains a variable. Many engines do not support unsafe rules at all.

   We address this issue by adding "typing" conditions (i.e., the timepoint atom) to the body that provide bindings.
3. This axiom could be expressed in Prolog in this fashion:

   ```
   before(X,Y) :- X==Y, !, fail.
   ```

Axiom 4 *The before relation is transitive.*

KIF

```
(forall (?t1 ?t2 ?t3)
        (implies  (and  (before ?t1 ?t2)
                        (before ?t2 ?t3))
                  (before ?t1 ?t3)))
```

SWSL-FOL

```
forall ?t1 ?t2 ?t3
  ( before(?t1, ?t2) and before(?t2, ?t3) ==> before(?t1, ?t3) ).
```

SWSL-Rules

```
neg before(?t1, ?t2) :- before(?t2, ?t3) and neg before(?t1, ?t3).
neg before(?t2, ?t3) :- before(?t1, ?t2) and neg before(?t1, ?t3).
before(?t1, ?t3) :- before(?t1, ?t2) and before(?t2, ?t3).
```

Axiom 5 *The timepoint inf- is before all other timepoints.*

KIF

```
(forall (?t)
(implies  (and (timepoint ?t) (not (= ?t inf-)))
          (before inf- ?t))
```

SWSL-FOL

```
forall ?t
  ( timepoint(?t) and neg (?t :=: minusInf) ==> before(minusInf, ?t) ).
```

SWSL-Rules

```
neg timepoint(?t) :- neg ?t :=: minusInf and neg before(minusInf, ?t).
?t :=: minusInf :- timepoint(?t) and neg before(minusInf, ?t).
before(minusInf, ?t) :- timepoint(?t) and neg ?t :=: minusInf.
```

Axiom 6 *Every other timepoint is before inf+.*

KIF

```
(forall (?t)
(implies  (and (timepoint ?t) (not (= ?t inf+)))
          (before ?t inf+))
```

SWSL-FOL

```
forall ?t
  ( timepoint(?t) and neg (?t :=: plusInf) ==> before(?t, plusInf) ).
```

SWSL-Rules

```
neg timepoint(?t) :- neg ?t :=: plusInf and neg before(?t, plusInf).
?t :=: plusInf :- timepoint(?t) and neg before(?t, plusInf).
before(?t, plusInf) :- timepoint(?t) and neg ?t :=: plusInf.
```

Axiom 7 *Given any timepoint t other than inf-, there is a timepoint between inf- and t.*

KIF

```
(forall (?t)
        (implies  (and  (timepoint ?t)
                        (not (= ?t inf-)))
                  (exists (?u)
                        (between inf- ?u ?t))))
```

SWSL-FOL

```
forall ?End
  ( timepoint(?End) and neg (?End :=: minusInf) ==>
    exists ?u ( between(minusInf, ?u, ?End) ) ).
```

SWSL-Rules

```
neg timepoint(?End) :- neg ?End :=: minusInf and neg between(minusInf, _#1(?End), ?End).
?End :=: minusInf :- timepoint(?End) and neg between(minusInf, _#1(?End), ?End).
between(minusInf, _#1(?End), ?End) :- timepoint(?End) and neg ?End :=: minusInf.
```

Axiom 8 *Given any timepoint t other than inf+, there is a timepoint between t and inf+.*

KIF

```
(forall (?t)
        (implies  (and  (timepoint ?t)
                        (not (= ?t inf+)))
                  (exists (?u)
                        (between ?t ?u inf+))))
```

SWSL-FOL

```
forall ?Beg
  ( timepoint(?Beg) and neg (?Beg :=: plusInf) ==>
    exists ?u ( between(?Beg, ?u, plusInf) ) ).
```

SWSL-Rules

```
neg timepoint(?Beg)  :- neg ?Beg :=: plusInf and neg between(?Beg, _#1(?Beg), plusInf).
?Beg :=: plusInf :- timepoint(?Beg) and neg between(?Beg, _#1(?Beg), plusInf).
between(?Beg, _#1(?Beg), plusInf) :- timepoint(?Beg) and neg ?Beg :=: plusInf.
```

Axiom 9 *Everything is either an activity, activity occurrence, timepoint, or object.*

KIF

```
(forall (?x)
        (or     (activity ?x)
                (activity_occurrence ?x)
                (timepoint ?x)
                (object ?x)))
```

SWSL-FOL

```
forall ?x
  ( activity(?x) or activity_occurrence(?x) or timepoint(?x) or object(?x) ).
```

SWSL-Rules

```
activity(?x) :- neg activity_occurrence(?x) and neg timepoint(?x) and neg object(?x).
activity_occurrence(?x) :- neg activity(?x) and neg timepoint(?x) and neg object(?x).
timepoint(?x) :- neg activity(?x) and neg activity_occurrence(?x) and neg object(?x).
object(?x) :- neg activity(?x) and neg activity_occurrence(?x) and neg timepoint(?x).
```

Axiom 10 *Objects, activities, activity occurrences, and timepoints are all distinct kinds of things.*

KIF

```
(forall (?x)
(and (implies (activity ?x)
         (not (or (activity_occurrence ?x) (object ?x) (timepoint ?x))))
     (implies (activity_occurrence ?x)
         (not (or (object ?x) (timepoint ?x))))
     (implies (object ?x)
         (not (timepoint ?x))))
```

SWSL-FOL

```
forall ?x
  ( ( activity(?x) ==>
      neg ( activity_occurrence(?x) or object(?x) or timepoint(?x) ) )
    and
    ( activity_occurrence(?x) ==>
      neg ( object(?x) or timepoint(?x) ) )
    and
    ( object(?x) ==>
      neg timepoint(?x) ) ).
```

SWSL-Rules

```
neg activity(?x) :- activity_occurrence(?x).
neg activity_occurrence(?x) :- activity(?x).

neg activity(?x) :- object(?x).
neg object(?x) :- activity(?x).

neg activity(?x) :- timepoint(?x).
neg timepoint(?x) :- activity(?x).

neg activity_occurrence(?x) :- object(?x).
neg object(?x) :- activity_occurrence(?x).

neg activity_occurrence(?x) :- timepoint(?x).
neg timepoint(?x) :- activity_occurrence(?x).

neg object(?x) :- timepoint(?x).
neg timepoint(?x) :- object(?x).
```

Axiom 11 *The occurrence relation only holds between activities and activity occurrences.*

KIF

```
(forall (?a ?occ)
        (implies  (occurrence_of ?occ ?a)
                  (and  (activity ?a)
                        (activity_occurrence ?occ))))
```

SWSL-FOL

```
forall ?a, ?occ
  ( occurrence_of(?occ, ?a) ==>
    ( activity(?a) and activity_occurrence(?occ) ) ).
```

SWSL-Rules

```
neg occurrence_of(?occ ?a) :- neg activity(?a).
activity(?a) :- occurrence_of(?occ ?a).

neg occurrence_of(?occ ?a) :- neg activity_occurrence(?occ).
activity_occurrence(?occ) :- occurrence_of(?occ ?a).
```

Axiom 12 *Every activity occurrence is the occurrence of some activity.*

KIF

```
(forall (?occ)
        (implies  (activity_occurrence ?occ)
                  (exists (?a)
                        (and    (activity ?a)
                                (occurrence_of ?occ ?a)))))
```

SWSL-FOL

```
forall ?occ
  ( activity_occurrence(?occ) ==>
    exists ?a
     ( activity(?a) and occurrence_of(?occ, ?a) ) ).
```

SWSL-Rules

```
neg activity_occurrence(?occ) :- neg activity(_#1(?occ)).
activity(_#1(?occ)) :- activity_occurrence(?occ).

neg activity_occurrence(?occ) :- neg occurrence_of(?occ, _#1(?occ)).
occurrence_of(?occ, _#1(?occ)) :- activity_occurrence(?occ).
```

Axiom 13 *An activity occurrence is associated with a unique activity.*

KIF

```
(forall (?occ ?a1 ?a2)
        (implies (and   (occurrence_of ?occ ?a1)
                        (occurrence_of ?occ ?a2))
                 (= ?a1 ?a2))))
```

SWSL-FOL

```
forall ?occ, ?a1, ?a2
  ( occurrence_of(?occ ?a1) and occurrence_of(?occ, ?a2) ==>
    ?a1 :=: ?a2 ).
```

SWSL-Rules

```
neg occurrence_of(?occ, ?a1) :- occurrence_of(?occ, ?a2) and neg ?a1 :=: ?a2.
neg occurrence_of(?occ, ?a2) :- occurrence_of(?occ, ?a1) and neg ?a1 :=: ?a2.
?a1 :=: ?a2 :- occurrence_of(?occ, ?a1) and occurrence_of(?occ, ?a2).
```

Axiom 14 *The begin and end of an activity occurrence or object are timepoints.*

KIF

```
(forall (?a ?x)
        (implies  (or   (occurrence_of ?x ?a)
                        (object ?x))
                  (and  (timepoint (beginof ?x))
                        (timepoint (endof ?x)))))
```

SWSL-FOL

```
forall ?a, ?x
  ( ( occurrence_of(?x, ?a) or object(?x) ) ==>
    ( timepoint(beginof(?x)) and timepoint(endof(?x)) ) ).
```

SWSL-Rules

```
neg occurrence_of(?x,?a) :- neg timepoint(beginof(?x)).
timepoint(beginof(?x)) :- occurrence_of(?x,?a).

neg occurrence_of(?x,?a) :- neg timepoint(endof(?x)).
timepoint(endof(?x))   :- occurrence_of(?x, ?a).

neg object(?x) :- neg timepoint(beginof(?x)).
timepoint(beginof(?x)) :- object(?x).

neg object(?x) :- neg timepoint(endof(?x)).
timepoint(endof(?x))   :- object(?x).
```

Axiom 15 *The begin point of every activity occurrence or object is before or equal to its end point.*

KIF

```
(forall (?a ?x)
        (implies (or    (activity_occurrence ?x)
                        (object ?x))
                 (beforeEq (beginof ?x) (endof ?x))))
```

SWSL-FOL

```
forall ?a, ?x
  ( activity_occurrence(?x) or object(?x) ==>
    beforeEq(beginof(?x), endof(?x)) ).
```

SWSL-Rules

```
neg activity_occurrence(?x) :- neg beforeEq(beginof(?x), endof(?x)).
beforeEq(beginof(?x), endof(?x)) :- activity_occurrence(?x).

neg object(?x) :- neg beforeEq(beginof(?x), endof(?x)).
beforeEq(beginof(?x), endof(?x)) :- object(?x).
```

Axiom 16 *The participates_in relation only holds between objects, activities, and timepoints, respectively.*

KIF

```
(forall (?x ?occ ?t)
        (implies  (participates_in ?x ?occ ?t)
                  (and  (object ?x)
                        (activity_occurrence ?occ)
                        (timepoint ?t))))
```

SWSL-FOL

```
forall ?x, ?occ, ?t
  ( participates_in(?x, ?occ, ?t) ==>
    object(?x) and activity_occurrence(?occ)and timepoint(?t) ).
```

SWSL-Rules

```
object(?x) :- participates_in(?x ?_Occ ?t).
neg participates_in(?x ?_Occ ?t) :- neg object(?x).

activity_occurrence(?occ) :- participates_in(?x ?occ ?t).
neg participates_in(?x ?occ ?t) :- neg activity_occurrence(?occ).

timepoint(?t) :- participates_in(?x ?_Occ ?t).
neg participates_in(?x ?_Occ ?t) :- neg timepoint(?t).
```

Axiom 17 *An object can participate in an activity only at those timepoints at which both the object exists and the activity is occurring.*

KIF

```
(forall (?x ?occ ?t)
        (implies  (participates_in ?x ?a ?t)
                  (and  (exists_at ?x ?t)
                        (is_occurring_at ?occ ?t))))
```

SWSL-FOL

```
forall ?x, ?occ, ?t
  ( participates_in(?x, ?a, ?t) ==>
    exists_at(?x, ?t) and is_occurring_at(?occ, ?t) ).
```

SWSL-Rules

```
exists_at(?x, ?t) :- participates_in(?x, ?a, ?t).
neg participates_in(?x, ?a, ?t) :- neg exists_at(?x, ?t).

is_occurring_at(?a, ?t) :- participates_in(?x, ?a, ?t).
neg participates_in(?x, ?a, ?t) :- neg is_occurring_at(?a, ?t).
```

Supporting Definitions

Definition 1 *Timepoint ?t2 is between timepoints ?t1 and ?t3 if and only if ?t1 is before ?t2 and ?t2 is before ?t3.*

KIF

```
(forall (?t1 ?t2 ?t) (iff (between ?t1 ?t2 ?t3)
  (and (before ?t1 ?t2) (before ?t2 ?t3))))
```

SWSL-FOL

```
forall ?t1, ?t2, ?t
  ( between(?t1, ?t2, ?t3) iff
    before( ?t1, ?t2) and before(?t2, ?t3) ).
```

SWSL-Rules

```
neg between(?Beg, ?Mid, ?End) :- neg before(?Beg, ?Mid).
before(?Beg, ?Mid) :- between(?Beg, ?Mid, ?End).

neg between(?Beg, ?Mid, ?End) :- neg before(?Mid, ?End).
before(?Mid, ?End) :- between(?Beg, ?Mid, ?End).

neg before(?Beg, ?Mid)  :- before(?Mid, ?End) and neg between(?Beg, ?Mid, ?End).
neg before(?Mid, ?End) :- before(?Beg, ?Mid) and neg between(?Beg, ?Mid, ?End).
between(?Beg, ?Mid, ?End) :- before(?Beg, ?Mid) and before(?Mid, ?End).
```

Definition 2 *Timepoint ?t1 is beforeEq timepoint ?t2 if and only if ?t1 is before or equal to ?t2.*

KIF

```
(forall (?t1 ?t2) (iff (beforeEq ?t1 ?t2)
  (and (timepoint ?t1) (timepoint ?t2)
       (or (before ?t1 ?t2) (= ?t1 ?t2)))))
```

SWSL-FOL

```
forall ?t1, ?t2
  ( beforeEq(?t1, ?t2) iff
    ( timepoint(?t1) and (timepoint ?t2) and
      ( before(?t1, ?t2) or ?t1 :=: ?t2 ) ) ).
```

SWSL-Rules

```
neg timepoint(?t1) :- timepoint(?t2) and before(?t1,?t2) and neg beforeEq(?t1,?t2).
neg timepoint(?t2) :- timepoint(?t1) and before(?t1,?t2) and neg beforeEq(?t1,?t2).
neg before(?t1,?t2) :- timepoint(?t1) and timepoint(?t2) and neg beforeEq(?t1,?t2).
beforeEq(?t1,?t2) :- timepoint(?t1) and timepoint(?t2) and before(?t1,?t2).

neg timepoint(?t1) :- timepoint(?t2) and ?t1 :=: ?t2 and neg beforeEq(?t1,?t2).
neg timepoint(?t2) :- timepoint(?t1)and ?t1 :=: ?t2 and neg beforeEq(?t1,?t2).
neg ?t1 :=: ?t2 :- timepoint(?t1) and timepoint(?t2) and neg beforeEq(?t1,?t2).
beforeEq(?t1,?t2) :- timepoint(?t1) and timepoint(?t2) and ?t1 :=: ?t2.

neg beforeEq(?t1,?t2) :- neg timepoint(?t1).
timepoint(?t1) :- beforeEq(?t1,?t2).

neg beforeEq(?t1,?t2) :- neg timepoint(?t2).
```

```
timepoint(?t2) :- beforeEq(?t1,?t2).

neg beforeEq(?t1,?t2) :- neg before(?t1,?t2) and neg ?t1 :=: ?t2.
before(?t1,?t2) :- beforeEq(?t1,?t2) and neg ?t1 :=: ?t2.
?t1 :=: ?t2 :- beforeEq(?t1,?t2) and neg before(?t1,?t2).
```

Definition 3 *Timepoint ?t2 is betweenEq timepoints ?t1 and ?t3 if and only if ?t1 is before or equal to ?t2, and ?t2 is before or equal to ?t3.*

KIF

```
(forall (?t1 ?t2 ?t3) (iff (betweenEq ?t1 ?t2 ?t3)
  (and (beforeEq ?t1 ?t2)
       (beforeEq ?t2 ?t3))))
```

SWSL-FOL

```
forall ?t1, ?t2, ?t3
  ( betweenEq(?t1, ?t2, ?t3) iff
    beforeEq(?t1, ?t2) and beforeEq(?t2, ?t3) ).
```

SWSL-Rules

```
betweenEq(?t1, ?t2, ?t3) :-  beforeEq(?t1, ?t2) and beforeEq(?t2, ?t3).
neg beforeEq(?t1, ?t2) :- neg betweenEq(?t1, ?t2, ?t3) and beforeEq(?t2, ?t3).
neg beforeEq(?t2, ?t3) :-neg betweenEq(?t1, ?t2, ?t3) and beforeEq(?t1, ?t2).

beforeEq(?t1, ?t2) :- betweenEq(?t1, ?t2, ?t3).
neg betweenEq(?t1, ?t2, ?t3) :- neg beforeEq(?t1, ?t2).

beforeEq(?t2, ?t3) :- betweenEq(?t1, ?t2, ?t3).
neg betweenEq(?t1, ?t2, ?t3) :- neg beforeEq(?t2, ?t3).
```

Definition 4 *An object exists at a timepoint ?t if and only if ?t is betweenEq its begin and end points.*

KIF

```
(forall (?x ?t) (iff (exists_at ?x ?t)
  (and (object ?x)
       (betweenEq (beginof ?x) ?t (endof ?x)))))
```

SWSL-FOL

```
forall ?x, ?t)
  ( exists_at(?x, ?t) iff
    object(?x) and betweenEq(beginof(?x), ?t, endof(?x)) ).
```

SWSL-Rules

```
neg exists_at(?x,?t) :- neg object(?x).
object(?x) :- exists_at(?x,?t).

neg exists_at(?x,?t) :- neg betweenEq(beginof(?x),?t,endof(?x)).
betweenEq(beginof(?x),?t,endof(?x)) :- exists_at(?x,?t).
```

```
exists_at(?x,?t) :- object(?x) and betweenEq(beginof(?x),?t,endof(?x)).
neg object(?x) :- neg exists_at(?x,?t) and betweenEq(beginof(?x),?t,endof(?x)).
neg betweenEq(beginof(?x),?t,endof(?x)) :- neg exists_at(?x,?t) and object(?x).
```

Definition 5 *An activity is occurring at a timepoint ?t if and only if ?t is betweenEq the activity's begin and end points.*

KIF

```
(forall (?occ ?t) (iff (is_occurring_at ?occ ?t)
        (and    (activity_occurrence ?occ)
                (betweenEq (beginof ?occ) ?t (endof ?occ))))))
```

SWSL-FOL

```
forall ?occ, ?t
 ( is_occurring_at(?occ, ?t) iff
   activity_occurrence(?occ) and betweenEq(beginof(?occ), ?t, endof(?occ)) ).
```

SWSL-Rules

```
is_occurring_at(?occ, ?t) :- activity_occurrence(?occ) and betweenEq(beginof(?occ), ?t, endof(?
occ)).
neg activity_occurrence(?occ) :- neg is_occurring_at(?occ, ?t) and betweenEq(beginof(?occ), ?t,
endof(?occ)).
neg betweenEq(beginof(?occ), ?t, endof(?occ)) :- neg is_occurring_at(?occ, ?t) and
activity_occurrence(?occ).

activity_occurrence(?occ) :- is_occurring_at(?occ, ?t).
neg is_occurring_at(?occ, ?t) :- neg activity_occurrence(?occ).

betweenEq(beginof(?occ), ?t, endof(?occ)) :- is_occurring_at(?occ, ?t).
neg is_occurring_at(?occ, ?t) :- neg betweenEq(beginof(?occ), ?t, endof(?occ)).
```

A.2 Theory of Subactivities

Extension Name: subactivity.th
Theories Required by this Extension: psl_core.th
Definitional Extensions Required by this Extension: None

Primitive Lexicon

Relations:

KIF

- `(subactivity ?a1 ?a2)`

SWSL-FOL and SWSL-Rules

- `subactivity(?a1, ?a2)`

Defined Lexicon

Relations:

KIF

- `(primitive ?a1 ?a2)`

SWSL-FOL and SWSL-Rules

- `primitive(?a1, ?a2)`

Axioms

Axiom 1 *subactivity is a relation over activities*

KIF

```
(forall (?a1 ?a2)
        (implies  (subactivity ?a1 ?a2)
                  (and  (activity ?a1)
                        (activity ?a2))))
```

SWSL-FOL

```
forall ?a1, ?a2
  ( subactivity(?a1 ,?a2) ==>
    ( activity(?a1) and activity(?a2) ) ).
```

SWSL-Rules

```
activity(?a1) :- subactivity(?a1, ?a2).
neg subactivity(?a1, ?a2) :- neg activity(?a1).

activity(?a2) :- subactivity(?a1, ?a2).
neg subactivity(?a1, ?a2) :- neg activity(?a2).
```

Axiom 2 *The subactivity relation is reflexive.*

KIF

```
(forall (?a)
        (implies  (activity ?a)
                  (subactivity ?a ?a)))
```

SWSL-FOL

```
forall ?a
  ( activity(?a) ==>
    subactivity(?a, ?a) ).
```

SWSL-Rules

```
subactivity(?a, ?a) :- activity(?a).
neg activity(?a) :- neg subactivity(?a, ?a).
```

Axiom 3 *The subactivity relation is antisymmetric.*

KIF

```
(forall (?a1 ?a2)
        (implies  (and  (subactivity ?a1 ?a2)
                        (subactivity ?a2 ?a1))
                  (= ?a1 ?a2)))
```

SWSL-FOL

```
forall ?a1 ?a2
  ( ( subactivity(?a1 ?a2) and
      subactivity(?a2 ?a1) ) ==>
    ?a1 :=: ?a2 ).
```

SWSL-Rules

```
?a1 :=: ?a2 :- subactivity(?a1, ?a2) and subactivity ?a2 ?a1).
neg subactivity(?a1, ?a2) :- subactivity ?a2 ?a1) and neg ?a1 :=: ?a2.
neg subactivity ?a2 ?a1) :- subactivity(?a1, ?a2) and neg ?a1 :=: ?a2.
```

Axiom 4 *The subactivity relation is transitive.*

KIF

```
(forall (?a1 ?a2 ?a3)
        (implies  (and  (subactivity ?a1 ?a2)
                        (subactivity ?a2 ?a3))
                  (subactivity ?a1 ?a3)))
```

SWSL-FOL

```
forall ?a1, ?a2, ?a3
  ( ( subactivity(?a1, ?a2) and
      subactivity(?a2, ?a3) ) ==>
    subactivity(?a1, ?a3) ).
```

SWSL-Rules

```
neg subactivity(?a1, ?a2) :- subactivity(?a2, ?a3) and neg subactivity(?a1, ?a3).
neg subactivity(?a2, ?a3) :- subactivity(?a1, ?a2) and neg subactivity(?a1, ?a3).
subactivity(?a1, ?a3) :- subactivity(?a1, ?a2) and subactivity(?a2, ?a3).
```

Axiom 5 *The subactivity relation is a discrete ordering, so every activity has an upwards successor in the ordering.*

KIF

```
(forall (?a1 ?a2)
        (implies  (subactivity ?a1 ?a2)
                  (exists (?a3)
                        (and    (subactivity ?a1 ?a3)
                                (subactivity ?a3 ?a2)
                                (forall (?a4)
                                        (implies  (and  (subactivity ?a1 ?a4)
                                                        (subactivity ?a4 ?a3)
                                                  (or   (= ?a4 ?a1)
                                                        (= ?a4 ?a3))))))))
```

SWSL-FOL

```
forall ?a1, ?a2
 ( subactivity(?a1, ?a2) ==>
   exists (?a3)
     ( subactivity(?a1, ?a3) and
       subactivity(?a3, ?a2) and
       forall ?a4
         ( ( subactivity(?a1, ?a4) and subactivity(?a4, ?a3) ) ==>
         ( ?a4 :=: ?a1 or ?a4 :=: ?a3 ) ) ) ).
```

SWSL-Rules

```
neg subactivity(?a1,?a2) :- neg subactivity(?a1,_#1(?a1,?a2)).
subactivity(?a1,_#1(?a1,?a2)) :- subactivity(?a1,?a2).

neg subactivity(?a1,?a2) :- neg subactivity(_#1(?a1,?a2),?a2).
subactivity(_#1(?a1,?a2),?a2) :- subactivity(?a1,?a2).

neg subactivity(?a1,?a2) :- subactivity(?a1,?a4) and subactivity(?a4,_#1(?a1,?a2)) and neg ?a4 :
=: ?a1 and neg ?a4 :=: _#1(?a1,?a2).
neg subactivity(?a1,?a4) :- subactivity(?a1,?a2) and subactivity(?a4,_#1(?a1,?a2)) and neg ?a4 :
=: ?a1 and neg ?a4 :=: _#1(?a1,?a2).
neg subactivity(?a4,_#1(?a1,?a2)) :- subactivity(?a1,?a2) and
subactivity(?a1,?a4) and neg ?a4 :=: ?a1 and neg ?a4 :=: _#1(?a1,?a2).
?a4 :=: ?a1 :- subactivity(?a1,?a2) and subactivity(?a1,?a4) and subactivity(?a4,_#1(?a1,?a2))
and neg ?a4 :=: _#1(?a1,?a2).
?a4 :=: _#1(?a1,?a2) :- subactivity(?a1,?a2) and subactivity(?a1,?a4) and subactivity(?a4,_#1(?
a1,?a2)) and neg ?a4 :=: ?a1.
```

Axiom 6 *The subactivity relation is a discrete ordering, so every activity has a downwards successor in the ordering.*

KIF

```
(forall (?a1 ?a2)
        (implies  (subactivity ?a1 ?a2)
                  (exists (?a3)
                        (and    (subactivity ?a1 ?a3)
                                (subactivity ?a3 ?a2)
                                (forall (?a4)
                                        (implies  (and  (subactivity ?a3 ?a4)
                                                        (subactivity ?a4 ?a2)
                                                  (or   (= ?a4 ?a2)
                                                          (= ?a4 ?a3))))))))
```

SWSL-FOL

```
forall ?a1, ?a2
  ( subactivity(?a1, ?a2) ==>
    exists ?a3
      ( subactivity(?a1, ?a3) and
        subactivity(?a3, ?a2) and
        forall ?a4
          ( ( subactivity(?a3, ?a4) and subactivity(?a4, ?a2) ) ==>
            ( ?a4 :=: ?a2 or ?a4 :=: ?a3 ) ) ) ).
```

SWSL-Rules

```
neg subactivity(?a1,?a2) :- neg subactivity(?a1,_#1(?a1,?a2)).
subactivity(?a1,_#1(?a1,?a2)) :- subactivity(?a1,?a2).

neg subactivity(?a1,?a2) :- neg subactivity(_#1(?a1,?a2),?a2).
subactivity(_#1(?a1,?a2),?a2) :- subactivity(?a1,?a2).

neg subactivity(?a1,?a2) :- subactivity(_#1(?a1,?a2),?a4) and subactivity(?a4,?a2) and neg ?a4 :
=: ?a2 and neg ?a4 :=: _#1(?a1,?a2).
neg subactivity(_#1(?a1,?a2),?a4) :- subactivity(?a1,?a2) and subactivity(?a4,?a2) and neg ?a4 :
=: ?a2 and neg ?a4 :=: _#1(?a1,?a2).
neg subactivity(?a4,?a2) :- subactivity(?a1,?a2) and subactivity(_#1(?a1,?a2),?a4) and neg ?a4 :
=: ?a2 and neg ?a4 :=: _#1(?a1,?a2).
?a4 :=: ?a2 :- subactivity(?a1,?a2) and subactivity(_#1(?a1,?a2),?a4) and subactivity(?a4,?a2)
and neg ?a4 :=: _#1(?a1,?a2).
?a4 :=: _#1(?a1,?a2) :- subactivity(?a1,?a2) and subactivity(_#1(?a1,?a2),?a4) and subactivity(?
a4,?a2) and neg ?a4 :=: ?a2.
```

Definitions

Definition 1 *An activity is primitive iff it has no proper subactivities.*

KIF

```
(forall (?a ?a1) (iff (primitive ?a)
(implies  (subactivity ?a1 ?a)
          (= ?a1 ?a)))))
```

SWSL-FOL

```
forall ?a, ?a1
  ( primitive(?a) <==>
    ( subactivity ?a1 ?a) ==> ?a1 :=: ?a ).
```

SWSL-Rules

```
primitive(?a) :-
  subactivity(?a1, ?a) ==> (?a1 :=: ?a).
subactivity(?a1, ?a) ==> (?a1 :=: ?a) :-
```

```
primitive(?a).
```

A.3 Theory of Occurrence Trees

Extension Name: occtree.th
Theories Required by this Extension: psl_core.th
Definitional Extensions Required by this Extension: None

Primitive Lexicon

Relations:

KIF

- `(earlier ?s1 ?s2)`
- `(initial ?s1)`
- `(legal ?s)`

SWSL-FOL and SWSL-Rules

- `earlier(?s1, ?s2)`
- `initial(?s1)`
- `legal(?s)`

Functions:

KIF

- `(successor ?a ?s)`

SWSL-FOL and SWSL-Rules

- `successor(?a, ?s)`

Defined Lexicon

Relations:

KIF

- `(precedes ?s1 ?s2)`
- `(earlierEq ?s1 ?s2)`
- `(poss ?a ?s)`

SWSL-FOL and SWSL-Rules

- `precedes(?s1, ?s2)`
- `earlierEq(?s1, ?s2)`
- `poss(?a, ?s)`

Axioms

Axiom 1 *The earlier relation is restricted to activity occurrences.*

KIF

```
(forall (?s1 ?s2)
        (implies  (earlier ?s1 ?s2)
                  (and  (activity_occurrence ?s1)
                        (activity_occurrence ?s2))))
```

SWSL-FOL

```
forall ?s1, ?s2
  ( earlier(?s1, ?s2) ==>
    activity_occurrence(?s1) and activity_occurrence(?s2) ).
```

SWSL-Rules

```
neg earlier(?s1, ?s2) :- neg activity_occurrence(?s1).
activity_occurrence(?s1) :- earlier(?s1, ?s2).

neg earlier(?s1, ?s2) :- neg activity_occurrence(?s2).
activity_occurrence(?s2) :- earlier(?s1, ?s2).
```

Axiom 2 *The ordering relation over occurrences is irreflexive.*

KIF

```
(forall (?s1 ?s2)
        (implies  (earlier ?s1 ?s2)
                  (not (earlier ?s2 ?s1))))
```

SWSL-FOL

```
forall ?s1, ?s2
  ( earlier(?s1, ?s2) ==>
    neg earlier(?s2, ?s1) ).
```

SWSL-Rules

```
neg earlier(?s2, ?s1) :- earlier(?s1, ?s2).
neg earlier(?s1, ?s2) :- earlier(?s2, ?s1).
```

Axiom 3 *The ordering relation over occurrences is transitive.*

KIF

```
(forall (?s1 ?s2 ?s3)
        (implies  (and  (earlier ?s1 ?s2)
                        (earlier ?s2 ?s3))
                  (earlier ?s1 ?s3)))
```

SWSL-FOL

```
forall ?s1, ?s2, ?s3
  ( earlier(?s1, ?s2) and earlier(?s2, ?s3) ==>
    earlier(?s1, ?s3) ).
```

SWSL-Rules

```
neg earlier(?s1, ?s2) :- earlier(?s2, ?s3) and neg earlier(?s1, ?s3).
neg earlier(?s2, ?s3) :- earlier(?s1, ?s2) and neg earlier(?s1, ?s3).
earlier(?s1, ?s3) :- earlier(?s1, ?s2) and earlier(?s2, ?s3).
```

Axiom 4 *A branch in the occurrence tree is totally ordered.*

KIF

```
(forall (?s1 ?s2 ?s3)
        (implies  (and  (earlier ?s1 ?s2)
                        (earlier ?s3 ?s2))
                  (or   (earlier ?s1 ?s3)
                        (earlier ?s3 ?s1)
                        (= ?s3 ?s1))))
```

SWSL-FOL

```
forall ?s1, ?s2, ?s3
  ( earlier(?s1, ?s2) and earlier(?s3, ?s2) ==>
    earlier(?s1, ?s3) or earlier(?s3, ?s1) or ?s3 :=: ?s ).
```

SWSL-Rules

```
neg earlier(?s1, ?s2) :-
    earlier(?s3, ?s2) and neg earlier(?s1, ?s3) and neg earlier(?s3, ?s1) and neg ?s3 :=: ?s1.
neg earlier(?s3, ?s2) :-
    earlier(?s1, ?s2) and neg earlier(?s1, ?s3) and neg earlier(?s3, ?s1) and neg ?s3 :=: ?s1.
earlier(?s1, ?s3) :-
    earlier(?s1, ?s2) and earlier(?s3, ?s2) and neg earlier(?s3, ?s1) and neg ?s3 :=: ?s1.
earlier(?s3, ?s1) :-
    earlier(?s1, ?s2) and earlier(?s3, ?s2) and neg earlier(?s1, ?s3) and neg ?s3 :=: ?s1.
?s3 :=: ?s1 :-
    earlier(?s1, ?s2) and earlier(?s3, ?s2) and neg earlier(?s1, ?s3) and neg earlier(?s3, ?s1).
```

Axiom 5 *No occurrence is earlier than an initial occurrence.*

KIF

```
(forall (?s1)
        (iff    (initial ?s1)
                (not (exists (?s2)
                        (earlier ?s2 ?s1)))))
```

SWSL-FOL

```
forall ?s1
  ( initial(?s1) <==> neg exists ?s2 ( earlier(?s2,?s1) ) ).
```

SWSL-Rules

```
initial(?s1) :- neg earlier(_#1(?s1),?s1).
earlier(_#1(?s1),?s1) :- neg initial(?s1).

neg initial(?s1) :- earlier(?s2,?s1).
neg earlier(?s2,?s1) :- initial(?s1).
```

Axiom 6 *Every branch of the occurrence tree has an initial occurrence.*

KIF

```
(forall (?s1 ?s2)
        (implies  (earlier ?s1 ?s2)
                  (exists (?sp)
                        (and    (initial ?sp)
                                (earlierEq ?sp ?s1)))))
```

SWSL-FOL

```
forall ?s1, ?s2
  ( earlier(?s1, ?s2) ==>
    exists ?sp
      ( initial(?sp) and earlierEq(?sp, ?s1) ) ).
```

SWSL-Rules

```
neg earlier(?s1, ?s2) :- neg initial(_#1(?s1, ?s2)).
initial(_#1(?s1, ?s2)) :- earlier(?s1, ?s2).

neg earlier(?s1, ?s2) :- neg earlierEq(_#1(?s1, ?s2), ?s1).
earlierEq(_#1(?s1, ?s2), ?s1) :- earlier(?s1, ?s2).
```

Axiom 7 *There is an initial occurrence of each activity.*

KIF

```
(forall (?a)
        (implies  (activity ?a)
                  (exists (?s)
                        (and    (occurrence_of ?s ?a)
                                (initial ?s)))))
```

SWSL-FOL

```
forall ?a
  ( activity(?a) ==>
    exists ?s
      ( occurrence_of(?s, ?a) and initial(?s) ) ).
```

SWSL-Rules

```
neg activity(?a) :- neg occurrence_of(_#1(?a), ?a).
occurrence_of(_#1(?a), ?a) :- activity(?a).

neg activity(?a) :- neg initial(_#1(?a)).
initial(_#1(?a)) :- activity(?a).
```

Axiom 8 *No two initial activity occurrences in the occurrence tree are occurrences of the same activity.*

KIF

```
(forall (?s1 ?s2 ?a)
        (implies  (and  (initial ?s1)
                        (initial ?s2)
                        (occurrence_of ?s1 ?a)
                        (occurrence_of ?s2 ?a))
                  (= ?s1 ?s2)))
```

SWSL-FOL

```
forall ?s1, ?s2, ?a
  ( initial(?s1) and initial(?s2) and
    occurrence_of(?s1, ?a) and occurrence_of(?s2, ?a) ==>
    ?s1 :=: ?s2 ).
```

SWSL-Rules

```
neg initial(?s1) :- initial(?s2) and occurrence_of(?s1, ?a) and occurrence_of(?s2, ?a) and neg ?
s1 :=: ?s2.
neg initial(?s2) :- initial(?s1) and occurrence_of(?s1, ?a) and occurrence_of(?s2, ?a) and neg ?
s1 :=: ?s2.
neg occurrence_of(?s1, ?a) :- initial(?s1) and initial(?s2) and occurrence_of(?s2, ?a) and neg ?
s1 :=: ?s2.
neg occurrence_of(?s2, ?a) :- initial(?s1) and initial(?s2) and occurrence_of(?s1, ?a) and neg ?
s1 :=: ?s2.
?s1 :=: ?s2 :- initial(?s1) and initial(?s2) and occurrence_of(?s1, ?a) and occurrence_of(?s2, ?
a).
```

Axiom 9 *The successor of an activity occurrence is an occurrence of the activity.*

KIF

```
(forall (?a ?s)
        (implies  (and  (activity ?a)
                        (activity_occurrence ?s))
                  (occurrence_of (successor ?a ?s) ?a)))
```

SWSL-FOL

```
forall ?a, ?s
  ( activity(?a) and activity_occurrence(?s) ==>
    occurrence_of(successor(?a, ?s), ?a) ).
```

SWSL-Rules

```
neg activity(?a) :- activity_occurrence(?s) and neg occurrence_of(successor(?a, ?s), ?a).
neg activity_occurrence(?s) :- activity(?a) and neg occurrence_of(successor(?a, ?s), ?a).
occurrence_of(successor(?a, ?s), ?a) :- activity(?a) and activity_occurrence(?s).
```

Axiom 10 *Every non-initial activity occurrence is the successor of another activity occurrence.*

KIF

```
(forall (?s)
        (implies  (not (initial ?s))
                  (exists (?a ?sp)
                        (= ?s (successor ?a ?sp)))))
```

SWSL-FOL

```
forall ?s
  ( neg initial(?s) ==>
    exists ?a ?sp
      ( ?s :=: successor(?a, ?sp) ) ).
```

SWSL-Rules

```
?s :=: successor(_#1(?s), _#2(?s)) :- neg initial(?s).
initial(?s) :- neg ?s :=: successor(_#1(?s), _#2(?s)).
```

Axiom 11 *An occurrence ?s1 is earlier than the successor occurrence of ?s2 if and only if the occurrence ?s2 is later than ?s1.*

KIF

```
(forall (?a ?s1 ?s2)
        (iff    (earlier ?s1 (successor ?a ?s2))
                (earlierEq ?s1 ?s2)))
```

SWSL-FOL

```
forall ?a, ?s1, ?s2
  ( earlier(?s1, successor(?a, ?s2)) iff earlierEq(?s1, ?s2) ).
```

SWSL-Rules

```
earlier(?s1, successor(?a, ?s2)) :- earlierEq(?s1, ?s2).
neg earlierEq(?s1, ?s2) :- neg earlier(?s1, successor(?a, ?s2)).

earlierEq(?s1, ?s2) :- earlier(?s1, successor(?a, ?s2)).
```

```
neg earlier(?s1, successor(?a, ?s2)) :- neg earlierEq(?s1, ?s2).
```

Axiom 12 *The legal relation restricts activity occurrences.*

KIF

```
(forall (?s)
        (implies  (legal ?s)
                  (activity_occurrence ?s)))
```

SWSL-FOL

```
forall ?s
  ( legal(?s) ==> activity_occurrence(?s) ).
```

SWSL-Rules

```
activity_occurrence(?s) :- legal(?s).
neg legal(?s) :- neg activity_occurrence(?s).
```

Axiom 13 *If an activity occurrence is legal, all earlier activity occurrences in the occurrence tree are also legal.*

KIF

```
(forall (?s1 ?s2)
        (implies  (and  (legal ?s1)
                        (earlier ?s2 ?s1))
                  (legal ?s2)))
```

SWSL-FOL

```
forall ?s1, ?s2
  ( legal(?s1) and earlier(?s2, ?s1) ==> legal(?s2) ).
```

SWSL-Rules

```
neg legal(?s1) :- earlier(?s2, ?s1) and neg legal(?s2).
neg earlier(?s2, ?s1) :- legal(?s1) and neg legal(?s2).
legal(?s2) :- legal(?s1) and earlier(?s2, ?s1).
```

Axiom 14 *The endof an activity occurrence is before the beginof the successor of the activity occurrence.*

KIF

```
(forall (?s1 ?s2)
        (implies  (earlier ?s1 ?s2)
                  (before (endof ?s1) (beginof ?s2))))

TBD - why isn't successor mentioned above?
```

SWSL-FOL

```
forall ?s1, ?s2
  ( earlier(?s1, ?s2) ==>
    before(endof(?s1), beginof(?s2)) ).
```

SWSL-Rules

```
before(endof(?s1), beginof(?s2)) :- earlier(?s1, ?s2).
neg earlier(?s1, ?s2) :- neg before(endof(?s1), beginof(?s2)).
```

Definitions

Definition 1 *An activity occurrence ?s1 precedes another activity occurrence ?s2 if and only if ?s1 is earlier than ?s2 in the occurrence tree and ?s2 is legal.*

KIF

```
(forall (?s1 ?s2) (iff (precedes ?s1 ?s2)
(and    (earlier ?s1 ?s2)
        (legal ?s2))))
```

SWSL-FOL

```
forall ?s1, ?s2
  ( precedes(?s1, ?s2) iff
    earlier(?s1, ?s2) and legal(?s2) ).
```

SWSL-Rules

```
neg precedes(?s1, ?s2) :- neg earlier(?s1, ?s2) .
earlier(?s1, ?s2)  :- precedes(?s1, ?s2).

neg precedes(?s1, ?s2) :- neg legal(?s2).
legal(?s2) :- precedes(?s1, ?s2).

neg earlier(?s1, ?s2) :- legal(?s2) and neg precedes(?s1, ?s2).
neg legal(?s2) :- earlier(?s1, ?s2) and neg precedes(?s1, ?s2).
precedes(?s1, ?s2) :- earlier(?s1, ?s2) and legal(?s2).
```

Definition 2 *An activity occurrence ?s1 is EarlierEq than an activity occurrence ?s2 if and only if it is either earlier than ?s2 or it is equal to ?s2.*

KIF

```
(forall (?s1 ?s2) (iff (earlierEq ?s1 ?s2)
(or     (earlier ?s1 ?s2)
        (= ?s1 ?s2))))
```

SWSL-FOL

```
forall ?s2, ?s3
  ( earlierEq(?s1, ?s2) iff
    earlier(?s1, ?s2) or ?s1 :=: ?s2 ).
```

SWSL-Rules

```
earlierEq(?s1,?s2) :- earlier(?s1, ?s2).
neg earlier(?s1, ?s2) :- neg earlierEq(?s1,?s2).

earlierEq(?s1,?s2) :- ?s1 :=: ?s2.
neg ?s1 :=: ?s2 :- neg earlierEq(?s1,?s2).

neg earlierEq(?s1,?s2) :- neg earlier(?s1, ?s2) and neg ?s1 :=: ?s2.
earlier(?s1, ?s2) :- neg earlierEq(?s1,?s2) and neg ?s1 :=: ?s2.
?s1 :=: ?s2 :- earlierEq(?s1,?s2) and neg earlier(?s1, ?s2).
```

Definition 3 *An activity is poss at some occurrence if and only if the successor occurrence of the activity is legal.*

KIF

```
(forall (?a ?s) (iff (poss ?a ?s)
(legal (successor ?a ?s))))
```

SWSL-FOL

```
forall ?a, ?s
  ( poss(?a, ?s) iff legal(successor(?a, ?s)) ).
```

SWSL-Rules

```
poss(?a, ?s) :- legal(successor(?a, ?s)).
neg legal(successor(?a, ?s)) :- neg poss(?a, ?s).

legal(successor(?a, ?s)) :- poss(?a, ?s).
neg poss(?a, ?s) :- neg legal(successor(?a, ?s)).
```

A.4 Theory of Discrete States

Extension Name: disc_state.th
Theories Required by this Extension: occtree.th, psl_core.th
Definitional Extensions Required by this Extension: None

Primitive Lexicon

Relations:

KIF

- `(state ?f)`
- `(holds ?f ?occ)`
- `(prior ?f ?occ)`

SWSL-FOL and SWSL-Rules

- `state(?f)`
- `holds(?f, ?occ)`
- `prior(?f, ?occ)`

Defined Lexicon

(none)

Axioms

Axiom 1 *States are objects.*

KIF

```
(forall (?f)
        (implies  (state ?f)
                  (object ?f)))
```

SWSL-FOL

```
forall ?f
  ( state(?f) ==> object(?f) ).
```

SWSL-Rules

```
object(?f) :- state(?f).
neg state(?f) :- neg object(?f).
```

Axiom 2 *The holds relation is only between states and activity occurrences. Intuitively, it means that the fluent (property of the world) is true after the activity occurrence ?occ.*

KIF

```
(forall (?f ?occ)
        (implies  (holds ?f ?occ)
                  (and  (state ?f)
                        (activity_occurrence ?occ))))
```

SWSL-FOL

```
forall ?f ?occ
  holds(?f ?occ) ==>
  ( state(?f) and activity_occurrence(?occ) ).
```

SWSL-Rules

```
state(?f) :- holds(?f, ?occ).
neg holds(?f, ?occ) :- neg state(?f).
```

```
activity_occurrence(?occ) :- holds(?f, ?occ).
neg holds(?f, ?occ) :- neg activity_occurrence(?occ).
```

Axiom 3 *The prior relation is only between states and activity occurrences. Intuitively, it means that the fluent (property of the world) is true before the activity occurrence ?occ.*

KIF

```
(forall (?f ?occ)
        (implies  (prior ?f ?occ)
                  (and  (state ?f)
                        (activity_occurrence ?occ))))
```

SWSL-FOL

```
forall ?f, ?occ
  ( prior(?f, ?occ) ==>
    ( state(?f) and activity_occurrence(?occ) ) ).
```

SWSL-Rules

```
state(?f) :- prior(?f, ?occ).
neg prior(?f, ?occ) :- neg state(?f).

activity_occurrence(?occ) :- prior(?f, ?occ).
neg prior(?f, ?occ) :- neg activity_occurrence(?occ).
```

Axiom 4 *All initial occurrences agree on the states that hold prior to them.*

KIF

```
(forall (?occ1 ?occ2 ?f)
        (implies  (and  (initial ?occ1)
                        (initial ?occ2))
                  (iff  (prior ?f ?occ1)
                        (prior ?f ?occ2))))
```

SWSL-FOL

```
forall ?occ1 ?occ2 ?f
  ( ( initial(?occ1) and initial(?occ2) ) ==>
    ( prior(?f, ?occ1) <==> prior(?f, ?occ2) ) ).
```

SWSL-Rules

```
neg initial(?occ1) :- initial(?occ2) and neg prior(?f, ?occ1) and prior(?f, ?occ2).
neg initial(?occ2) :- initial(?occ1) and neg prior(?f, ?occ1) and prior(?f, ?occ2).
prior(?f, ?occ1) :- initial(?occ1) and initial(?occ2) and prior(?f, ?occ2).
neg prior(?f, ?occ2) :- initial(?occ1) and initial(?occ2) and neg prior(?f, ?occ1).

neg initial(?occ1) :- initial(?occ2) and prior(?f, ?occ1) and neg prior(?f, ?occ2).
neg initial(?occ2) :- initial(?occ1) and prior(?f, ?occ1) and neg prior(?f, ?occ2).
neg prior(?f, ?occ1) :- initial(?occ1) and initial(?occ2) and neg prior(?f, ?occ2).
prior(?f, ?occ2) :- initial(?occ1) and initial(?occ2) and prior(?f, ?occ1).
```

Axiom 5 *A state holds after an occurrence if and only if it holds prior to the successor occurrence.*

KIF

```
(forall (?a ?occ)
        (iff    (holds ?f ?occ)
                (prior ?f (successor ?a ?occ))))
```

SWSL-FOL

```
forall ?a ,?occ
  ( holds(?f ,?occ) <==>
    prior(?f, successor(?a ,?occ)) ).
```

SWSL-Rules

```
holds(?f, ?occ) :- prior(?f, successor(?a, ?occ)).
neg prior(?f, successor(?a, ?occ)) :- neg holds(?f, ?occ).

prior(?f, successor(?a, ?occ)) :- holds(?f, ?occ).
neg holds(?f, ?occ) :- neg prior(?f, successor(?a, ?occ)).
```

Axiom 6 *If a fluent holds after some activity occurrence, then there exists an earliest activity occurrence along the branch where the fluent holds.*

KIF

```
(forall (?occ1 ?f)
        (implies  (holds ?f ?occ1)
                  (exists (?occ2)
                        (and    (precedes ?occ2 ?occ1)
                                (holds ?f ?occ2)
                                (or     (initial ?occ2)
                                        (not (prior ?f ?occ2)))
                                (forall (?occ3)
                                        (implies  (and  (precedes ?occ2 ?occ3)
                                                        (precedes ?occ3 ?occ1))
                                                  (holds ?f ?occ3)))))))
```

SWSL-FOL

```
forall ?occ1, ?f
  ( holds(?f, ?occ1) ==>
    exists ?occ2
      ( precedes(?occ2, ?occ1) and
        holds(?f, ?occ2) and
        ( initial(?occ2) or neg prior(?f, ?occ2) ) and
        forall ?occ3
          ( ( precedes(?occ2, ?occ3) and precedes(?occ3, ?occ1) ) ==>
              holds(?f, ?occ3) ) ) ).
```

SWSL-Rules

```
neg holds(?f,?occ1) :- neg precedes(_#1(?occ1,?f),?occ1).
```

```
precedes(_#1(?occ1,?f),?occ1) :- holds(?f,?occ1).

neg holds(?f,?occ1) :- neg holds(?f,_#1(?occ1,?f)).
holds(?f,_#1(?occ1,?f)) :- holds(?f,?occ1).

neg holds(?f,?occ1) :- neg initial(_#1(?occ1,?f)) and prior(?f,_#1(?occ1,?f)).
initial(_#1(?occ1,?f)) :- holds(?f,?occ1) and prior(?f,_#1(?occ1,?f)).
neg prior(?f,_#1(?occ1,?f)) :- holds(?f,?occ1) and neg initial(_#1(?occ1,?f)).

neg holds(?f,?occ1) :- precedes(_#1(?occ1,?f),?occ3) and precedes(?occ3,?occ1) and neg holds(?f,?
occ3).
neg precedes(_#1(?occ1,?f),?occ3) :- holds(?f,?occ1) and precedes(?occ3,?occ1) and neg holds(?f,?
occ3).
neg precedes(?occ3,?occ1) :- holds(?f,?occ1) and precedes(_#1(?occ1,?f),?occ3) and neg holds(?f,?
occ3).
holds(?f,?occ3) :- holds(?f,?occ1) and precedes(_#1(?occ1,?f),?occ3) and precedes(?occ3,?occ1).
```

Axiom 7 *If a fluent does not hold after some activity occurrence, then there exists an earliest activity occurrence along the branch where the fluent does not hold.*

KIF

```
(forall (?occ1 ?f)
        (implies  (not (holds ?f ?occ1))
                  (exists (?occ2)
                        (and    (precedes ?occ2 ?occ1)
                                (not (holds ?f ?occ2))
                                (or     (initial ?occ2)
                                        (prior ?f ?occ2))
                                (not (exists (?occ3)
                                        (and    (precedes ?occ2 ?occ3)
                                                (precedes ?occ3 ?occ1))
                                                (holds ?f ?occ3))))))))
```

SWSL-FOL

```
forall ?occ1, ?f
  ( neg holds(?f, ?occ1) ==>
    exists ?occ2
      ( precedes(?occ2, ?occ1) and
        neg holds(?f, ?occ2) and
        ( initial(?occ2) or prior(?f, ?occ2) ) and
        neg exists ?occ3
          ( precedes(?occ2, ?occ3) and
            precedes(?occ3, ?occ1) and
            holds(?f, ?occ3) ) ) ).
```

SWSL-Rules

```
holds(?f,?occ1) :- neg precedes(_#1(?occ1,?f),?occ1).
precedes(_#1(?occ1,?f),?occ1) :- neg holds(?f,?occ1).

holds(?f,?occ1) :- holds(?f,_#1(?occ1,?f)).
neg holds(?f,_#1(?occ1,?f)) :- neg holds(?f,?occ1).

holds(?f,?occ1) :- neg initial(_#1(?occ1,?f)) and neg prior(?f,_#1(?occ1,?f)).
initial(_#1(?occ1,?f)) :- neg holds(?f,?occ1) and neg prior(?f,_#1(?occ1,?f)).
```

```
prior(?f,_#1(?occ1,?f)) :- neg holds(?f,?occ1) and neg initial(_#1(?occ1,?f)).

holds(?f,?occ1) :- precedes(_#1(?occ1,?f),?occ3) and precedes(?occ3,?occ1) and holds(?f,?occ3).
neg precedes(_#1(?occ1,?f),?occ3) :- neg holds(?f,?occ1) and precedes(?occ3,?occ1) and holds(?f,?
occ3).
neg precedes(?occ3,?occ1) :- neg holds(?f,?occ1) and precedes(_#1(?occ1,?f),?occ3) and holds(?f,?
occ3).
neg holds(?f,?occ3) :- neg holds(?f,?occ1) and precedes(_#1(?occ1,?f),?occ3) and precedes(?occ3,?
occ1).
```

A.5 Theory of Atomic Activities

Extension Name: atomic.th
Theories Required by this Extension: occtree.th,subactivity.th, psl_core.th
Definitional Extensions Required by this Extension: None

Primitive Lexicon

Relations:

KIF

- `(atomic ?a)`

SWSL-FOL and SWSL-Rules

- `atomic(?a)`

Functions:

KIF

- `(conc ?a1 ?a2)`

SWSL-FOL and SWSL-Rules

- `conc(?a1, ?a2)`

Defined Lexicon

(None)

Axioms

Axiom 1 *Primitive activities are atomic.*

KIF

```
(forall (?a)
        (implies  (primitive ?a)
                  (atomic ?a)))
```

SWSL-FOL

```
forall ?a
  ( primitive(?a) ==> atomic(?a) )
```

SWSL-Rules

```
atomic(?a) :- primitive(?a).
neg primitive(?a) :- neg atomic(?a).
```

Axiom 2 *The function conc is idempotent.*

KIF

```
(forall (?a)
        (= ?a (conc ?a ?a)))
```

SWSL-FOL

```
forall ?a
  ( ?a :=: conc(?a, ?a) ).
```

SWSL-Rules

```
?a :=: conc(?a, ?a) :- activity(?a).
```

Axiom 3 *The function conc is commutative.*

KIF

```
(forall (?a1 ?a2)
        (= (conc ?a1 ?a2) (conc ?a2 ?a1)))
```

SWSL-FOL

```
forall ?a1, ?a2
  ( conc(?a1, ?a2) :=: conc(?a2 ?a1) ).
```

SWSL-Rules

```
conc(?a1, ?a2) :=: conc(?a2, ?a1) :- activity(?a).
```

Axiom 4 *The function conc is associative.*

KIF

```
(forall (?a1 ?a2 ?a3)
        (= (conc ?a1 (conc ?a2 ?a3)) (conc (conc ?a1 ?a2) ?a3)))
```

SWSL-FOL

```
forall ?a1, ?a2, ?a3
  ( conc(?a1 conc(?a2, ?a3)) :=: conc(conc(?a1, ?a2), ?a3) ).
```

SWSL-Rules

```
conc(?a1, conc(?a2, ?a3)) :=: conc(conc(?a1, ?a2), ?a3) :- activity(?a).
```

Axiom 5 *The concurrent aggregation of atomic action is an atomic action.*

KIF

```
(forall (?a1 ?a2)
        (iff    (atomic (conc ?a1 ?a2))
                (and    (atomic ?a1)
                        (atomic ?a2))))
```

SWSL-FOL

```
forall ?a1, ?a2
  ( atomic(conc(?a1, ?a2)) <==>
    ( atomic(?a1) and atomic(?a2) ) ).
```

SWSL-Rules

```
neg atomic(?a1) :- atomic(?a2) and neg atomic(conc(?a1, ?a2)).
neg atomic(?a2) :- atomic(?a1) and neg atomic(conc(?a1, ?a2)).
atomic(conc(?a1, ?a2)) :- atomic(?a1) and atomic(?a2).

neg atomic(conc(?a1, ?a2)) :- neg atomic(?a1).
atomic(?a1) :- atomic(conc(?a1, ?a2)).

neg atomic(conc(?a1, ?a2)) :- neg atomic(?a2).
atomic(?a2) :- atomic(conc(?a1, ?a2)).
```

Axiom 6 *An atomic activity ?a1 is a subactivity of an atomic activity ?a2 if and only if ?a2 is an idempotent for ?a1.*

KIF

```
(forall (?a1 ?a2)
        (implies  (and  (atomic ?a1)
                        (atomic ?a2))
                  (iff  (subactivity ?a1 ?a2)
                        (= ?a2 (conc ?a1 ?a2)))))
```

SWSL-FOL

```
forall ?a1, ?a2
```

```
  ( ( atomic(?a1) and atomic(?a2) ) ==>
    ( subactivity(?a1, ?a2) <==>
      ?a2 :=: conc(?a1, ?a2) ) ).
```

SWSL-Rules

```
neg atomic(?a1) :- atomic(?a2) and neg subactivity(?a1, ?a2) and ?a2 :=: conc(?a1, ?a2).
neg atomic(?a2) :- atomic(?a1) and neg subactivity(?a1, ?a2) and ?a2 :=: conc(?a1, ?a2).
subactivity(?a1, ?a2) :- atomic(?a1) and atomic(?a2) and ?a2 :=: conc(?a1, ?a2).
neg ?a2 :=: conc(?a1, ?a2) :- atomic(?a1) and atomic(?a2) and neg subactivity(?a1, ?a2).

neg atomic(?a1) :- atomic(?a2) and subactivity(?a1, ?a2) and neg ?a2 :=: conc(?a1, ?a2).
neg atomic(?a2) :- atomic(?a1) and subactivity(?a1, ?a2) and neg ?a2 :=: conc(?a1, ?a2).
neg subactivity(?a1, ?a2) :- atomic(?a1) and atomic(?a2) and neg ?a2 :=: conc(?a1, ?a2).
?a2 :=: conc(?a1, ?a2) :- atomic(?a1) and atomic(?a2) and subactivity(?a1, ?a2).
```

Axiom 7 *An atomic action has a subactivity if and only if there exists another atomic activity which can be concurrently aggregated.*

KIF

```
(forall (?a1 ?a2)
        (implies  (atomic ?a2)
                  (iff  (subactivity ?a1 ?a2)
                        (exists (?a3)
                                (= ?a2 (conc ?a1 ?a3))))))
```

SWSL-FOL

```
forall ?a1, ?a2
  ( atomic(?a2) ==>
    ( subactivity(?a1, ?a2) <==>
      exists ?a3
        ( ?a2 :=: conc(?a1, ?a3) ) ) ).
```

SWSL-Rules

```
neg atomic(?a2) :- subactivity(?a1,?a2) and neg ?a2 :=: conc(?a1,_#1(?a1,?a2)).
neg subactivity(?a1,?a2) :- atomic(?a2) and neg ?a2 :=: conc(?a1,_#1(?a1,?a2)).
?a2 :=: conc(?a1,_#1(?a1,?a2)) :- atomic(?a2) and subactivity(?a1,?a2).

neg atomic(?a2) :- neg subactivity(?a1,?a2) and ?a2 :=: conc(?a1,?a3).
subactivity(?a1,?a2) :- atomic(?a2) and ?a2 :=: conc(?a1,?a3).
neg ?a2 :=: conc(?a1,?a3) :- atomic(?a2) and neg subactivity(?a1,?a2).
```

Axiom 8 *The semilattice of atomic activities is distributive.*

KIF

```
(forall (?a ?b0 ?b1)
        (implies  (and  (subactivity ?a (conc ?b0 ?b1))
                        (not (primitive ?a)))
                  (exists (?a0 ?a1)
                        (and    (subactivity ?a0 ?a)
                                (subactivity ?a1 ?a)
                                (= ?a (conc ?a0 ?a1))))))
```

SWSL-FOL

```
forall ?a, ?b0, ?b1
  ( ( subactivity(?a, conc(?b0, ?b1)) and neg primitive(?a) ) ==>
    exists ?a0, ?a1
      ( subactivity(?a0, ?a) and
        subactivity(?a1, ?a) and
        ?a :=: conc(?a0, ?a1) ) ).
```

SWSL-Rules

```
neg subactivity(?a,conc(?b0,?b1)) :- neg primitive(?a) and neg subactivity(_#2(?a,?b0,?b1),?a).
primitive(?a) :- subactivity(?a,conc(?b0,?b1)) and neg subactivity(_#2(?a,?b0,?b1),?a).
subactivity(_#2(?a,?b0,?b1),?a) :- subactivity(?a,conc(?b0,?b1)) and neg primitive(?a).

neg subactivity(?a,conc(?b0,?b1)) :- neg primitive(?a) and neg subactivity(_#1(?a,?b0,?b1),?a).
primitive(?a) :- subactivity(?a,conc(?b0,?b1)) and neg subactivity(_#1(?a,?b0,?b1),?a).
subactivity(_#1(?a,?b0,?b1),?a) :- subactivity(?a,conc(?b0,?b1)) and neg primitive(?a).

neg subactivity(?a,conc(?b0,?b1)) :- neg primitive(?a) and neg ?a :=: conc(_#2(?a,?b0,?b1),_#1(?
a,?b0,?b1)).
primitive(?a) :- subactivity(?a,conc(?b0,?b1)) and neg ?a :=: conc(_#2(?a,?b0,?b1),_#1(?a,?b0,?
b1)).
?a :=: conc(_#2(?a,?b0,?b1),_#1(?a,?b0,?b1)) :- subactivity(?a,conc(?b0,?b1)) and neg primitive(?
a).
```

Axiom 9 *Only atomic activities can be elements of the legal occurrence tree.*

KIF

```
(forall (?s ?a)
        (implies  (and  (occurrence ?s ?a)
                        (legal ?s))
                  (atomic ?a)))
```

SWSL-FOL

```
forall ?s, ?a
  ( ( occurrence(?s, ?a) and legal(?s) ) ==>
    atomic(?a) ).
```

SWSL-Rules

```
neg occurrence(?s, ?a) :- legal(?s) and neg atomic(?a).
neg legal(?s) :- occurrence(?s, ?a) and neg atomic(?a).
atomic(?a) :- occurrence(?s, ?a) and legal(?s).
```

A.6 Theory of Complex Activities

Extension Name: complex.th
Theories Required by this Extension: occtree.th, atomic.th, subactivity.th, psl_core.th
Definitional Extensions Required by this Extension: None

Primitive Lexicon

KIF

- `(min_precedes ?s1 ?s2 ?a)`
- `(root ?s ?a)`

SWSL-FOL and SWSL-Rules

- `min_precedes(?s1, ?s2, ?a)`
- `root(?s, ?a)`

Defined Lexicon

KIF

- `(do ?a ?s1 ?s2)`
- `(leaf ?s ?a)`
- `(next_subocc ?s1 ?s2 ?a)`
- `(subtree ?s ?a1 ?a2)`
- `(sibling ?s1 ?s2 ?a)`

SWSL-FOL and SWSL-Rules

- `do(?a, ?s1, ?s2)`
- `leaf(?s, ?a)`
- `next_subocc(?s1, ?s2, ?a)`
- `subtree(?s, ?a1, ?a2)`
- `sibling(?s1, ?s2, ?a)`

Axioms

Axiom 1 *Occurrences in the activity tree for an activity correspond to atomic subactivity occurrences of the activity.*

KIF

```
(forall (?a ?s1 ?s2)
        (implies  (min_precedes ?s1 ?s2 ?a)
                  (exists (?a1 ?ap)
                        (and    (subactivity ?a1 ?a)
                                (atomic ?ap)
                                (subactivity ?a1 ?ap)
                                (occurrence_of ?s2 ?ap)))))
```

SWSL-FOL

```
forall ?a, ?s1, ?s2
```

```
  min_precedes(?s1, ?s2, ?a) ==>
  exists ?a1, ?ap
    ( subactivity(?a1, ?a) and
      atomic(?ap) and
      subactivity(?a1, ?ap) and
      occurrence_of(?s2, ?ap) ).
```

SWSL-Rules

```
neg min_precedes(?s1, ?s2, ?a) :- neg subactivity(_#1(?a, ?s1, ?s2), ?a).
subactivity(_#1(?a, ?s1, ?s2), ?a) :- min_precedes(?s1, ?s2, ?a).

neg min_precedes(?s1, ?s2, ?a) :- neg atomic(_#2(?a, ?s1, ?s2)).
atomic(_#2(?a, ?s1, ?s2)) :- min_precedes(?s1, ?s2, ?a).

neg min_precedes(?s1, ?s2, ?a) :- neg subactivity(_#1(?a, ?s1, ?s2), _#2(?a, ?s1, ?s2)).
subactivity(_#1(?a, ?s1, ?s2), _#2(?a, ?s1, ?s2)) :- min_precedes(?s1, ?s2, ?a).

neg min_precedes(?s1, ?s2, ?a) :- neg occurrence_of(?s2, _#2(?a, ?s1, ?s2)).
occurrence_of(?s2, _#2(?a, ?s1, ?s2)) :- min_precedes(?s1, ?s2, ?a).
```

Axiom 2 *Occurrences in the activity tree for an activity correspond to atomic subactivity occurrences of the activity.*

KIF

```
(forall (?a ?s1 ?s2)
        (implies  (min_precedes ?s1 ?s2 ?a)
                  (exists (?a2 ?ap)
                        (and    (subactivity ?a2 ?a)
                                (atomic ?ap)
                                (subactivity ?a2 ?ap)
                                (occurrence_of ?s1 ?ap)))))
```

SWSL-FOL

```
forall ?a, ?s1, ?s2
  ( min_precedes(?s1, ?s2, ?a) ==>
    exists ?a2, ?ap
      ( subactivity(?a2, ?a) and
        atomic(?ap) and
        subactivity(?a2, ?ap) and
        occurrence_of(?s1, ?ap) ) ).
```

SWSL-Rules

```
neg min_precedes(?s1, ?s2, ?a) :- neg subactivity(_#1(?a, ?s1, ?s2), ?a).
subactivity(_#1(?a, ?s1, ?s2), ?a) :- min_precedes(?s1, ?s2, ?a).

neg min_precedes(?s1, ?s2, ?a) :- neg atomic(_#2(?a, ?s1, ?s2)).
atomic(_#2(?a, ?s1, ?s2)) :- min_precedes(?s1, ?s2, ?a).

neg min_precedes(?s1, ?s2, ?a) :- neg subactivity(_#1(?a, ?s1, ?s2), _#2(?a, ?s1, ?s2)).
subactivity(_#1(?a, ?s1, ?s2), _#2(?a, ?s1, ?s2)) :- min_precedes(?s1, ?s2, ?a).

neg min_precedes(?s1, ?s2, ?a) :- neg occurrence_of(?s1, _#2(?a, ?s1, ?s2)).
occurrence_of(?s1, _#2(?a, ?s1, ?s2)) :- min_precedes(?s1, ?s2, ?a).
```

Axiom 3 *Root occurrences in the activity tree correspond to atomic subactivity occurrences of the activity.*

KIF

```
(forall (?a ?s1)
        (implies  (root ?s1 ?a)
                  (exists (?a2 ?ap)
                        (and    (subactivity ?a2 ?a)
                                (atomic ?ap)
                                (subactivity ?a2 ?ap)
                                (occurrence_of ?s1 ?ap)))))
```

SWSL-FOL

```
forall ?a ?s1
  ( root(?s1 ?a) ==>
    exists ?a2 ?ap
      ( subactivity(?a2 ?a) and
        atomic(?ap) and
        subactivity(?a2 ?ap) and
        occurrence_of(?s1 ?ap) ) ).
```

SWSL-Rules

```
neg root(?s1, ?a) :- neg subactivity(_#1(?a, ?s1, ?s2), ?a).
subactivity(_#1(?a, ?s1, ?s2), ?a) :- root(?s1, ?a).

neg root(?s1, ?a) :- neg atomic(_#2(?a, ?s1, ?s2)).
atomic(_#2(?a, ?s1, ?s2)) :- root(?s1, ?a).

neg root(?s1, ?a) :- neg subactivity(_#1(?a, ?s1, ?s2), _#2(?a, ?s1, ?s2)).
subactivity(_#1(?a, ?s1, ?s2), _#2(?a, ?s1, ?s2)) :- root(?s1, ?a).

neg root(?s1, ?a) :- neg occurrence_of(?s1, _#2(?a, ?s1, ?s2)).
occurrence_of(?s1, _#2(?a, ?s1, ?s2)) :- root(?s1, ?a).
```

Axiom 4 *All activity trees have a root subactivity occurrence.*

KIF

```
(forall (?s1 ?s2 ?a)
        (implies  (min_precedes ?s1 ?s2 ?a)
                  (exists (?s3)
                        (and    (root ?s3 ?a)
                                (or     (min_precedes ?s3 ?s1 ?a)
                                        (= ?s3 ?s1))))))
```

SWSL-FOL

```
forall ?s1, ?s2, ?a
  ( min_precedes(?s1, ?s2, ?a) ==>
    exists ?s3
      ( root(?s3, ?a) and
        ( min_precedes(?s3, ?s1, ?a) or ?s3 :=: ?s1 ) ) ).
```

SWSL-Rules

```
neg min_precedes(?s1,?s2,?a) :- neg root(_#1(?s1,?s2,?a),?a).
root(_#1(?s1,?s2,?a),?a) :- min_precedes(?s1,?s2,?a).

neg min_precedes(?s1,?s2,?a) :- neg min_precedes(_#1(?s1,?s2,?a),?s1,?a) and neg _#1(?s1,?s2,?a) :
=: ?s1.
min_precedes(_#1(?s1,?s2,?a),?s1,?a) :- min_precedes(?s1,?s2,?a) and neg _#1(?s1,?s2,?a) :=: ?s1.
_#1(?s1,?s2,?a) :=: ?s1 :- min_precedes(?s1,?s2,?a) and neg min_precedes(_#1(?s1,?s2,?a),?s1,?a).
```

Axiom 5 *No subactivity occurrences in an activity tree occur earlier than the root subactivity occurrence.*

KIF

```
(forall (?s ?a)
        (implies  (root ?s ?a)
                  (not (exists (?s2)
                        (min_precedes ?s2 ?s ?a)))))
```

SWSL-FOL

```
forall ?s, ?a
  ( root(?s, ?a) ==>
    neg exists ?s2
      ( min_precedes(?s2, ?s, ?a) ) ).
```

SWSL-Rules

```
neg root(?s, ?a) :- min_precedes(?s2, ?s, ?a).
neg min_precedes(?s2, ?s, ?a) :- root(?s, ?a).
```

Axiom 6 *An activity tree is a subtree of the occurrence tree.*

KIF

```
(forall (?s1 ?s2 ?a)
        (implies  (min_precedes ?s1 ?s2 ?a)
                  (exists (?s0)
                        (and    (initial ?s0)
                                (or     (precedes ?s0 ?s1)
                                        (= ?s0 ?s1))
                                (precedes ?s1 ?s2)))))
```

SWSL-FOL

```
forall ?s1, ?s2, ?a
  ( min_precedes(?s1, ?s2, ?a) ==>
    exists ?s0
      ( initial(?s0) and
        ( precedes(?s0, ?s1) or ?s0 :=: ?s1 ) and
        precedes(?s1, ?s2) ) ).
```

SWSL-Rules

```
neg min_precedes(?s1, ?s2, ?a) :- neg initial(_#1(?s1,?s2,?a)).
initial(_#1(?s1,?s2,?a)) :- min_precedes(?s1, ?s2, ?a).

neg min_precedes(?s1, ?s2, ?a) :- neg precedes(_#1(?s1,?s2,?a), ?s1) and neg _#1(?s1,?s2,?a) :=: ?
s1.
precedes(_#1(?s1,?s2,?a), ?s1) :- min_precedes(?s1, ?s2, ?a) and neg _#1(?s1,?s2,?a) :=: ?s1.
_#1(?s1,?s2,?a) :=: ?s1 :- min_precedes(?s1, ?s2, ?a) and neg precedes(_#1(?s1,?s2,?a), ?s1).

neg min_precedes(?s1, ?s2, ?a) :- neg precedes(?s1, ?s2).
precedes(?s1, ?s2) :- min_precedes(?s1, ?s2, ?a).
```

Axiom 7 *Root occurrences are elements of the occurrence tree.*

KIF

```
(forall (?s ?a)
        (implies  (root ?s ?a)
                  (exists (?s0)
                        (and    (initial ?s0)
                                (or     (precedes ?s0 ?s)
                                        (= ?s0 ?s))))))
```

SWSL-FOL

```
forall ?s, ?a
  root(?s ?a) ==>
  exists ?s0
    ( initial(?s0) and
      ( precedes(?s0, ?s) or ?s0 :=: ?s ) ).
```

SWSL-Rules

```
neg root(?s ?a)  :- neg initial(_#1(?s,?a)).
initial(_#1(?s,?a)) :- root(?s ?a) .

neg root(?s ?a) :- neg precedes(_#1(?s,?a), ?s) and neg _#1(?s,?a) :=: ?s.
precedes(_#1(?s,?a), ?s) :- root(?s ?a) and neg _#1(?s,?a) :=: ?s.
_#1(?s,?a) :=: ?s :- root(?s ?a) and neg precedes(_#1(?s,?a), ?s).
```

Axiom 8 *Every atomic activity occurrence is an activity tree containing only one occurrence.*

KIF

```
(forall (?a1 ?a2 ?s)
        (implies  (and  (atomic ?a1)
                        (occurrence_of ?s ?a1)
                        (subactivity ?a2 ?a1))
                  (root ?s ?a2)))
```

SWSL-FOL

```
forall ?a1, ?a2, ?s
  ( ( atomic(?a1) and occurrence_of(?s, ?a1) and subactivity(?a2, ?a1) ) ==>
    root(?s, ?a2) ).
```

SWSL-Rules

```
neg atomic(?a1) :- occurrence_of(?s, ?a1) and subactivity(?a2, ?a1) and neg root(?s, ?a2).
neg occurrence_of(?s, ?a1) :- atomic(?a1) and subactivity(?a2, ?a1) and neg root(?s, ?a2).
neg subactivity(?a2, ?a1) :- atomic(?a1) and occurrence_of(?s, ?a1) and neg root(?s, ?a2).
root(?s, ?a2) :- atomic(?a1) and occurrence_of(?s, ?a1) and subactivity(?a2, ?a1).
```

Axiom 9 *Activity trees are discrete.*

KIF

```
(forall (?s1 ?s2)
        (implies  (min_precedes ?s1 ?s2 ?a)
                  (exists (?s3)
                        (and    (next_subocc ?s1 ?s3 ?a)
                                (or     (min_precedes ?s3 ?s2 ?a)
                                        (= ?s3 ?s2))))))
```

SWSL-FOL

```
forall ?s1 ?s2
  ( min_precedes(?s1, ?s2, ?a) ==>
    exists ?s3
      ( next_subocc(?s1, ?s3, ?a) and
        ( min_precedes(?s3, ?s2, ?a) or ?s3 :=: ?s2 ) ) ).
```

SWSL-Rules

```
neg min_precedes(?s1, ?s2, ?a) :- neg next_subocc(?s1, _#1(?s1,?s2), ?a).
next_subocc(?s1, _#1(?s1,?s2), ?a) :- min_precedes(?s1, ?s2, ?a).

neg min_precedes(?s1, ?s2, ?a) :- neg min_precedes(_#1(?s1,?s2), ?s2, ?a) and neg _#1(?s1,?s2) :
=: ?s2.
min_precedes(_#1(?s1,?s2), ?s2, ?a) :- min_precedes(?s1, ?s2, ?a) and neg _#1(?s1,?s2) :=: ?s2.
_#1(?s1,?s2) :=: ?s2 :- min_precedes(?s1, ?s2, ?a) and neg min_precedes(_#1(?s1,?s2), ?s2, ?a).
```

Axiom 10 *Subactivity occurrences on the same branch of the occurrence tree are on the same branch of the activity tree.*

KIF

```
(forall (?a ?s1 ?s2 ?s3)
        (implies  (and  (min_precedes ?s1 ?s2 ?a)
                        (min_precedes ?s1 ?s3 ?a)
                        (precedes ?s2 ?s3))
                  (min_precedes ?s2 ?s3 ?a)))
```

SWSL-FOL

```
forall ?a, ?s1, ?s2, ?s3
  ( ( min_precedes(?s1, ?s2, ?a) and
      min_precedes(?s1, ?s3, ?a) and
      precedes(?s2, ?s3) ) ==>
    min_precedes(?s2, ?s3, ?a) ).
```

SWSL-Rules

```
neg min_precedes(?s1, ?s2, ?a) :- min_precedes(?s1, ?s3, ?a) and precedes(?s2, ?s3) and neg
min_precedes(?s2, ?s3, ?a).
neg min_precedes(?s1, ?s3, ?a) :- min_precedes(?s1, ?s2, ?a) and precedes(?s2, ?s3) and neg
min_precedes(?s2, ?s3, ?a).
neg precedes(?s2, ?s3) :- min_precedes(?s1, ?s2, ?a) and min_precedes(?s1, ?s3, ?a) and neg
min_precedes(?s2, ?s3, ?a).
min_precedes(?s2, ?s3, ?a) :- min_precedes(?s1, ?s2, ?a) and min_precedes(?s1, ?s3, ?a) and
precedes(?s2, ?s3).
```

Axiom 11 *The activity tree for a complex subactivity occurrence is a subtree of the activity tree for the activity occurrence.*

KIF

```
(forall (?a1 ?a2)
        (implies  (subactivity ?a1 ?a2)
                  (not (exists (?s)
                        (subtree ?s ?a2 ?a1)))))
```

SWSL-FOL

```
forall ?a1, ?a2
  ( subactivity(?a1, ?a2) ==>
    neg exists ?s
          ( subtree(?s, ?a2, ?a1) ) ).
```

SWSL-Rules

```
neg subactivity(?a1, ?a2) :- subtree(?s, ?a2, ?a1).
neg subtree(?s, ?a2, ?a1) :- subactivity(?a1, ?a2).
```

Definitions

Definition 1 *An occurrence is the leaf of an activity tree if and only if there exists an earlier atomic subactivity occurrence but there does not exist a later atomic subactivity occurrence.*

KIF

```
(forall (?s ?a) (iff (leaf ?s ?a)
(exists (?s1)
        (and    (or     (root ?s ?a)
                        (min_precedes ?s1 ?s ?a))
                (not (exists (?s2)
                        (min_precedes ?s ?s2 ?a)))))))
```

SWSL-FOL

```
forall ?s ?a
  ( leaf(?s, ?a) <==>
    exists ?s1
      ( ( root(?s, ?a) or min_precedes(?s1, ?s, ?a) ) and
        neg exists ?s2
              ( min_precedes(?s, ?s2, ?a) ) ) ).
```

SWSL-Rules

```
neg leaf(?s,?a) :- neg root(?s,?a) and neg min_precedes(_#1(?s,?a),?s,?a).
root(?s,?a) :- leaf(?s,?a) and neg min_precedes(_#1(?s,?a),?s,?a).
min_precedes(_#1(?s,?a),?s,?a) :- leaf(?s,?a) and neg root(?s,?a).

neg leaf(?s,?a) :- min_precedes(?s,?s2,?a).
neg min_precedes(?s,?s2,?a) :- leaf(?s,?a).

leaf(?s,?a) :- root(?s,?a) and neg min_precedes(?s,_#2(?s,?a,?s1),?a).
neg root(?s,?a) :- neg leaf(?s,?a) and neg min_precedes(?s,_#2(?s,?a,?s1),?a).
min_precedes(?s,_#2(?s,?a,?s1),?a) :- neg leaf(?s,?a) and root(?s,?a).

leaf(?s,?a) :- min_precedes(?s1,?s,?a) and neg min_precedes(?s,_#2(?s,?a,?s1),?a).
neg min_precedes(?s1,?s,?a) :- neg leaf(?s,?a) and neg min_precedes(?s,_#2(?s,?a,?s1),?a).
min_precedes(?s,_#2(?s,?a,?s1),?a) :- neg leaf(?s,?a) and min_precedes(?s1,?s,?a).
```

Definition 2 *The do relation specifies the initial and final atomic subactivity occurrences of an occurrence of an activity.*

KIF

```
(forall (?a ?s1 ?s2) (iff (do ?a ?s1 ?s2)
(and    (root ?s1 ?a)
        (leaf ?s2 ?a)
        (or     (min_precedes ?s1 ?s2 ?a)
                (= ?s1 ?s2)))))
```

SWSL-FOL

```
forall ?a, ?s1, ?s2
  ( do(?a, ?s1, ?s2) <==>
    ( root(?s1, ?a) and
      leaf(?s2, ?a) and
      ( min_precedes(?s1, ?s2, ?a) or ?s1 :=: ?s2) ) ).
```

SWSL-Rules

```
neg do(?a,?s1,?s2) :- neg root(?s1,?a).
root(?s1,?a) :- do(?a,?s1,?s2).

neg do(?a,?s1,?s2) :- neg leaf(?s2,?a).
leaf(?s2,?a) :- do(?a,?s1,?s2).

neg do(?a,?s1,?s2) :- neg min_precedes(?s1,?s2,?a) and neg ?s1 :=: ?s2.
min_precedes(?s1,?s2,?a) :- do(?a,?s1,?s2) and neg ?s1 :=: ?s2.
?s1 :=: ?s2 :- do(?a,?s1,?s2) and neg min_precedes(?s1,?s2,?a).

do(?a,?s1,?s2) :- root(?s1,?a) and leaf(?s2,?a) and min_precedes(?s1,?s2,?a).
neg root(?s1,?a) :- neg do(?a,?s1,?s2) and leaf(?s2,?a) and min_precedes(?s1,?s2,?a).
neg leaf(?s2,?a) :- neg do(?a,?s1,?s2) and root(?s1,?a) and min_precedes(?s1,?s2,?a).
neg min_precedes(?s1,?s2,?a) :- neg do(?a,?s1,?s2) and root(?s1,?a) and leaf(?s2,?a).

do(?a,?s1,?s2) :- root(?s1,?a) and leaf(?s2,?a) and ?s1 :=: ?s2.
```

```
neg root(?s1,?a) :- neg do(?a,?s1,?s2) and leaf(?s2,?a) and ?s1 :=: ?s2.
neg leaf(?s2,?a) :- neg do(?a,?s1,?s2) and root(?s1,?a) and ?s1 :=: ?s2.
neg ?s1 :=: ?s2 :- neg do(?a,?s1,?s2) and root(?s1,?a) and leaf(?s2,?a).
```

Definition 3 *An activity occurrence ?s2 is the next subactivity occurrence after ?s1 in an activity tree for ?a if and only of ?s1 precedes ?s2 in the tree and there does not exist a subactivity occurrence that is between them in the tree.*

KIF

```
(forall (?s1 ?s2 ?a) (iff (next_subocc ?s1 ?s2 ?a)
(and    (min_precedes ?s1 ?s2 ?a)
        (not (exists (?s3)
                (and    (min_precedes ?s1 ?s3 ?a)
                        (min_precedes ?s3 ?s2 ?a)))))))
```

SWSL-FOL

```
forall ?s1, ?s2, ?a
  ( next_subocc(?s1, ?s2, ?a) <==>
    ( min_precedes(?s1, ?s2, ?a) and
      neg exists ?s3
            ( min_precedes(?s1, ?s3, ?a) and
              min_precedes(?s3, ?s2, ?a) ) ) ).
```

SWSL-Rules

```
neg next_subocc(?s1,?s2,?a) :- neg min_precedes(?s1,?s2,?a).
min_precedes(?s1,?s2,?a) :- next_subocc(?s1,?s2,?a).

neg next_subocc(?s1,?s2,?a) :- min_precedes(?s1,?s3,?a) and min_precedes(?s3,?s2,?a).
neg min_precedes(?s1,?s3,?a) :- next_subocc(?s1,?s2,?a) and min_precedes(?s3,?s2,?a).
neg min_precedes(?s3,?s2,?a) :- next_subocc(?s1,?s2,?a) and min_precedes(?s1,?s3,?a).

next_subocc(?s1,?s2,?a) :- min_precedes(?s1,?s2,?a) and neg min_precedes(?s1,_#1(?s1,?s2,?a),?a).
neg min_precedes(?s1,?s2,?a) :- neg next_subocc(?s1,?s2,?a) and neg min_precedes(?s1,_#1(?s1,?s2,?
a),?a).
min_precedes(?s1,_#1(?s1,?s2,?a),?a) :- neg next_subocc(?s1,?s2,?a) and min_precedes(?s1,?s2,?a).

next_subocc(?s1,?s2,?a) :- min_precedes(?s1,?s2,?a) and neg min_precedes(_#1(?s1,?s2,?a),?s2,?a).
neg min_precedes(?s1,?s2,?a) :- neg next_subocc(?s1,?s2,?a) and neg min_precedes(_#1(?s1,?s2,?a),?
s2,?a).
min_precedes(_#1(?s1,?s2,?a),?s2,?a) :- neg next_subocc(?s1,?s2,?a) and min_precedes(?s1,?s2,?a).
```

Definition 4 *The activity tree for ?a1 with root occurrence ?s1 is a subtree of an activity tree for ?a2 if and only if every atomic asubactivity occurrence in the activity tree for ?a1 is an element of the activity tree for ?a2.*

KIF

```
(forall (?s1 ?a1 ?a2) (iff (subtree ?s1 ?a1 ?a2)
(and    (root ?s1 ?a1)
        (exists (?s2 ?s3)
                (and    (root ?s2 ?a2)
                        (min_precedes ?s1 ?s2 ?a1)
                        (min_precedes ?s1 ?s3 ?a1)
                        (not (min_precedes ?s2 ?s3 ?a2)))))))
```

SWSL-FOL

```
forall ?s1, ?a1, ?a2
  ( subtree(?s1, ?a1, ?a2) <==>
    ( root(?s1, ?a1) and
      exists ?s2, ?s2
        ( root(?s2, ?a2) and min_precedes(?s1, ?s2, ?a1) ) and
        ( min_precedes(?s1, ?s3, ?a1) and neg min_precedes(?s2, ?s3, ?a2) ) ) ).
```

SWSL-Rules

```
neg subtree(?s1,?a1,?a2) :- neg root(?s1,?a1).
root(?s1,?a1) :- subtree(?s1,?a1,?a2).

neg subtree(?s1,?a1,?a2) :- neg root(_#2(?s1,?a1,?a2),?a2).
root(_#2(?s1,?a1,?a2),?a2) :- subtree(?s1,?a1,?a2).

neg subtree(?s1,?a1,?a2) :- neg min_precedes(?s1,_#2(?s1,?a1,?a2),?a1).
min_precedes(?s1,_#2(?s1,?a1,?a2),?a1) :- subtree(?s1,?a1,?a2).

neg subtree(?s1,?a1,?a2) :- neg min_precedes(?s1,_#1(?s1,?a1,?a2),?a1).
min_precedes(?s1,_#1(?s1,?a1,?a2),?a1) :- subtree(?s1,?a1,?a2).

neg subtree(?s1,?a1,?a2) :- min_precedes(_#2(?s1,?a1,?a2),_#1(?s1,?a1,?a2),?a2).
neg min_precedes(_#2(?s1,?a1,?a2),_#1(?s1,?a1,?a2),?a2) :- subtree(?s1,?a1,?a2).

subtree(?s1,?a1,?a2) :- root(?s1,?a1) and root(?s2,?a2) and min_precedes(?s1,?s2,?a1) and
min_precedes(?s1,?s3,?a1) and neg min_precedes(?s2,?s3,?a2).
neg root(?s1,?a1) :- neg subtree(?s1,?a1,?a2) and root(?s2,?a2) and min_precedes(?s1,?s2,?a1) and
min_precedes(?s1,?s3,?a1) and neg min_precedes(?s2,?s3,?a2).
neg root(?s2,?a2) :- neg subtree(?s1,?a1,?a2) and root(?s1,?a1) and min_precedes(?s1,?s2,?a1) and
min_precedes(?s1,?s3,?a1) and neg min_precedes(?s2,?s3,?a2).
neg min_precedes(?s1,?s2,?a1) :- neg subtree(?s1,?a1,?a2) and root(?s1,?a1) and root(?s2,?a2) and
min_precedes(?s1,?s3,?a1) and neg min_precedes(?s2,?s3,?a2).
neg min_precedes(?s1,?s3,?a1) :- neg subtree(?s1,?a1,?a2) and root(?s1,?a1) and root(?s2,?a2) and
min_precedes(?s1,?s2,?a1) and neg min_precedes(?s2,?s3,?a2).
min_precedes(?s2,?s3,?a2) :- neg subtree(?s1,?a1,?a2) and root(?s1,?a1) and root(?s2,?a2) and
min_precedes(?s1,?s2,?a1) and min_precedes(?s1,?s3,?a1).
```

Definition 5 *The atomic subactivity occurrences ?s1 and ?s2 are siblings in an activity tree for ?a iff they either have a common predecessor in the activity tree or they are both roots of activity trees that have a common predecessor in the occurrence tree.*

KIF

```
(forall (?s1 ?s2 ?a) (iff (sibling ?s1 ?s2 ?a)
(or     (exists (?s3)
                (and    (next_subocc ?s3 ?s1 ?a)
                        (next_subocc ?s3 ?s2 ?a)))
        (and    (root ?s1 ?a)
                (root ?s2 ?a)
                (or     (and    (initial ?s1)
                                (initial ?s2))
                        (exists (?s4 ?a1 ?a2)
                                (and    (= ?s1 (successor ?a1 ?s4))
                                        (= ?s2 (successor ?a2 ?s4)))))))))
```

SWSL-FOL

```
forall ?s1, ?s2, ?a
  ( sibling(?s1, ?s2, ?a) <==>
    ( exists ?s3
        ( next_subocc(?s3, ?s1, ?a) and next_subocc(?s3, ?s2, ?a) ) or
      ( root(?s1, ?a) and
        root(?s2, ?a) and
        ( ( initial(?s1) and initial(?s2) ) or
          exists ?s4, ?a1, ?a2
            ( ?s1 :=: successor(?a1, ?s4) and
              ?s2 :=: successor(?a2, ?s4) ) ) ) ) ).
```

SWSL-Rules

```
neg sibling(?s1,?s2,?a) :- neg next_subocc(_#1(?s1,?s2,?a),?s1,?a) and neg root(?s1,?a).
next_subocc(_#1(?s1,?s2,?a),?s1,?a) :- sibling(?s1,?s2,?a) and neg root(?s1,?a).
root(?s1,?a) :- sibling(?s1,?s2,?a) and neg next_subocc(_#1(?s1,?s2,?a),?s1,?a).

neg sibling(?s1,?s2,?a) :- neg next_subocc(_#1(?s1,?s2,?a),?s1,?a) and neg root(?s2,?a).
next_subocc(_#1(?s1,?s2,?a),?s1,?a) :- sibling(?s1,?s2,?a) and neg root(?s2,?a).
root(?s2,?a) :- sibling(?s1,?s2,?a) and neg next_subocc(_#1(?s1,?s2,?a),?s1,?a).

neg sibling(?s1,?s2,?a) :- neg next_subocc(_#1(?s1,?s2,?a),?s1,?a) and neg initial(?s1) and neg ?
s1 :=: successor(_#3(?s1,?s2,?a),_#4(?s1,?s2,?a)).
next_subocc(_#1(?s1,?s2,?a),?s1,?a) :- sibling(?s1,?s2,?a) and neg initial(?s1) and neg ?s1 :=:
successor(_#3(?s1,?s2,?a),_#4(?s1,?s2,?a)).
initial(?s1) :- sibling(?s1,?s2,?a) and neg next_subocc(_#1(?s1,?s2,?a),?s1,?a) and neg ?s1 :=:
successor(_#3(?s1,?s2,?a),_#4(?s1,?s2,?a)).
?s1 :=: successor(_#3(?s1,?s2,?a),_#4(?s1,?s2,?a)) :- sibling(?s1,?s2,?a) and neg next_subocc(_#1
(?s1,?s2,?a),?s1,?a) and neg initial(?s1).

neg sibling(?s1,?s2,?a) :- neg next_subocc(_#1(?s1,?s2,?a),?s1,?a) and neg initial(?s1) and neg ?
s2 :=: successor(_#2(?s1,?s2,?a),_#4(?s1,?s2,?a)).
next_subocc(_#1(?s1,?s2,?a),?s1,?a) :- sibling(?s1,?s2,?a) and neg initial(?s1) and neg ?s2 :=:
successor(_#2(?s1,?s2,?a),_#4(?s1,?s2,?a)).
initial(?s1) :- sibling(?s1,?s2,?a) and neg next_subocc(_#1(?s1,?s2,?a),?s1,?a) and neg ?s2 :=:
successor(_#2(?s1,?s2,?a),_#4(?s1,?s2,?a)).
?s2 :=: successor(_#2(?s1,?s2,?a),_#4(?s1,?s2,?a)) :- sibling(?s1,?s2,?a) and neg next_subocc(_#1
(?s1,?s2,?a),?s1,?a) and neg initial(?s1).

neg sibling(?s1,?s2,?a) :- neg next_subocc(_#1(?s1,?s2,?a),?s1,?a) and neg initial(?s2) and neg ?
s1 :=: successor(_#3(?s1,?s2,?a),_#4(?s1,?s2,?a)).
next_subocc(_#1(?s1,?s2,?a),?s1,?a) :- sibling(?s1,?s2,?a) and neg initial(?s2) and neg ?s1 :=:
successor(_#3(?s1,?s2,?a),_#4(?s1,?s2,?a)).
initial(?s2) :- sibling(?s1,?s2,?a) and neg next_subocc(_#1(?s1,?s2,?a),?s1,?a) and neg ?s1 :=:
successor(_#3(?s1,?s2,?a),_#4(?s1,?s2,?a)).
?s1 :=: successor(_#3(?s1,?s2,?a),_#4(?s1,?s2,?a)) :- sibling(?s1,?s2,?a) and neg next_subocc(_#1
(?s1,?s2,?a),?s1,?a) and neg initial(?s2).

neg sibling(?s1,?s2,?a) :- neg next_subocc(_#1(?s1,?s2,?a),?s1,?a) and neg initial(?s2) and neg ?
s2 :=: successor(_#2(?s1,?s2,?a),_#4(?s1,?s2,?a)).
next_subocc(_#1(?s1,?s2,?a),?s1,?a) :- sibling(?s1,?s2,?a) and neg initial(?s2) and neg ?s2 :=:
successor(_#2(?s1,?s2,?a),_#4(?s1,?s2,?a)).
initial(?s2) :- sibling(?s1,?s2,?a) and neg next_subocc(_#1(?s1,?s2,?a),?s1,?a) and neg ?s2 :=:
successor(_#2(?s1,?s2,?a),_#4(?s1,?s2,?a)).
?s2 :=: successor(_#2(?s1,?s2,?a),_#4(?s1,?s2,?a)) :- sibling(?s1,?s2,?a) and neg next_subocc(_#1
(?s1,?s2,?a),?s1,?a) and neg initial(?s2).

neg sibling(?s1,?s2,?a) :- neg next_subocc(_#1(?s1,?s2,?a),?s2,?a) and neg root(?s1,?a).
```

```
next_subocc(_#1(?s1,?s2,?a),?s2,?a) :- sibling(?s1,?s2,?a) and neg root(?s1,?a).
root(?s1,?a) :- sibling(?s1,?s2,?a) and neg next_subocc(_#1(?s1,?s2,?a),?s2,?a).

neg sibling(?s1,?s2,?a) :- neg next_subocc(_#1(?s1,?s2,?a),?s2,?a) and neg root(?s2,?a).
next_subocc(_#1(?s1,?s2,?a),?s2,?a) :- sibling(?s1,?s2,?a) and neg root(?s2,?a).
root(?s2,?a) :- sibling(?s1,?s2,?a) and neg next_subocc(_#1(?s1,?s2,?a),?s2,?a).

neg sibling(?s1,?s2,?a) :- neg next_subocc(_#1(?s1,?s2,?a),?s2,?a) and neg initial(?s1) and neg ?
s1 :=: successor(_#3(?s1,?s2,?a),_#4(?s1,?s2,?a)).
next_subocc(_#1(?s1,?s2,?a),?s2,?a) :- sibling(?s1,?s2,?a) and neg initial(?s1) and neg ?s1 :=:
successor(_#3(?s1,?s2,?a),_#4(?s1,?s2,?a)).
initial(?s1) :- sibling(?s1,?s2,?a) and neg next_subocc(_#1(?s1,?s2,?a),?s2,?a) and neg ?s1 :=:
successor(_#3(?s1,?s2,?a),_#4(?s1,?s2,?a)).
?s1 :=: successor(_#3(?s1,?s2,?a),_#4(?s1,?s2,?a)) :- sibling(?s1,?s2,?a) and neg next_subocc(_#1
(?s1,?s2,?a),?s2,?a) and neg initial(?s1).

neg sibling(?s1,?s2,?a) :- neg next_subocc(_#1(?s1,?s2,?a),?s2,?a) and neg initial(?s1) and neg ?
s2 :=: successor(_#2(?s1,?s2,?a),_#4(?s1,?s2,?a)).
next_subocc(_#1(?s1,?s2,?a),?s2,?a) :- sibling(?s1,?s2,?a) and neg initial(?s1) and neg ?s2 :=:
successor(_#2(?s1,?s2,?a),_#4(?s1,?s2,?a)).
initial(?s1) :- sibling(?s1,?s2,?a) and neg next_subocc(_#1(?s1,?s2,?a),?s2,?a) and neg ?s2 :=:
successor(_#2(?s1,?s2,?a),_#4(?s1,?s2,?a)).
?s2 :=: successor(_#2(?s1,?s2,?a),_#4(?s1,?s2,?a)) :- sibling(?s1,?s2,?a) and neg next_subocc(_#1
(?s1,?s2,?a),?s2,?a) and neg initial(?s1).

neg sibling(?s1,?s2,?a) :- neg next_subocc(_#1(?s1,?s2,?a),?s2,?a) and neg initial(?s2) and neg ?
s1 :=: successor(_#3(?s1,?s2,?a),_#4(?s1,?s2,?a)).
next_subocc(_#1(?s1,?s2,?a),?s2,?a) :- sibling(?s1,?s2,?a) and neg initial(?s2) and neg ?s1 :=:
successor(_#3(?s1,?s2,?a),_#4(?s1,?s2,?a)).
initial(?s2) :- sibling(?s1,?s2,?a) and neg next_subocc(_#1(?s1,?s2,?a),?s2,?a) and neg ?s1 :=:
successor(_#3(?s1,?s2,?a),_#4(?s1,?s2,?a)).
?s1 :=: successor(_#3(?s1,?s2,?a),_#4(?s1,?s2,?a)) :- sibling(?s1,?s2,?a) and neg next_subocc(_#1
(?s1,?s2,?a),?s2,?a) and neg initial(?s2).

neg sibling(?s1,?s2,?a) :- neg next_subocc(_#1(?s1,?s2,?a),?s2,?a) and neg initial(?s2) and neg ?
s2 :=: successor(_#2(?s1,?s2,?a),_#4(?s1,?s2,?a)).
next_subocc(_#1(?s1,?s2,?a),?s2,?a) :- sibling(?s1,?s2,?a) and neg initial(?s2) and neg ?s2 :=:
successor(_#2(?s1,?s2,?a),_#4(?s1,?s2,?a)).
initial(?s2) :- sibling(?s1,?s2,?a) and neg next_subocc(_#1(?s1,?s2,?a),?s2,?a) and neg ?s2 :=:
successor(_#2(?s1,?s2,?a),_#4(?s1,?s2,?a)).
?s2 :=: successor(_#2(?s1,?s2,?a),_#4(?s1,?s2,?a)) :- sibling(?s1,?s2,?a) and neg next_subocc(_#1
(?s1,?s2,?a),?s2,?a) and neg initial(?s2).

sibling(?s1,?s2,?a) :- next_subocc(?s3,?s1,?a) and next_subocc(?s3,?s2,?a).
neg next_subocc(?s3,?s1,?a) :- neg sibling(?s1,?s2,?a) and next_subocc(?s3,?s2,?a).
neg next_subocc(?s3,?s2,?a) :- neg sibling(?s1,?s2,?a) and next_subocc(?s3,?s1,?a).

neg sibling(?s1,?s2,?a) :- root(?s1,?a) and root(?s2,?a) and initial(?s1) and initial(?s2).
root(?s1,?a) :- neg sibling(?s1,?s2,?a) and root(?s2,?a) and initial(?s1) and initial(?s2).
root(?s2,?a) :- neg sibling(?s1,?s2,?a) and root(?s1,?a) and initial(?s1) and initial(?s2).
initial(?s1) :- neg sibling(?s1,?s2,?a) and root(?s1,?a) and root(?s2,?a) and initial(?s2).
initial(?s2) :- neg sibling(?s1,?s2,?a) and root(?s1,?a) and root(?s2,?a) and initial(?s1).

neg sibling(?s1,?s2,?a) :- root(?s1,?a) and root(?s2,?a) and neg ?s1! :=: successor(?a1,?s4) and
neg ?s2! :=: successor(?a2,?s4).
root(?s1,?a) :- neg sibling(?s1,?s2,?a) and root(?s2,?a) and neg ?s1! :=: successor(?a1,?s4) and
neg ?s2! :=: successor(?a2,?s4).
root(?s2,?a) :- neg sibling(?s1,?s2,?a) and root(?s1,?a) and neg ?s1! :=: successor(?a1,?s4) and
neg ?s2! :=: successor(?a2,?s4).
neg ?s1! :=: successor(?a1,?s4) :- neg sibling(?s1,?s2,?a) and root(?s1,?a) and root(?s2,?a) and
neg ?s2! :=: successor(?a2,?s4).
neg ?s2! :=: successor(?a2,?s4) :- neg sibling(?s1,?s2,?a) and root(?s1,?a) and root(?s2,?a) and
neg ?s1! :=: successor(?a1,?s4).
```

Appendix B: Axiomatization of the FLOWS Process Model

This is Appendix B of the Semantic Web Services Ontology (SWSO) document.

This appendix contains the axiomatization of the First-order Logic Ontology for Web Services (FLOWS) process model, expressed in SWSL-FOL. FLOWS is also known as SWSO-FOL. The hyperlinked concept and relation names refer to concepts and relations defined in the Process Specification Language (PSL).

Service

Every service is associated with an activity.

```
    forall ?s
            (service(?s) ==>
                    exists ?a
                            (service_activity(?s,?a))).
```

A service activity is an activity in the PSL Ontology.

```
    forall ?s,?a        (service_activity(?s,?a)  ==>                activity(?a)).
```

A service occurrence is an occurrence of the activity that is associated with the service.

```
    forall ?s,?o
            ((service_occurrence(?s,?o) and
            occurrence_of(?o,?a)) ==>
                    service_activity(?s,?a)).
```

AtomicProcess

An AtomicProcess is equivalent to a primitive activity in the PSL Ontology such that its preconditions and effects depend only on the fluents that hold prior to the an occurrence of the activity.

```
    forall ?a        (AtomicProcess(?a)  <==>                (primitive(?a)
    and                          markov_precond(?a)  and
    (markov_effects(?a)  or context_free(?a))).
```

If a fluent is an input to an activity, then the reference of the fluent must be known as a precondition for the activity to occur.

```
    forall ?s,?a,?iofluent
            ((occurrence_of(?s,?a) and
            legal(?s) and
            input(?a,?iofluent)) ==>
                    prior(Kref(?iofluent),?s)).
```

If ?iofluent is an output to an activity that is conditional on the fluent ?f, then the effect of the activity occurrence when ?f holds prior to the activity occurrence is that the reference of ?iofluent is known.

```
forall ?s,?a,?f,?iofluent
        ((occurrence_of(?s,?a) and
        legal(?s) and
        prior(?f,s) and
        output(?a,?f,?iofluent)) ==>
                holds(Kref(?iofluent),?s)).
```

The following axioms define the epistemic fluents.

The K fluent represents the accessibility relation in the possible-world model of knowledge. An activity occurrence ?s1 is accessible from an activity occurrence ?s if as far as is known in ?s, ?s1 might have occurred.

```
forall ?a,?s,?s2
        (occurrence_of(?s,?a) ==>
                (holds(K(?s2),?s) <==>
                        exists ?s1
                                ((?s2 :=: successor(?a,?s1)) and
                                legal(?s2) and
                                holds(K(?s1),?s) and
                                (SR(?a,?s) :=: SR(?a,?s1))))).
```

The Knows fluent holds if the K fluent holds at all accessible activity occurrences.

```
forall ?f,?s
        (holds(Knows(?f),?s) <==>
                forall ?sp
                        (holds(K(?sp),?s) ==> holds(?f,?sp))).
```

The fluent neg is used to reify negation for fluent terms.

```
forall ?f,?s
        (holds(neg(?f),?s) <==> (neg holds(?f,?s))).
```

The Kref fluent holds whenever the reference of its argument is known.

```
forall ?f,?s
        (holds(Kref(?f),?s) <==>
                exists ?x
                        (holds(Knows(?f :=: ?x),?s))).
```

composedOf

The composedOf relation is equivalent to the subactivity relation in the PSL Ontology.

```
forall ?a1,?a2        (composedOf(?a1,?a2) <==>                subactivity(?a2,?
a1)).
```

Split

An Split activity is equivalent to a strong_poset activity in PSL. The activity trees all have the same structure and each activity tree consists of branches that are in one-to-one correspondence with sequences of subactivity occurrences that satisfy the partial ordering constraints.

```
forall ?a        (Split(?a)  <==>                (uniform(?a)
and                         exists ?occ
(occurrence_of(?occ,?a)  and                                  neg simple(?
occ)  and                                      ordered(?occ)
and                                      strong_poset(?occ)))).
```

Sequence

An Sequence activity is equivalent to an activity in PSL whose activity trees all have the same structure and such that each activity tree consists of a unique branch.

```
forall ?a        (Sequence(?a)  <==>                (uniform(?a)
and                         exists ?occ
(occurrence_of(?occ,?a)  and                                  simple(?occ)
and                                      rigid(?occ)
and                                      ordered(?occ)
and                                      strong_poset(?occ)))).
```

Unordered

An Unordered activity is equivalent to an activity in PSL whose activity trees all have the same structure and such that each activity tree is a bag. In this case, there is a one-to-one correspondence between branches of the activity tree and all permutations of subactivity occurrences.

```
forall ?a        (Unordered(?a)  <==>                (uniform(?a)
and                         exists ?occ
(occurrence_of(?occ,?a) and                                  bag(?occ)))).
```

Choice

A Choice activity is equivalent to an activity in PSL whose activity trees all have the same structure and such that each activity tree is nondeterministic (that is, each branch contains occurrences of different subactivities.

```
forall ?a        (Choice(?a)  <==>                (uniform(?a)
and                         exists ?occ
(occurrence_of(?occ,?a)  and                                  simple(?occ)
and                                      rigid(?occ)
and                                      unordered(?occ)
and                                      choice_poset(?occ)))).
```

IfThenElse

An IfThenElse activity is equivalent to a conditional activity in the PSL Ontology.

```
forall ?a       (IfThenElse(?a)  <==>                conditional(?a)).
```

Iterate

An Iterate activity is equivalent to an activity in the PSL Ontology whose occurrences are repetitive. Activity trees for this activity have different structure, depending on the cardinality of the repeated subtrees.

```
forall ?a   (Iterate(?a)  <==>            activity(?a) and           (forall ?
occc                  (occurrence_of(?occ,?a)  ==>
(repetitive(?occ) and                                  multiple_outcome(?
occ)))).
```

RepeatUntil

A RepeatUntil activity is equivalent to a conditional activity in the PSL Ontology whose occurrences are repetitive. Activity trees for this activity have different structure, depending on the cardinality of the repeated subtrees.

```
forall ?a   (RepeatUntil(?a)  <==>            activity(?a) and
(conditional(?a)  and                forall ?occ
(occurrence_of(?occ,?a)  ==>                              (repetitive(?occ)
and                                          multiple_outcome(?occ)))).
```

OrderedActivity

An Ordered activity is equivalent to an activity in PSL whose activity trees all have the same structure and such that each activity tree is ordered if and only if the branches contain occurrences of the same subactivities.

```
forall ?a       (OrderedActivity(?a)  <==>                (uniform(?a)
and                        exists ?occ
(occurrence_of(?occ,?a)  and                              (ordered(?
occ)  <==>   (neg simple(?occ))))).
```

OccActivity

An OccActivity is equivalent to an activity in PSL whose activity trees all have the same structure and such that each activity tree consists of subtrees whose branches contain occurrences of the same subactivities.

```
forall ?a       (OccActivity(?a) <==>                (uniform(?a)
and                        exists ?occ
(occurrence_of(?occ,?a)  and                            (permuted(?occ)  or
(nondet_permuted(?occ))))).
```

TriggeredActivity

A TriggeredActivity is equivalent to a trigger activity in the PSL Ontology.

```
forall ?a       (TriggeredActivity(?a)  <==>                trigger(?a)).
```

Message

Messages are objects in the ontology.

```
    forall ?m,?o,?msgtype
            (message_info(?m,?msgtype,?x) ==> object(?m)).
```

A message is produced,read, or destroyed by an AtomicProcess.

```
    forall ?o,?m,?a
            ((produces(?o,?m) or reads(?o,?m) or destroys(?o,?m)) and
            occurrence_of(?o,?a) ==>
                    AtomicProcess(?a)).
```

For any occurrence that reads a message, the the reference of the input described by the message type is known after the occurrence.

```
    forall ?o,?m,?msgtype,?x,?iofluent
            ((reads(?o,?m) and
            message_info(?m,?msgtype,?x) and
            described_by(?msgtype,?iofluent) and
            legal(?o)) ==>
                holds(Kref(?iofluent),?o)).
```

For any occurrence that produces a message, the reference of the output described by the message is known before the occurrence.

```
    forall ?o,?m,?msgtype,?x,?iofluent
            ((produces(?o,?m) and
            message_info(?m,?msgtype,?x) and
            described_by(?msgtype,?iofluent) and
            legal(?o)) ==>
                prior(Kref(?iofluent),?o)).
```

Every activity occurrence that reads a message is preceded by an activity occurrence that produces the message.

```
    forall ?o1,?m
            (reads(?o1,?m) ==>
            exists ?o2
                    (produces(?o2,m) and
                    precedes(?o2,o1))).
```

A message object is created by an activity occurrence that produces a message.

```
    forall ?o,?m
            (produces(?o,?m) ==>
                ((beginof(?m) :=: endof(?o)) and
                holds(existing_mobject(?m),?o))).
```

A message object is destroyed by an activity occurrence that destroys a message.

```
    forall ?o,?m
```

```
                (destroys(?o,?m) ==>
                    ((endof(?m) :=: endof(?o)) and
                    (neg holds(existing_mobject(?m),?o)))).
```

A ProduceMessage activity is an activity whose occurrences produce messages.

```
        forall ?a
                (ProduceMessage(?a) <==>
                    (activity(?a) and
                        (forall ?o
                                (occurrence_of(?o,?a) ==>
                                        exists ?m
                                                (produces(?o,?m))))).
```

A ReadMessage activity is an activity whose occurrences read messages.

```
        forall ?a
                (ReadMessage(?a) <==>
                    (activity(?a) and
                        (forall ?o
                                (occurrence_of(?o,?a) ==>
                                        exists ?m
                                                (reads(?o,?m))))).
```

A DestroyMessage activity is an activity whose occurrences destroy messages.

```
        forall ?a
                (DestroyMessage(?a) <==>
                    (activity(?a) and
                        (forall ?o
                                (occurrence_of(?o,?a) ==>
                                        exists ?m
                                                (destroys(?o,?m))))).
```

Channel

If a message is contained in a channel, then it is produced by an occurrence of a service that is a source for the channel.

```
        forall ?c,?m
                (channel_mobject(?c,?m) ==>
                        exists ?s,?o,?o1
                                (channel_source(?c,?s) and
                                occurrence_of(?o,?s) and
                                produces(?o1,?m) and
                                subactivity_occurrence(?o1,?o))).
```

If a message is contained in a channel, then it is read by an occurrence of a service that is a target for the channel.

```
        forall ?c,?m
                (channel_mobject(?c,?m) ==>
                        exists ?s,?o,?o1
                                (channel_target(?c,?s) and
                                occurrence_of(?o,?s) and
                                reads(?o1,?m) and
                                subactivity_occurrence(?o1,?o))).
```

Exceptions

Exceptions form a sublcass of fluents.

```
    forall ?e
            (exception(e) ==> fluent(e)).
```

The following axioms capture the relationships in Figure 2.2.

An exception ?e is handled by an activity ?a.

```
    forall ?a,?e
            (is_handled_by(?e,?a) ==>
                    (activity(?a) and exception(?e))).
```

An activity ?a has an exception ?e.

```
    forall ?a,?e
            (has_exception(?a,?e) ==>
                    (activity(?a) and exception(?e))).
```

An activity ?a is an exception-handling activity if and only if it is a TriggeredActivity that there exists an exception that is handled by ?a.

```
    forall ?a
            (handle_exception(?a) <==>
                    exists ?e (is_handled_by(?e,?a) and TriggeredActivity(?a))).
```

find_exception and fix_exception are the two subclasses of exception-handling activities.

```
    forall ?a
            (handle_exception(?a) <==>
                    (find_exception(?a) or fix_exception(?a))).
```

detect_exception and anticipate_exception are the two subclasses of exception-finding activities.

```
    forall ?a
            (find_exception(?a) <==>
                    (detect_exception(?a) or anticipate_exception(?a))).
```

avoid_exception and resolve_exception are the two subclasses of exception-fixing activities.

```
    forall ?a
            (fix_exception(?a) <==>
                    (avoid_exception(?a) or resolve_exception(?a))).
```

The relation is_found_by is the restriction of the is_handled_by relation to exception-finding activities. The relation is_fixed_by is the restriction of the is_handled_by relation to exception-fixing activities.

```
    forall ?a,?e
            (is_handled_by(?e,?a) <==>
                    ((is_found_by(?e,?a) and find_exception(?a)) or
                    (is_fixed_by(?e,?a) and fix_exception(?a)))).
```

The relation is_detected_by is the restriction of the is_found_by relation to exception-detecting activities. The relation is_anticipated_by is the restriction of the is_found_by relation to exception-anticipating activities.

```
    forall ?a,?e
            (is_found_by(?e,?a) <==>
                    ((is_detected_by(?e,?a) and detect_exception(?a)) or
                    (is_anticipated_by(?e,?a) and anticipate_exception(?a)))).
```

The relation is_avoided_by is the restriction of the is_fixed_by relation to exception-avoiding activities. The relation is_resolved_by is the restriction of the is_fixed_by relation to exception-resolving activities.

```
    forall ?a,?e
            (is_fixed_by(?e,?a) <==>
                    ((is_avoided_by(?e,?a) and avoid_exception(?a)) or
                    (is_resolved_by(?e,?a) and resolve_exception(?a)))).
```

Appendix C: Axiomatization of the Process Model in SWSL-Rules

This is Appendix C of the Semantic Web Services Ontology (SWSO) document.

Here we give the Rules Ontology for Web Services (ROWS) formulations of the axioms of the process model, as described in Section 3 of the SWSO document. ROWS is also known as SWSO-Rules. These formulations are derived directly from the FLOWS (SWSO-FOL) formulations, as given in Appendix B, according to the translation approach described in Section 3 of the SWSL document. They can easily be compared to the FLOWS axioms by viewing Appendices B and C in two browser windows side by side.

The hyperlinked concept and relation names refer to concepts and relations defined in the Process Specification Language (PSL).

Service

Every service is associated with an activity.

```
service_activity(?s, _#(?s)) :- service(?s).
neg service(?s) :- neg service_activity(?s, _#(?s)).

Note: The two rules above are the omnidirectional set of rules for
this (skolemized) clause:
  neg service(?s) or service_activity(?s, _#(?s)).
which is equivalent to this:
  service(?s) ==> service_activity(?s, _#(?s)).
```

A service activity is an activity in the PSL Ontology.

```
activity(?a) :- service_activity(?s,?a).
neg service_activity(?s,?a) :- neg activity(?a).
```

A service occurrence is an occurrence of the activity that is associated with the service.

```
neg service_occurrence(?s,?o) :- occurrence_of(?o,?a) and neg service_activity(?
s,?a).
neg occurrence_of(?o,?a) :- service_occurrence(?s,?o) and neg service_activity(?
s,?a).
service_activity(?s,?a) :- service_occurrence(?s,?o) and occurrence_of(?o,?a).

Note: The three rules above are the omnidirectional set of rules for
this clause:
  neg service_occurrence(?s,?o) or neg occurrence_of(?o,?a) or service_activity
(?s,?a).
which is equivalent to this:
  service_occurrence(?s,?o) and occurrence_of(?o,?a) ==> service_activity(?s,?
a).
```

AtomicProcess

An AtomicProcess is equivalent to a primitive activity in the PSL Ontology such that its preconditions and effects depend only on the fluents that hold prior to the an occurrence of the activity.

```
AtomicProcess(?a) :- primitive(?a) and markov_precond(?a) and markov_effects(?
a).
neg primitive(?a) :- neg AtomicProcess(?a) and markov_precond(?a) and
markov_effects(?a).
neg markov_precond(?a) :- neg AtomicProcess(?a) and primitive(?a) and
markov_effects(?a).
neg markov_effects(?a) :- neg AtomicProcess(?a) and primitive(?a) and
markov_precond(?a).

AtomicProcess(?a) :- primitive(?a) and markov_precond(?a) and context_free(?a).
neg primitive(?a) :- neg AtomicProcess(?a) and markov_precond(?a) and
context_free(?a).
neg markov_precond(?a) :- neg AtomicProcess(?a) and primitive(?a) and
context_free(?a).
neg context_free(?a) :- neg AtomicProcess(?a) and primitive(?a) and
markov_precond(?a).

primitive(?a) :- AtomicProcess(?a).
neg AtomicProcess(?a) :- neg primitive(?a).

markov_precond(?a) :- AtomicProcess(?a).
neg AtomicProcess(?a) :- neg markov_precond(?a).

neg AtomicProcess(?a) :- neg markov_effects(?a) and neg context_free(?a).
markov_effects(?a) :- neg context_free(?a) and AtomicProcess(?a).
context_free(?a) :- AtomicProcess(?a) and neg markov_effects(?a).

Note: The first 8 rules above are the omnidirectional sets of
rules for these two disjunctions:
  AtomicProcess(?a) or not primitive(?a) or not markov_precond(?a) or neg
markov_effects(?a).
  AtomicProcess(?a) or not primitive(?a) or not markov_precond(?a) or neg
context_free(?a).
which (conjoined) are equivalent to the impliedBy (<==) direction of the
SWSO-FOL axiom:
  forall ?a
        (AtomicProcess(?a)  <==
                (primitive(?a) and
                        markov_precond(?a)  and
                        (markov_effects(?a)  or context_free(?a))).

The last 7 rules above are the omnidirectional sets for these clauses:
  neg AtomicProcess(?a)  or primitive(?a)
  neg AtomicProcess(?a)  or markov_precond(?a)
  neg AtomicProcess(?a)  or markov_effects(?a)  or context_free(?a)
which together are equivalent to the implies
  (==>) direction of the SWSO-FOL axiom.
```

If a fluent is an input to an activity, then the reference of the fluent must be known as a precondition for the activity *to occur.*

```
prior(Kref(?iofluent),?s)) :-
  occurrence_of(?s,?a) and legal(?s) and input(?a,?iofluent).
neg occurrence_of(?s,?a) :-
  neg prior(Kref(?iofluent),?s)) and legal(?s) and input(?a,?iofluent).
neg legal(?s) :-
  neg prior(Kref(?iofluent),?s)) and occurrence_of(?s,?a) and input(?a,?
iofluent).
neg input(?a,?iofluent) :-
  neg prior(Kref(?iofluent),?s)) and occurrence_of(?s,?a) and legal(?s).
```

If ?iofluent is an output to an activity that is conditional on the fluent ?f, then the effect of the activity occurrence when ?f holds prior to the activity occurrence is that the reference of ?iofluent is known.

```
holds(Kref(?iofluent),?s)) :-
  occurrence_of(?s,?a) and legal(?s) and prior(?f,s) and output(?a,?iofluent,?
f).
neg occurrence_of(?s,?a) :-
  neg holds(Kref(?iofluent),?s)) and legal(?s) and prior(?f,s) output(?a,?
iofluent,?f).
neg legal(?s) :-
  neg holds(Kref(?iofluent),?s)) and occurrence_of(?s,?a) and prior(?f,s) and
output(?a,?iofluent,?f).
neg prior(?f,s) :-
  neg holds(Kref(?iofluent),?s)) and occurrence_of(?s,?a) and legal(?s) and
output(?a,?iofluent,?f).
neg output(?a,?iofluent,?f) :-
  neg holds(Kref(?iofluent),?s)) and occurrence_of(?s,?a) and legal(?s) and
prior(?f,s).
```

The following axioms define the epistemic fluents.

The K fluent represents the accessibility relation in the possible-world model of knowledge. An activity occurrence ?s1 is accessible from an activity occurrence ?s if as far as is known in ?s, ?s1 might have occurred.

```
neg occurrence_of(?s,?a) :- holds(K(?s2),?s) and ?s2 :=: successor(?a,_#1(?a,?
s,?s2)).
neg holds(K(?s2),?s) :- occurrence_of(?s,?a) and neg ?s2 :=: successor(?a,_#1(?
a,?s,?s2)).
?s2 :=: successor(?a,_#1(?a,?s,?s2)) :- occurrence_of(?s,?a) and holds(K(?s2),?
s).

neg occurrence_of(?s,?a) :- holds(K(?s2),?s) and neg legal(?s2).
neg holds(K(?s2),?s) :- occurrence_of(?s,?a) and neg legal(?s2).
legal(?s2) :- occurrence_of(?s,?a) and holds(K(?s2),?s).

neg occurrence_of(?s,?a) :- holds(K(?s2),?s) and neg holds(K(_#1(?a,?s,?s2)),?
s).
neg holds(K(?s2),?s) :- occurrence_of(?s,?a) and neg holds(K(_#1(?a,?s,?s2)),?
s).
holds(K(_#1(?a,?s,?s2)),?s) :- occurrence_of(?s,?a) and holds(K(?s2),?s).

neg occurrence_of(?s,?a) :- holds(K(?s2),?s) and neg SR(?a,?s) :=: SR(?a,_#1(?
a,?s,?s2)).
neg holds(K(?s2),?s) :- occurrence_of(?s,?a) and neg SR(?a,?s) :=: SR(?a,_#1(?
a,?s,?s2)).
SR(?a,?s) :=: SR(?a,_#1(?a,?s,?s2)) :- occurrence_of(?s,?a) and holds(K(?s2),?
s).

neg occurrence_of(?s,?a) :-
  neg holds(K(?s2),?s) and (?s2 :=: successor(?a,?s1) and
  legal(?s2) and holds(K(?s1),?s) and (SR(?a,?s) :=: SR(?a,?s1)).
neg holds(K(?s2),?s) :-
  occurrence_of(?s,?a) and (?s2 :=: successor(?a,?s1) and
  legal(?s2) and holds(K(?s1),?s) and (SR(?a,?s) :=: SR(?a,?s1)).
neg (?s2 :=: successor(?a,?s1) :-
  occurrence_of(?s,?a) and neg holds(K(?s2),?s) and
  legal(?s2) and holds(K(?s1),?s) and (SR(?a,?s) :=: SR(?a,?s1)).
neg legal(?s2) :-
  occurrence_of(?s,?a) and neg holds(K(?s2),?s) and
  (?s2 :=: successor(?a,?s1) and holds(K(?s1),?s) and (SR(?a,?s) :=: SR(?a,?
s1)).
neg holds(K(?s1),?s) :-
  occurrence_of(?s,?a) and neg holds(K(?s2),?s) and
  (?s2 :=: successor(?a,?s1) and legal(?s2) and (SR(?a,?s) :=: SR(?a,?s1)).
neg (SR(?a,?s) :=: SR(?a,?s1)) :-
  occurrence_of(?s,?a) and neg holds(K(?s2),?s) and
  (?s2 :=: successor(?a,?s1) and legal(?s2) and holds(K(?s1),?s).
```

The Knows fluent holds if the K fluent holds at all accessible activity occurrences.

```
neg holds(Knows(?f),?s) :- holds(K(?sp),?s) and neg holds(?f,?sp).
neg holds(K(?sp),?s) :- holds(Knows(?f),?s) and neg holds(?f,?sp).
holds(?f,?sp) :- holds(Knows(?f),?s) and holds(K(?sp),?s).

holds(Knows(?f),?s) :- neg holds(K(_#1(?f,?s)),?s) and holds(?f,_#1(?f,?s)).
holds(K(_#1(?f,?s)),?s) :- neg holds(Knows(?f),?s) and holds(?f,_#1(?f,?s)).
neg holds(?f,_#1(?f,?s)) :- neg holds(Knows(?f),?s) and neg holds(K(_#1(?f,?
s)),?s).
```

The fluent neg is used to reify negation for fluent terms.

```
holds(neg(?f), ?s) :- neg holds(?f, ?s).
holds(?f, ?s) :- neg holds(neg(?f), ?s).

neg holds(?f, ?s) :- holds(neg(?f), ?s).
neg holds(neg(?f), ?s) :- holds(?f, ?s).
```

The Kref fluent holds whenever the reference of its argument is known.

```
holds(Kref(?f), ?s) :-
  holds(Knows(?f :=: ?x), ?s).
neg (holds(Knows(?f :=: ?x),?s)) :-
  neg holds(Kref(?f),?s).

holds(Knows(?f :=: _#(?f,?s)), ?s) :-
  holds(Kref(?f), ?s).
neg holds(Kref(?f), ?s) :-
  neg holds(Knows(?f :=: _#(?f,?s)), ?s).
```

composedOf

The composedOf relation is equivalent to the subactivity relation in the PSL Ontology.

```
composedOf(?a1,?a2) :- subactivity(?a2,?a1).
neg subactivity(?a2,?a1) :- neg composedOf(?a1,?a2).

subactivity(?a2,?a1) :- composedOf(?a1,?a2).
neg composedOf(?a1,?a2) :- neg subactivity(?a2,?a1).
```

Split

A Split activity is equivalent to a strong_poset activity in PSL. The activity trees all have the same structure and each activity tree consists of branches that are in one-to-one correspondence with sequences of subactivity occurrences that satisfy the partial ordering constraints.

```
Split(?a) :-
  uniform(?a) and occurrence_of(?occ,?a) and neg simple(?occ) and
  ordered(?occ) and strong_poset(?occ).
neg uniform(?a) :-
  neg Split(?a) and occurrence_of(?occ,?a) and neg simple(?occ) and
  ordered(?occ) and strong_poset(?occ).
neg occurrence_of(?occ,?a) :-
  neg Split(?a) and uniform(?a) and neg simple(?occ) and
  ordered(?occ) and strong_poset(?occ).
simple(?occ) :-
  neg Split(?a) and uniform(?a) and occurrence_of(?occ,?a) and
  ordered(?occ) and strong_poset(?occ).
neg ordered(?occ) :-
  neg Split(?a) and uniform(?a) and occurrence_of(?occ,?a) and
  neg simple(?occ) and strong_poset(?occ).
neg strong_poset(?occ) :-
  neg Split(?a) and uniform(?a) and occurrence_of(?occ,?a) and
  neg simple(?occ) and ordered(?occ).

uniform(?a) and occurrence_of(_#1(?a),?a) and neg simple(_#1(?a)) and
ordered(_#1(?a)) and strong_poset(_#1(?a)) :-
  Split(?a).

neg Split(?a) :- neg uniform(?a).
neg Split(?a) :- neg occurrence_of(_#1(?a),?a).
neg Split(?a) :- simple(_#1(?a)).
neg Split(?a) :- neg ordered(_#1(?a)).
neg Split(?a) :- neg strong_poset(_#1(?a)).
```

Sequence

A Sequence activity is equivalent to an activity in PSL whose activity trees all have the same structure and such that each activity tree consists of a unique branch.

```
Sequence(?a) :-
  uniform(?a) and occurrence_of(?occ,?a) and simple(?occ) and
  rigid(?occ) and ordered(?occ) and strong_poset(?occ).
neg uniform(?a) :-
  neg Sequence(?a) and occurrence_of(?occ,?a) and simple(?occ)
  and rigid(?occ) and ordered(?occ) and strong_poset(?occ).
neg occurrence_of(?occ,?a) :-
  neg Sequence(?a) and uniform(?a) and simple(?occ)
  and rigid(?occ) and ordered(?occ) and strong_poset(?occ).
neg simple(?occ) :-
  neg Sequence(?a) and uniform(?a) and occurrence_of(?occ,?a) and
   rigid(?occ) and ordered(?occ) and strong_poset(?occ).
neg rigid(?occ) :-
  neg Sequence(?a) and uniform(?a) and occurrence_of(?occ,?a) and
  simple(?occ) and ordered(?occ) and strong_poset(?occ).
neg ordered(?occ) :-
  neg Sequence(?a) and uniform(?a) and occurrence_of(?occ,?a) and
  simple(?occ) and rigid(?occ) and strong_poset(?occ).
neg strong_poset(?occ) :-
  neg Sequence(?a) and uniform(?a) and occurrence_of(?occ,?a) and
  simple(?occ) and rigid(?occ) and ordered(?occ).

uniform(?a) and occurrence_of(_#1(?a),?a) and simple(_#1(?a)) and
rigid(_#1(?a)) and ordered(_#1(?a)) and strong_poset(_#1(?a)) :-
  Sequence(?a).

neg Sequence(?a) :- neg uniform(?a).
neg Sequence(?a) :- neg occurrence_of(_#1(?a),?a).
neg Sequence(?a) :- neg simple(_#1(?a)).
neg Sequence(?a) :- neg rigid(_#1(?a)).
neg Sequence(?a) :- neg ordered(_#1(?a)).
neg Sequence(?a) :- neg strong_poset(_#1(?a)).
```

Unordered

An Unordered activity is equivalent to an activity in PSL whose activity trees all have the same structure and such that each activity tree is a bag. In this case, there is a one-to-one correspondence between branches of the activity tree and all permutations of subactivity occurrences.

```
Unordered(?a) :-
  uniform(?a) and occurrence_of(?occ,?a) and bag(?occ).
neg uniform(?a) :-
  Unordered(?a) and occurrence_of(?occ,?a) and bag(?occ).
neg occurrence_of(?occ,?a) :-
  neg Unordered(?a) and uniform(?a) and bag(?occ).
neg bag(?occ) :-
  neg Unordered(?a) and uniform(?a) and occurrence_of(?occ,?a).

uniform(?a) and occurrence_of(_#1(?a),?a) and bag(_#1(?a)) :-
  Unordered(?a).
```

```
neg Unordered(?a) :- neg uniform(?a).
neg Unordered(?a) :- neg occurrence_of(_#1(?a),?a).
neg Unordered(?a) :- neg bag(_#1(?a)).
```

Choice

A Choice activity is equivalent to an activity in PSL whose activity trees all have the same structure and such that each activity tree is nondeterministic (that is, each branch contains occurrences of different subactivities.

```
Choice(?a) :-
  uniform(?a) and occurrence_of(?occ,?a) and simple(?occ) and
  rigid(?occ) and unordered(?occ) and choice_poset(?occ).
neg uniform(?a) :-
  neg Choice(?a) and occurrence_of(?occ,?a) and simple(?occ) and
  rigid(?occ) and unordered(?occ) and choice_poset(?occ).
neg occurrence_of(?occ,?a) :-
  neg Choice(?a) and uniform(?a) and simple(?occ) and
  rigid(?occ) and unordered(?occ) and choice_poset(?occ).
neg simple(?occ) :-
  neg Choice(?a) and uniform(?a) and occurrence_of(?occ,?a) and
  rigid(?occ) and unordered(?occ) and choice_poset(?occ).
neg rigid(?occ) :-
  neg Choice(?a) and uniform(?a) and occurrence_of(?occ,?a) and
  simple(?occ) and unordered(?occ) and choice_poset(?occ).
neg unordered(?occ) :-
  neg Choice(?a) and uniform(?a) and occurrence_of(?occ,?a) and
  simple(?occ) and rigid(?occ) and choice_poset(?occ).
neg choice_poset(?occ) :-
  neg Choice(?a) and uniform(?a) and occurrence_of(?occ,?a) and
  simple(?occ) and rigid(?occ) and unordered(?occ).

uniform(?a) and occurrence_of(_#1(?a),?a) and neg simple(_#1(?a)) and
rigid(_#1(?a)) and unordered(_#1(?a)) and choice_poset(_#1(?a)) :-
  Choice(?a).

neg Choice(?a) :- neg uniform(?a).
neg Choice(?a) :- neg occurrence_of(_#1(?a),?a).
neg Choice(?a) :- simple(_#1(?a)).
neg Choice(?a) :- neg rigid(_#1(?a)).
neg Choice(?a) :- neg unordered(_#1(?a)).
neg Choice(?a) :- neg choice_poset(_#1(?a)).
```

IfThenElse

An IfThenElse activity is equivalent to a conditional activity in the PSL Ontology.

```
IfThenElse(?a) :- conditional(?a).
neg conditional(?a) :- neg IfThenElse(?a).

conditional(?a) :- IfThenElse(?a).
neg IfThenElse(?a) :- neg conditional(?a).
```

Iterate

An Iterate activity is equivalent to an activity in the PSL Ontology whose occurrences are repetitive. Activity trees for this activity have different structure, depending on the cardinality of the repeated subtrees.

```
Iterate(?a) :- activity(?a) and neg occurrence_of(_#1(?a),?a).
neg activity(?a) :- neg Iterate(?a) and neg occurrence_of(_#1(?a),?a).
occurrence_of(_#1(?a),?a) :- neg Iterate(?a) and activity(?a).

Iterate(?a) :- activity(?a) and repetitive(_#1(?a)) and multiple_outcome(_#1(?
a)).
neg activity(?a) :- neg Iterate(?a) and repetitive(_#1(?a)) and multiple_outcome
(_#1(?a)).
neg repetitive(_#1(?a)) :- neg Iterate(?a) and activity(?a) and multiple_outcome
(_#1(?a)).
neg multiple_outcome(_#1(?a)) :- neg Iterate(?a) and activity(?a) and repetitive
(_#1(?a)).

neg Iterate(?a) :- neg activity(?a).
activity(?a) :- Iterate(?a).

neg Iterate(?a) :- occurrence_of(?occ,?a) and neg repetitive(?occ).
neg occurrence_of(?occ,?a) :- Iterate(?a) and neg repetitive(?occ).
repetitive(?occ) :- Iterate(?a) and occurrence_of(?occ,?a).

neg Iterate(?a) :- occurrence_of(?occ,?a) and neg multiple_outcome(?occ).
neg occurrence_of(?occ,?a) :- Iterate(?a) and neg multiple_outcome(?occ).
multiple_outcome(?occ) :- Iterate(?a) and  occurrence_of(?occ,?a).
```

RepeatUntil

A RepeatUntil activity is equivalent to a conditional activity in the PSL Ontology whose occurrences are repetitive. Activity trees for this activity have different structure, depending on the cardinality of the repeated subtrees.

```
RepeatUntil(?a) :- conditional(?a) and neg occurrence_of(_#1(?a),?a).
neg conditional(?a) :- neg RepeatUntil(?a) and neg occurrence_of(_#1(?a),?a).
occurrence_of(_#1(?a),?a) :- neg RepeatUntil(?a) and conditional(?a).

RepeatUntil(?a) :- conditional(?a) and repetitive(_#1(?a)) and multiple_outcome
(_#1(?a).
neg conditional(?a) :- neg RepeatUntil(?a) and repetitive(_#1(?a)) and
multiple_outcome(_#1(?a)).
neg repetitive(_#1(?a)) :- neg RepeatUntil(?a) and conditional(?a) and
multiple_outcome(_#1(?a)).
neg multiple_outcome(_#1(?a)) :- neg RepeatUntil(?a) and conditional(?a) and
repetitive(_#1(?a)).

neg RepeatUntil(?a) :- neg activity(?a).
activity(?a) :- RepeatUntil(?a).

neg RepeatUntil(?a) :- neg conditional(?a).
conditional(?a) :- RepeatUntil(?a).
```

```
neg RepeatUntil(?a) :- occurrence_of(?occ,?a) and neg repetitive(?occ).
neg occurrence_of(?occ,?a) :- RepeatUntil(?a) and neg repetitive(?occ).
repetitive(?occ) :- RepeatUntil(?a) and occurrence_of(?occ,?a).

neg RepeatUntil(?a) :- occurrence_of(?occ,?a) and neg multiple_outcome(?occ).
neg occurrence_of(?occ,?a) :- RepeatUntil(?a) and neg multiple_outcome(?occ).
multiple_outcome(?occ) :- RepeatUntil(?a) and occurrence_of(?occ,?a).
```

OrderedActivity

An Ordered activity is equivalent to an activity in PSL whose activity trees all have the same structure and such that each activity tree is ordered if and only if the branches contain occurrences of the same subactivities.

```
OrderedActivity(?a) :- uniform(?a) and occurrence_of(?occ,?a) and neg simple(?
occ) and ordered(?occ).
neg uniform(?a) :- neg OrderedActivity(?a) and occurrence_of(?occ,?a) and neg
simple(?occ) and ordered(?occ).
neg occurrence_of(?occ,?a) :- neg OrderedActivity(?a) and uniform(?a) and neg
simple(?occ) and ordered(?occ).
simple(?occ) :- neg OrderedActivity(?a) and uniform(?a) and occurrence_of(?occ,?
a) and ordered(?occ).
neg ordered(?occ) :- neg OrderedActivity(?a) and uniform(?a) and occurrence_of(?
occ,?a) and neg simple(?occ).

OrderedActivity(?a) :- uniform(?a) and occurrence_of(?occ,?a) and simple(?occ)
and neg ordered(?occ).
neg uniform(?a) :- neg OrderedActivity(?a) and occurrence_of(?occ,?a) and simple
(?occ) and neg ordered(?occ).
neg occurrence_of(?occ,?a) :- neg OrderedActivity(?a) and uniform(?a) and simple
(?occ) and neg ordered(?occ).
neg simple(?occ) :- neg OrderedActivity(?a) and uniform(?a) and occurrence_of(?
occ,?a) and neg ordered(?occ).
ordered(?occ) :- neg OrderedActivity(?a) and uniform(?a) and occurrence_of(?
occ,?a) and simple(?occ).

neg OrderedActivity(?a) :- neg uniform(?a).
uniform(?a) :- OrderedActivity(?a).

neg OrderedActivity(?a) :- neg occurrence_of(_#1(?a),?a).
occurrence_of(_#1(?a),?a) :- OrderedActivity(?a).

neg OrderedActivity(?a) :- neg ordered(_#1(?a)) and neg simple(_#1(?a)).
ordered(_#1(?a)) :- OrderedActivity(?a) and neg simple(_#1(?a)).
simple(_#1(?a)) :- OrderedActivity(?a) and neg ordered(_#1(?a)).

neg OrderedActivity(?a) :- ordered(_#1(?a)) and simple(_#1(?a)).
neg ordered(_#1(?a)) :- OrderedActivity(?a) and simple(_#1(?a)).
neg simple(_#1(?a)) :- OrderedActivity(?a) and ordered(_#1(?a)).
```

OccActivity

An Ordered activity is equivalent to an activity in PSL whose activity trees all have the same structure and such that each activity tree is ordered if and only if the branches contain occurrences of the same subactivities.

```
OccActivity(?a) :- uniform(?a) and occurrence_of(?occ,?a) and permuted(?occ).
neg uniform(?a) :- neg OccActivity(?a) and occurrence_of(?occ,?a) and permuted(?
occ).
neg occurrence_of(?occ,?a) :- neg OccActivity(?a) and uniform(?a) and permuted(?
occ).
neg permuted(?occ) :- neg OccActivity(?a) and uniform(?a) and occurrence_of(?
occ,?a).

OccActivity(?a) :- uniform(?a) and occurrence_of(?occ,?a) and neg
nondet_permuted(?occ).
neg uniform(?a) :- neg OccActivity(?a) and occurrence_of(?occ,?a) and neg
nondet_permuted(?occ).
neg occurrence_of(?occ,?a) :- neg OccActivity(?a) and uniform(?a) and neg
nondet_permuted(?occ).
nondet_permuted(?occ) :- neg OccActivity(?a) and uniform(?a) and occurrence_of(?
occ,?a).

neg OccActivity(?a) :- neg uniform(?a).
uniform(?a) :- OccActivity(?a).

neg OccActivity(?a) :- neg occurrence_of(_#1(?a),?a).
occurrence_of(_#1(?a),?a) :- OccActivity(?a).

neg OccActivity(?a) :- neg permuted(_#1(?a)) and neg nondet_permuted(_#1(?a)).
permuted(_#1(?a)) :- OccActivity(?a) and neg nondet_permuted(_#1(?a)).
nondet_permuted(_#1(?a)) :- OccActivity(?a) and neg permuted(_#1(?a)).
```

TriggeredActivity

A TriggeredActivity is equivalent to a trigger activity in the PSL Ontology.

```
TriggeredActivity(?a) :- trigger(?a).
neg trigger(?a) :- neg TriggeredActivity(?a).

trigger(?a) :- TriggeredActivity(?a).
neg TriggeredActivity(?a) :- neg trigger(?a).
```

Message

Messages are objects in the ontology.

```
object(?m) :- message_info(?m,?msgtype,?x).
neg message_info(?m,?msgtype,?x) :- neg object(?m).
```

A message is produced, read, or destroyed by an AtomicProcess.

```
neg produces(?o,?m) :- occurrence_of(?o,?a) and neg AtomicProcess(?a).
neg occurrence_of(?o,?a) :- produces(?o,?m) and neg AtomicProcess(?a).
AtomicProcess(?a) :- produces(?o,?m) and occurrence_of(?o,?a).

neg reads(?o,?m) :- occurrence_of(?o,?a) and neg AtomicProcess(?a).
neg occurrence_of(?o,?a) :- reads(?o,?m) and neg AtomicProcess(?a).
AtomicProcess(?a) :- reads(?o,?m) and occurrence_of(?o,?a).

neg destroys(?o,?m) :- occurrence_of(?o,?a) and neg AtomicProcess(?a).
neg occurrence_of(?o,?a) :- destroys(?o,?m) and neg AtomicProcess(?a).
AtomicProcess(?a) :- destroys(?o,?m) and occurrence_of(?o,?a).
```

For any occurrence that reads a message, the reference of the input described by the message type is known after the occurrence.

```
holds(Kref(?iofluent),?o)) :-
  reads(?o,?m) and message_info(?m,?msgtype,?x) and
  described_by(?msgtype,?iofluent) and legal(?o).
neg reads(?o,?m) :-
  message_info(?m,?msgtype,?x) and described_by(?msgtype,?iofluent) and
  legal(?o) and neg holds(Kref(?iofluent),?o)).
neg message_info(?m,?msgtype,?x) :-
  reads(?o,?m) and described_by(?msgtype,?iofluent) and legal(?o) and
  neg holds(Kref(?iofluent),?o)).
neg described_by(?msgtype,?iofluent) :-
  reads(?o,?m) and message_info(?m,?msgtype,?x) and
  legal(?o) and neg holds(Kref(?iofluent),?o)).
neg legal(?o) :-
  reads(?o,?m) and message_info(?m,?msgtype,?x) and
  described_by(?msgtype,?iofluent) and neg holds(Kref(?iofluent),?o)).
```

For any occurrence that produces a message, the reference of the output described by the message is known before the occurrence.

```
prior(Kref(?iofluent),?o)) :-
  produces(?o,?m) and  message_info(?m,?msgtype,?x) and
  described_by(?msgtype,?iofluent) and legal(?o).
neg produces(?o,?m) :-
  message_info(?m,?msgtype,?x) and described_by(?msgtype,?iofluent) and
  legal(?o) and neg prior(Kref(?iofluent),?o)).
neg message_info(?m,?msgtype,?x) :-
   produces(?o,?m) and described_by(?msgtype,?iofluent) and
   legal(?o) and neg prior(Kref(?iofluent),?o)).
neg described_by(?msgtype,?iofluent) :-
  produces(?o,?m) and message_info(?m,?msgtype,?x) and
  legal(?o) and neg prior(Kref(?iofluent),?o)).
neg legal(?o) :-
  produces(?o,?m) and message_info(?m,?msgtype,?x) and
  described_by(?msgtype,?iofluent) and neg prior(Kref(?iofluent),?o)).
```

Every activity occurrence that reads a message is preceded by an activity occurrence that produces the message.

```
produces(?o2,m) :- reads(?o1,?m).
neg reads(?o1,?m) :- neg produces(?o2,m).
precedes(?o2,o1)) :- reads(?o1,?m).
neg reads(?o1,?m) :- neg precedes(?o2,o1)).
```

A message object is created by an activity occurrence that produces a message.

```
beginof(?m) :=: endof(?o) :- produces(?o,?m).
neg produces(?o,?m) :- neg beginof(?m) :=: endof(?o).
holds(existing_mobject(?m),?o) :- produces(?o,?m).
neg produces(?o,?m) :- neg holds(existing_mobject(?m),?o).
```

A message object is destroyed by an activity occurrence that destroys a message.

```
beginof(?m) :=: endof(?o) :- destroys(?o,?m).
neg destroys(?o,?m) :- neg beginof(?m) :=: endof(?o).
neg holds(existing_mobject(?m),?o) :- destroys(?o,?m).
neg destroys(?o,?m) :- holds(existing_mobject(?m),?o).
```

A ProduceMessage activity is an activity whose occurrences produce messages.

```
ProduceMessage(?a) :- activity(?a) and neg occurrence_of(_#1(?a),?a).
neg activity(?a) :- neg ProduceMessage(?a) and neg occurrence_of(_#1(?a),?a).
occurrence_of(_#1(?a),?a) :- neg ProduceMessage(?a) and activity(?a).

ProduceMessage(?a) :- activity(?a) and produces(_#1(?a),?m).
neg activity(?a) :- produces(_#1(?a),?m) and ProduceMessage(?a).
neg produces(_#1(?a),?m) :- neg ProduceMessage(?a) and activity(?a).

neg ProduceMessage(?a) :- neg activity(?a).
activity(?a) :- ProduceMessage(?a).

neg ProduceMessage(?a) :- occurrence_of(?o,?a) and neg produces(?o,_#2(?a,?o)).
neg occurrence_of(?o,?a) :- ProduceMessage(?a) and neg produces(?o,_#2(?a,?o)).
produces(?o,_#2(?a,?o)) :- ProduceMessage(?a) and occurrence_of(?o,?a).
```

A ReadMessage activity is an activity whose occurrences read messages.

```
ReadMessage(?a) :- activity(?a) and neg occurrence_of(_#1(?a),?a).
neg activity(?a) :- neg ReadMessage(?a) and neg occurrence_of(_#1(?a),?a).
occurrence_of(_#1(?a),?a) :- neg ReadMessage(?a) and activity(?a).

ReadMessage(?a) :- activity(?a) and reads(_#1(?a),?m).
neg activity(?a) :- reads(_#1(?a),?m) and ReadMessage(?a).
neg reads(_#1(?a),?m) :- neg ReadMessage(?a) and activity(?a).

neg ReadMessage(?a) :- neg activity(?a).
activity(?a) :- ReadMessage(?a).

neg ReadMessage(?a) :- occurrence_of(?o,?a) and neg reads(?o,_#2(?a,?o)).
neg occurrence_of(?o,?a) :- ReadMessage(?a) and neg reads(?o,_#2(?a,?o)).
reads(?o,_#2(?a,?o)) :- ReadMessage(?a) and occurrence_of(?o,?a).
```

A DestroyMessage activity is an activity whose occurrences destroy messages.

```
DestroyMessage(?a) :- activity(?a) and neg occurrence_of(_#1(?a),?a).
neg activity(?a) :- neg DestroyMessage(?a) and neg occurrence_of(_#1(?a),?a).
```

```
occurrence_of(_#1(?a),?a) :- neg DestroyMessage(?a) and activity(?a).

DestroyMessage(?a) :- activity(?a) and destroys(_#1(?a),?m).
neg activity(?a) :- destroys(_#1(?a),?m) and DestroyMessage(?a).
neg destroys(_#1(?a),?m) :- neg DestroyMessage(?a) and activity(?a).

neg DestroyMessage(?a) :- neg activity(?a).
activity(?a) :- DestroyMessage(?a).

neg DestroyMessage(?a) :- occurrence_of(?o,?a) and neg destroys(?o,_#2(?a,?o)).
neg occurrence_of(?o,?a) :- DestroyMessage(?a) and neg destroys(?o,_#2(?a,?o)).
destroys(?o,_#2(?a,?o)) :- DestroyMessage(?a) and occurrence_of(?o,?a).
```

Channel

If a message is contained in a channel, then it is produced by an occurrence of a service that is a source for the channel.

```
channel_source(?c,_#1(?c,?m)) :- channel_mobject(?c,?m).
neg channel_mobject(?c,?m) :- neg channel_source(?c,_#1(?c,?m)).
occurrence_of(_#2(?c,?m),_#1(?c,?m)) :- channel_mobject(?c,?m).
neg channel_mobject(?c,?m) :- neg occurrence_of(_#2(?c,?m),_#1(?c,?m)).
produces(_#3(?c,?m),?m) :- channel_mobject(?c,?m).
neg channel_mobject(?c,?m) :- produces(_#3(?c,?m),?m).
subactivity_occurrence(_#3(?c,?m),_#2(?c,?m)) :- channel_mobject(?c,?m).
neg channel_mobject(?c,?m) :- neg subactivity_occurrence(_#3(?c,?m),_#2(?c,?m)).
```

If a message is contained in a channel, then it is read by an occurrence of a service that is a target for the channel.

```
channel_target(?c,_#1(?c,?m)) :- channel_mobject(?c,?m).
neg channel_mobject(?c,?m) :- neg channel_target(?c,_#1(?c,?m)).
occurrence_of(_#2(?c,?m),_#1(?c,?m)) :- channel_mobject(?c,?m).
neg channel_mobject(?c,?m) :- neg occurrence_of(_#2(?c,?m),_#1(?c,?m)).
reads(_#3(?c,?m),?m) :- channel_mobject(?c,?m).
neg channel_mobject(?c,?m) :- reads(_#3(?c,?m),?m).
subactivity_occurrence(_#3(?c,?m),_#2(?c,?m)) :- channel_mobject(?c,?m).
neg channel_mobject(?c,?m) :- neg subactivity_occurrence(_#3(?c,?m),_#2(?c,?m)).
```

Exceptions

Exceptions form a sublcass of fluents.

```
fluent(e) :- exception(e).
neg exception(e) :- neg fluent(e).
```

The following axioms capture the relationships in Figure 7.

An exception ?e is handled by an activity ?a.

```
exception(?e) :- is_handled_by(?e,?a).
neg is_handled_by(?e,?a) :- neg exception(?e)
```

```
activity(?a) :- is_handled_by(?e,?a).
neg is_handled_by(?e,?a) :- neg activity(?a).
```

An activity ?a has an exception ?e.

```
activity(?a) :- has_exception(?a,?e).
neg has_exception(?a,?e) :- neg activity(?a).
exception(?e) :- has_exception(?a,?e).
neg has_exception(?a,?e) :- neg exception(?e).
```

An activity ?a is an exception-handling activity if and only if it is a TriggeredActivity that there exists an exception that is handled by ?a.

```
handle_exception(?a) :- is_handled_by(?e, ?a) and TriggeredActivity(?a).
neg is_handled_by(?e, ?a) :- neg handle_exception(?a) and TriggeredActivity(?a).
neg TriggeredActivity(?a) :- is_handled_by(?e, ?a) and neg handle_exception(?a).

is_handled_by(_#1(?a),?a) :- handle_exception(?a).
neg handle_exception(?a) :- neg is_handled_by(_#1(?a),?a).
TriggeredActivity(?a) :- handle_exception(?a).
neg handle_exception(?a) :- neg TriggeredActivity(?a).
```

find_exception and fix_exception are the two subclasses of exception-handling activities.

```
handle_exception(?a) :- find_exception(?a).
neg  find_exception(?a) :- neg handle_exception(?a).
handle_exception(?a) :- fix_exception(?a).
neg fix_exception(?a) :- neg handle_exception(?a).

neg handle_exception(?a) :- neg find_exception(?a) and neg fix_exception(?a).
find_exception(?a) :- neg fix_exception(?a) and handle_exception(?a).
fix_exception(?a) :- handle_exception(?a) and neg find_exception(?a).
```

detect_exception and anticipate_exception are the two subclasses of exception-finding activities.

```
find_exception(?a) :- detect_exception(?a).
neg  detect_exception(?a) :- neg find_exception(?a).
find_exception(?a) :- anticipate_exception(?a).
neg anticipate_exception(?a) :- neg find_exception(?a).

neg find_exception(?a) :- neg detect_exception(?a) and neg anticipate_exception
(?a).
detect_exception(?a) :- neg anticipate_exception(?a) and find_exception(?a).
anticipate_exception(?a) :- find_exception(?a) and neg detect_exception(?a).
```

avoid_exception and resolve_exception are the two subclasses of exception-fixing activities.

```
fix_exception(?a) :- avoid_exception(?a).
neg  avoid_exception(?a) :- neg fix_exception(?a).
fix_exception(?a) :- resolve_exception(?a).
neg resolve_exception(?a) :- neg fix_exception(?a).

neg fix_exception(?a) :- neg avoid_exception(?a) and neg resolve_exception(?a).
avoid_exception(?a) :- neg resolve_exception(?a) and fix_exception(?a).
resolve_exception(?a) :- fix_exception(?a) and neg avoid_exception(?a).
```

The relation is_found_by is the restriction of the is_handled_by relation to exception-finding activities. The relation is_fixed_by is the restriction of the is_handled_by relation to exception-fixing activities.

```
is_handled_by(?e,?a) :- is_found_by(?e,?a) and find_exception(?a).
neg is_found_by(?e,?a) :- neg is_handled_by(?e,?a) and find_exception(?a).
neg find_exception(?a) :- neg is_handled_by(?e,?a) and is_found_by(?e,?a).

is_handled_by(?e,?a) :- is_fixed_by(?e,?a) and fix_exception(?a).
neg is_fixed_by(?e,?a) :- neg is_handled_by(?e,?a) and fix_exception(?a).
neg fix_exception(?a) :- neg is_handled_by(?e,?a) and is_fixed_by(?e,?a).

neg is_handled_by(?e,?a) :- neg is_found_by(?e,?a) and neg is_fixed_by(?e,?a).
is_found_by(?e,?a) :- is_handled_by(?e,?a) and neg is_fixed_by(?e,?a).
is_fixed_by(?e,?a) :- is_handled_by(?e,?a) and neg is_found_by(?e,?a).

neg is_handled_by(?e,?a) :- neg find_exception(?a) and neg is_fixed_by(?e,?a).
find_exception(?a) :- is_handled_by(?e,?a) and neg is_fixed_by(?e,?a).
is_fixed_by(?e,?a) :- is_handled_by(?e,?a) and neg find_exception(?a).

neg is_handled_by(?e,?a) :- neg is_found_by(?e,?a) and neg fix_exception(?a).
is_found_by(?e,?a) :- is_handled_by(?e,?a) and neg fix_exception(?a).
fix_exception(?a) :- is_handled_by(?e,?a) and neg is_found_by(?e,?a).

neg is_handled_by(?e,?a) :- neg find_exception(?a) and neg fix_exception(?a).
find_exception(?a) :- is_handled_by(?e,?a) and neg fix_exception(?a).
fix_exception(?a) :- is_handled_by(?e,?a) and neg find_exception(?a).
```

Note: The first triplet of rules is the omnidirectional set for this clause:
  (a) is_handled_by(?e,?a) or neg is_found_by(?e,?a) or neg find_exception(?a)
The second triplet is the omnidirectional set for this clause:
  (b) is_handled_by(?e,?a) or neg is_fixed_by(?e,?a) or neg fix_exception(?a)
The third triplet is the omnidirectional set for this clause:
  (c) neg is_handled_by(?e,?a) or is_found_by(?e,?a) or is_fixed_by(?e,?a).
The fourth triplet is the omnidirectional set for this clause:
  (d) neg is_handled_by(?e,?a) or find_exception(?a) or is_fixed_by(?e,?a).
The fifth triplet is the omnidirectional set for this clause:
  (e) neg is_handled_by(?e,?a) or is_found_by(?e,?a) or fix_exception(?a).
The sixth triplet is the omnidirectional set for this clause:
  (f) neg is_handled_by(?e,?a) or find_exception(?a) or fix_exception(?a).

Clauses (a) and (b) are equivalent to the <== direction of the FLOWS rule.
Clauses (c), (d), (e), and (f) are equivalent to the ==> direction of the FLOWS rule.

The relation is_detected_by is the restriction of the is_found_by relation to exception-detecting activities. The relation is_anticipated_by is the restriction of the is_found_by relation to exception-anticipating activities.

```
is_found_by(?e,?a) :- is_detected_by(?e,?a) and detect_exception(?a).
neg is_detected_by(?e,?a) :- neg is_found_by(?e,?a) and detect_exception(?a).
neg detect_exception(?a) :- neg is_found_by(?e,?a) and is_detected_by(?e,?a).

is_found_by(?e,?a) :- is_anticipated_by(?e,?a) and anticipate_exception(?a).
neg is_anticipated_by(?e,?a) :- neg is_found_by(?e,?a) and anticipate_exception
(?a).
neg anticipate_exception(?a) :- neg is_found_by(?e,?a) and is_anticipated_by(?
e,?a).
```

```
neg is_found_by(?e,?a) :- neg is_detected_by(?e,?a) and neg is_anticipated_by(?
e,?a).
is_detected_by(?e,?a) :- is_found_by(?e,?a) and neg is_anticipated_by(?e,?a).
is_anticipated_by(?e,?a) :- is_found_by(?e,?a) and neg is_detected_by(?e,?a).

neg is_found_by(?e,?a) :- neg detect_exception(?a) and neg is_anticipated_by(?
e,?a).
detect_exception(?a) :- is_found_by(?e,?a) and neg is_anticipated_by(?e,?a).
is_anticipated_by(?e,?a) :- is_found_by(?e,?a) and neg detect_exception(?a).

neg is_found_by(?e,?a) :- neg is_detected_by(?e,?a) and neg anticipate_exception
(?a).
is_detected_by(?e,?a) :- is_found_by(?e,?a) and neg anticipate_exception(?a).
anticipate_exception(?a) :- is_found_by(?e,?a) and neg is_detected_by(?e,?a).

neg is_found_by(?e,?a) :- neg detect_exception(?a) and neg anticipate_exception
(?a).
detect_exception(?a) :- is_found_by(?e,?a) and neg anticipate_exception(?a).
anticipate_exception(?a) :- is_found_by(?e,?a) and neg detect_exception(?a).
```

The relation is_avoided_by is the restriction of the is_fixed_by relation to exception-avoiding activities. The relation is_resolved_by is the restriction of the is_fixed_by relation to exception-resolving activities.

```
is_fixed_by(?e,?a) :- is_avoided_by(?e,?a) and avoid_exception(?a).
neg is_avoided_by(?e,?a) :- neg is_fixed_by(?e,?a) and avoid_exception(?a).
neg avoid_exception(?a) :- neg is_fixed_by(?e,?a) and is_avoided_by(?e,?a).

is_fixed_by(?e,?a) :- is_resolved_by(?e,?a) and resolve_exception(?a).
neg is_resolved_by(?e,?a) :- neg is_fixed_by(?e,?a) and resolve_exception(?a).
neg resolve_exception(?a) :- neg is_fixed_by(?e,?a) and is_resolved_by(?e,?a).

neg is_fixed_by(?e,?a) :- neg is_avoided_by(?e,?a) and neg is_resolved_by(?e,?
a).
is_avoided_by(?e,?a) :- is_fixed_by(?e,?a) and neg is_resolved_by(?e,?a).
is_resolved_by(?e,?a) :- is_fixed_by(?e,?a) and neg is_avoided_by(?e,?a).

neg is_fixed_by(?e,?a) :- neg avoid_exception(?a) and neg is_resolved_by(?e,?a).
avoid_exception(?a) :- is_fixed_by(?e,?a) and neg is_resolved_by(?e,?a).
is_resolved_by(?e,?a) :- is_fixed_by(?e,?a) and neg avoid_exception(?a).

neg is_fixed_by(?e,?a) :- neg is_avoided_by(?e,?a) and neg resolve_exception(?
a).
is_avoided_by(?e,?a) :- is_fixed_by(?e,?a) and neg resolve_exception(?a).
resolve_exception(?a) :- is_fixed_by(?e,?a) and neg is_avoided_by(?e,?a).

neg is_fixed_by(?e,?a) :- neg avoid_exception(?a) and neg resolve_exception(?a).
avoid_exception(?a) :- is_fixed_by(?e,?a) and neg resolve_exception(?a).
resolve_exception(?a) :- is_fixed_by(?e,?a) and neg avoid_exception(?a).
```

Appendix D: Reference Grammars

This is Appendix D of the Semantic Web Services Ontology (SWSO) document.

AtomicProcess Precondition Axioms

If an occurrence of an activity is legal (i.e., if its preconditions are met) then it implies that a particular condition must be true in the situation prior to that activity occuring. That condition is known as a precondition.

```
< atomic_process_precond > ::=  forall ?s (occurrence(?s,< term >) and legal(?
s)) ==> < simple_state_axiom >)))

< simple_state_axiom > ::=
      ({forall | exists} < variable >*) < simple_state_formula >)

< simple_state_formula > ::=
          < simple_state_literal > |
          (not < simple_state_formula >) |
          (< simple_state_formula > and < simple_state_formula >)* |
          (< simple_state_formula > or < simple_state_formula >)* |
          (==> < simple_state_formula >) |
          (<==> < simple_state_formula >)

< simple_state_literal > ::=    prior(< term  >,?s)
```

AtomicProcess Effect Axiom

If a certain condition holds in a situation and an activity occurs in that situation, then another condition will hold in the resulting situation. The condition that holds in the resulting situation is known as an effect of the activity occurrence.

```
< atomic_process_effect > ::=
          forall ?s (occurrence(?s,<term >) and < simple_state_axiom >) ==> <
simple_holds_axiom >))

< simple_holds_axiom > ::=
          ({forall | exists} < variable >*) < simple_holds_formula >)

< simple_holds_formula > ::=
          < simple_holds_literal > |
          (not < simple_holds_formula >) |
          < simple_holds_formula >) and < simple_holds_formula >) |
          (< simple_holds_formula > or < simple_holds_formula >) |
          (< simple_holds_formula > <==> < simple_holds_formula >) |
          (< simple_holds_formula > <==> < simple_holds_formula >)

< simple_holds_literal > ::= holds(< term >,?s)
```

AtomicProcess Input Axioms, i.e., Knowledge Precondition Axioms over inputs

Note that these have the same form as the AtomicProcess Precondition Axioms except that the condition that must hold is a formula involving Kref (Know the value of) fluents.

```
< atomic_process_k_precond > ::=
         forall ?s (occurrence(?s,< term >) and legal(?s)) < ==>
simple_kref_axiom >)))


< simple_kref_axiom > ::=
         ({forall | exists} < variable >*) < simple_kref_formula >)


< simple_kref_formula > ::=
         < simple_kref_literal > |
         (not < simple_kref_formula >) |
         (simple_kref_formula > and < simple_kref_formula >) |
         (simple_kref_formula > or < simple_kref_formula >) |
         (< simple_kref_formula > ==> < simple_kref_formula >) |
         (< simple_kref_formula > <==> < simple_kref_formula & gt;)


< simple_kref_literal > ::=    prior(kref(< term  >,?s)
```

AtomicProcess Output Axioms, i.e., Knowlege Effect Axioms over outputs

Note that these have the same form as the AtomicProcess Effect Axioms except that the condition that must hold after occurrence of the activity is a formula involving Kref (Know the value of) fluents.

```
< atomic_process_k_effect > ::=
          forall ?s (occurrence(?s,<term >) and < simple_state_axiom >) ==> <
simple_kref_axiom >


< simple_kref_axiom > ::=
          ({forall | exists} < variable >*) < simple_kref_formula >)


< simple_kref_formula > ::=
          < simple_kref_holds_literal > |
          (not < simple_kref_formula >) | (< simple_kref_formula > and <
simple_kref_formula >) |
          (< simple_kref_formula > or < simple_kref_formula >)


< simple_kref_holds_literal > ::= holds(kref(< term >,?s))
```

Split

```
< split_axiom > ::=forall ?occ < variable >*        occurrence(?occ, < variable
>) ==>  (exists < variable >+                        (< precedes_formula >*
and                                 < parallel_formula >*
and                                 subactivity_occurrence(< variable > <
variable >)))
< precedes_formula > ::= soo_precedes(< variable > < variable > < term >)
|                             (< precedes_formula > and < precedes_formula >)
< parallel_formula > ::= parallel(< variable > < variable > < term >)
|                             (< precedes_formula > and < precedes_formula >)
```

Sequence

```
< sequence_axiom > ::=forall ?occ,< variable >*        occurrence_of(?occ, <
variable > ==>  (exists < variable >+                        <
precedes_formula > and < precedes_formula >*
and                                subactivity_occurrence(< variable > <
variable >)))
```

Unordered

```
< unordered_axiom > ::=forall ?occ,< variable >*)        occurrence_of(?occ, <
variable > ==>  (exists (< variable >+)
subactivity_occurrence(< variable > < variable > and subactivity_occurrence(<
variable > < variable >*)))
```

Choice

```
< choice_axiom > ::=  forall < variable >*
(same_grove < variable > ?occ) ==>
(< occurrence_sentence > or < occurrence_sentence >*)))
< occurrence_literal > ::=  occurrence_of(< variable > < term >)
|                              subactivity_occurrence(< variable >,< variable
>)
< occurrence_formula > ::=  < occurrence_literal >
|                              (< occurrence_literal > and <
occurrence_literal >*)
< occurrence_sentence > ::=         (exists (< variable
>*)                                    < occurrence_formula >)
```

Conditional

```
< conditional_axiom > ::=   (< simple_state_axiom >
==>                                         < variation_formula >)
```

Iterate

```
< iterate_axiom > ::= forall (< term >,?s1,?s2
                                do(< term >,?s1,?s2) <==>
                                        < rep_formula >))

< rep_formula > ::=  exists (< term >)
                        (subactivity(< term >,< term >) and
                                (forall ?s3
                                   (do(< term >,?s1,?s3) ==>
                                            (?s2 = ?s3) or
                                            do(< term >,?s3,?s2))
```

Repeat

```
< repeat_axiom > ::=   (< simple_state_axiom >
==>                                 < iterate_axiom >)
```

OrderedActivity

```
< ordered_axiom > ::=    forall < variable >
                                (same_grove(< variable >,?occ) ==>
                                          < ordered_sentence >) |
                        forall < variable >
                                (same_grove(< variable >,?occ) ==>
                                          < ordered_formula >)

< ordered_literal > ::=   min_precedes(< variable >,< variable >,< term >) |
                          next_subocc(< variable >,< variable >,< term >)

< ordered_list > ::=      < ordered_literal > |
                          (< ordered_literal > and < ordered_literal >)*

< conditional_occurrence > ::=  (< occurrence_formula > ==>
                                          < ordered_list >)

< ordered_sentence > ::=        (exists ?occ,< variable >
                                        root_occ(< variable >,?occ) and
                                        < occurrence_disjunct > and
                                        < conditional_occurrence >)

< ordered_conjunct > ::=        < ordered_literal > |
                                (and < ordered_formula >*)

< ordered_formula > ::=         (exists ?occ,< variable >
                                        root_occ(< variable >,?occ)
                                        < occurrence_disjunct > and
                                        < ordered_conjunct >) |
                                (< ordered_formula > or < ordered_formula >)*
```

OccActivity

```
< occactivity_axiom > ::=   forall ?occ,< variable >
                                (same_grove(< variable >,?occ) ==>
                                          < occurrence_sentence >) |
                             forall ?occ,< variable >*
                                  same_grove(< variable >,?occ) ==>
                                  (< occurrence_sentence > or <
occurrence_sentence >)*))

< occurrence_literal > ::=      occurrence_f(< variable >,< term >) |
                                subactivity_occurrence(< variable >,< variable
>)

< occurrence_formula > ::=      < occurrence_literal > |
                                (< occurrence_literal > and <
occurrence_literal >)*

< occurrence_sentence > ::=     (exists < variable >*
                                        < occurrence_formula >)
```

TriggeredActivity

```
< trigger_activity > ::=        forall ?s
                                        (< simple_state_axiom > ==>
                                                  < distribution_formula >)
< distribution_formula > ::=    exists ?occ
                                        (occurrence(?occ,?a) and
                                        root_occ(?s,?occ))
```