A Dependency Markup Language for Web Services

Robert Tolksdorf

Freie Universität Berlin, Institut für Informatik
Netzbasierte Informationssysteme
Takustr. 9, D-14195 Berlin, Germany,
mailto:research@robert-tolksdorf.de, *http://www.robert-tolksdorf.de*
DRAFT of CRC

Abstract. Current mechanisms for the description of Web Services and their composition are either to coarse – by specifying a functional interface only – or too fine – by specifying a concrete control flow amongst services.
We argue that more adequate specifications can be built on the notion of *dependency* of activities and coordination activities to manage these dependencies. We propose a Dependency Markup Language to capture dependencies amongst activities and generalizations/specializations amongst processes. With that, we can describe composite services at more suited levels of abstraction and have several options to use such descriptions for service coordination, service discovery and service classification.

1 Introduction

Web Services implement the notion of small networked services that can be combined to realize more complex processes or composite services (that we sometimes call processes in the following). With standards for the representation of their interfaces, their composition and the exchange of messages with them, an open market for services becomes possible. They are enabling to compose complex processes from services that are offered and implemented by different organizations.

The goal with Web Services is to allow an *automated* discovery and composition of services. For that, means and languages for machine-readable descriptions of services are necessary. XML is considered to offer the best chances for a wide exchangeability of such descriptions.

Currently, there is a number of proposed XML-based languages existing and emerging for the description of Web Services. We can assume that they will converge in the midterm. Table 1 shows some of the currently discussed standards for the representation of combined services and their main characteristic. [AMS02] gives a brief introduction to most of them.

The proposed languages usually address two aspects that describe a service: its external interface and its internal behavior. The external interface is specified by a functional description of the services provided. Semantic information can be given in the form of contracts, either for the service provider – for example with pre- and postconditions – or the service user – how the services have to be used. The internal behavior can be specified as a service flow which is commonly described by the specification of a control flow.

Language	External	Internal
WSFL [Ley01]	functional interface specified	data- and control flows specified
XLANG [Tha01]	interface specified with WSDL	control flow specified, event handling
WSCL [BBB+02]	allowed interactions (conversation) specified by interaction-transition net	not specified
DAML-S [ABH+02]	functional interface specified	control flow specified by imperative constructs
ASDL [ZC02]	allowed usage specified by state-(conditioned)transition net	control flow specified
WSMF [FB02]	functional interface specified with pre- and postconditions	not specified
BPSS [Bus01]	interactions specified by message exchanged	state/transition net
BPML [Agr01]	interactions specified by message exchanged	control flow specified by imperative constructs

Table 1. Current (proto-)standards for combined services

In order to identify a service, one searches for a specific interface, perhaps with further requirements on contracts. In the next section, we discuss why this is not always the right kind of service description.

2 How to describe processes

[WL95] describes an interesting set of processes, that describe different ways of eating at a restaurant. For example, the visit to a full-service restaurant is a process ordering–cooking–serving–eating–paying. But there are more ways to visit a restaurant (that is, to implement the composite service restaurantVisit), as shown in table 2. To speak consistently from the perspective of the service-provider, we use the terms take order, cook, serve and collect. We also assume that anything serves is eaten, so we leave out eating in the following.

Restaurant	Service flow
Full service	take order–cook–serve–collect
Fast food	cook–take order–collect–serve
Buffet	cook–take order–serve–collect
Church supper	collect–take order–cook–serve

Table 2. Ways to visit restaurants (after [WL95])

All these processes do have the same external interface void visitRestaurant(Money wallet, Order whatToEat) perhaps with being repleted=TRUE as a postcondition. When selecting a restaurant for an evening, however, that interface is most obvious – the actual

choice is on the internal structure of the service. The quality of services that is of interest to the user therefore is not solely the external interface, it can also be the internal behavior.

The current description mechanism are not sufficient to allow this for clients. There are two main reasons for this:

- Several languages do not specify the internal behavior with a service description at all (eg. WSCL, WSMF).
- The means to specify the internal service-flow are low-level and allow only the expression of the control flow. This addresses only the syntactic structure of the process and leaves no room to have multiple semantically equal processes fulfill the specification. The only exception in this respect is WSFL which can express dataflows.

We could say that precision and recall are low when describing processes by their functional interfaces only. Returning to the restaurant example, "visiting a restaurant" is an *abstract* process that is implemented by various concrete ones, the table above shows four of them.

If we want a restaurant service where the food is cooked freshly after we order it, only Full service and Church supper can satisfy our request, while Buffet and Fast food cannot. The "cooked fresh" service is both an implementation of the abstract restaurant visits, but at the same time an *abstract* process which is implemented by the two named processes. The specification, that we want cooking to occur after ordering discriminates it from the other two processes that we do not want.

In the following, we propose to introduce notions of *abstraction* and *specialization* as relations amongst composite Web Services and the notion of *dependencies* to abstractly specify the internal behavior of composed services.

3 Specifying and relating processes

We now look closer at the notions of dependency and specializations.

3.1 Dependencies

The semantical construct we use above for the "cooked fresh" service is that what is cooked *depends* on what is ordered. Also, the start of cooking depends on the end of ordering. The dependencies imply that the control flow in the implementation of such a service will reach order before cook. This makes, however, no statement on whether we pay at the beginning of our visit or at the end. If necessary, we could specify this by stating a dependency of collect on serve. All four composite services described above can be specific by stating that we want to eat something cooked, serve depends on cook.

[MC94] observes that dependencies are the basis on which processes are coordinated and defines coordination as the *management of dependencies*. This is currently one of the most accepted notions of coordination.

How the dependencies are managed depends on the coordination mechanism applied. If multiple entities depend on the availability of some resource, a central coordination mechanism could be applied that selects one entity to get a lock on the resource. Another mechanism would be a market-like coordination where entities would have to bid for the resource. If no dependency amongst activities exist, the scheduling can be arbitrary – if we only want a place where food is cooked fresh after we ordered, we do not care when we have to pay.

In our example, we use a temporal dependency between ordering and cooking. In the beginning, we have hidden another dependency for simplicity – the food has to be served to be eaten. That dependency expresses that the food has to be transferred to the customer and that the dependency is resolved when is it placed on his desk.

[Cro91,Del96] have studied kinds of dependencies and associated mechanisms. Table 3 shows the main kinds of dependencies distilled by these studies. If a resource is shared, entities can depend on exclusive access to the resource. Some resource allocation mechanism manages that dependency, for example by scheduling, locking etc. If two entities exchange information the work of the consumer depends on the ability to use that information, for example by understanding the format and semantics of the data. Standardization is a mechanism that tries to resolve that dependency by globally understandable data formats. See the references for more elaborate analysis.

Coordination mechanism	Dependency managed	
Resource allocation		Shared resources
Notification Transportation Standardization	Prerequisite Transfer Usability	Producer/ Consumer
Synchronization		Simultaneity
Goal selection Decomposition		Task/Subtask

Table 3. Coordination mechanisms and managed dependencies [Cro91,Del96]

[RG00,RG01] introduce dependencies as a modeling concept to describe scenarios for testing software components. The dependencies modeled and the mechanism to manage them – specific styles of execution order – are shown in table 4.

From these results – there are certainly more studies on dependencies – we see that the usual focus on a control flow focuses on a secondary aspect. A concrete control flow describes the result of a mechanism that manages dependencies. There are many ways to find such a result, eg. solving a set of constraints by a central scheduler or a decentralized mechanism like bidding for a resource like a CPU. Also, there are many concrete schedules to manage some dependency. If, for example, a and b depend on some resource r, the control flows a.b and b.a (the dot means sequential composition here), are equally valid results of managing this dependency by establishing exclusive access.

Dependency class	Dependency
Temporal	Strict sequence
	Real-time
Causal	Loose sequence
	Data dependency
	Resource dependency
Abstraction	Generalization/Refinement
	Aggregation

Coordination Mechanism	Execution order
Sequence	Sequence
Alternative	Choice
	Conditioned choice
Iteration	Loop
	Conditioned loop
Concurrency	Accidental
	Enforced
	Prohibited

Table 4. Modeling concepts for dependency charts [RG00,RG01]

3.2 Abstraction and specialization

The above example talks about processes in three levels of abstractions: 1) the concrete processes in restaurants, 2) the abstract specification of "freshly cooked" and 3) the most abstract notion "restaurant visit".

For computational processes, sound notions for the formal specification of behavior exist. For business processes, however, "qualities" like "freshly cooked" become important that are hard to capture. For the "computation" "visit a restaurant", all behaviors shown above are computationally equivalent. With the intention that one has when expressing abstractions on business processes, other notations are necessary.

[MCL+99] mentions two dimensions when analyzing processes. The most common view is to talk about the *parts* of a process, that is the sub-activities that have to be taken. In our example, this refers to the respective services that compose the restaurant visit. Another, equally important dimension is the *type* of process. In our example, this refers to the grouping of Full service and Church supper into the type "freshly cooked". The levels of abstraction mentioned then lead to a hierarchy of specializations and generalizations of processes.

[WL95] approach specialization concepts for processes. They develop specialization and refinement transformations and the respective generalization and abstraction transformations. With these, hierarchies of processes can be derived. The kinds of restaurant visits can be generalized into a generic restaurant visit which includes the union of all possible orderings of services. With specialization then, new processes can be derived.

Still, for business processes, not all possible specializations of the most abstract service "do something" are useful ones. It remains to the modeler to put the focus on interesting, useful and possible processes.

3.3 Typing Web Services

The typing concepts for Web services are currently not very elaborate. The notions of port- and service-types are quite similar to the notions of interfaces in object-based standards such as OMG/CORBA. There, interface-types are related by specialization and generalization and a formal notion of a contravariant subtyping of interfaces.

We expect that such mechanisms are equally usable for Web Services and will enable some sort of Web Services trading. At such a trader, one would request a service of some interface/service type and get a reference to a service with a compatible interface/service type.

This well known mechanism of service-discovery addresses only the syntactic aspects of what a client expects from the service. The contract between the client and the server concerns only the kind and format of data exchanged but says nothing about what the service does.

We propose that dependencies are used as an additional information about the internal workings of a service. It could be provided together with the service type description and can be taken into account during service discovery. The description of internal service characteristics by dependencies is complementary to the description of external characteristics by interfaces.

Abstraction and specialization serve several purposes:

- During discovery, clients can express abstract expectations on the dependencies ruling the workings of the service. The service found during service discovery will observe these dependencies, perhaps some additional ones (see section 5.2).
- The level of detail in the description can be more abstract or more concrete depending on how much information the service provide is willing to disclose. All descriptions along the abstraction/specialization hierarchy are, however, valid descriptions of the service.
- Abstraction and specialization express a relative semantic of what the services do. This semantic is explicitly provided by the service description, but there are also option for deriving it automatically (see section 5.3).

4 A Dependency Markup Language

Starting with some example processes that are similar we claimed that the notions of dependencies and generalization/specialization are better suited to describe composite services than just functional interfaces or just a specific control flow.

For Web Services, we propose a *Dependency Markup Language* (DML) which provides the necessary language to express both. The resulting specification is more abstract than a concrete control flow and a more specific service description than a functional interface.

Figure 1 shows the structure of DML in terms of defined elements. A DML description consists of a set of dependency type declarations and a set of process descriptions. Each process contains a set of dependency descriptions.

Figure 2 shows an excerpt of the respective XML Schema to document attributes. Dependency types declare a name for them and also can be related by a specializes-declaration.

In the current version, each dependency in a process connects two services (from and to). It can refer to the events of starting or ending the services. The type-Attribute defines the kind of dependency. Processes them self can be related by specialization.

With that, we can put our knowledge on restaurants into a DML file as shown in figure 3.

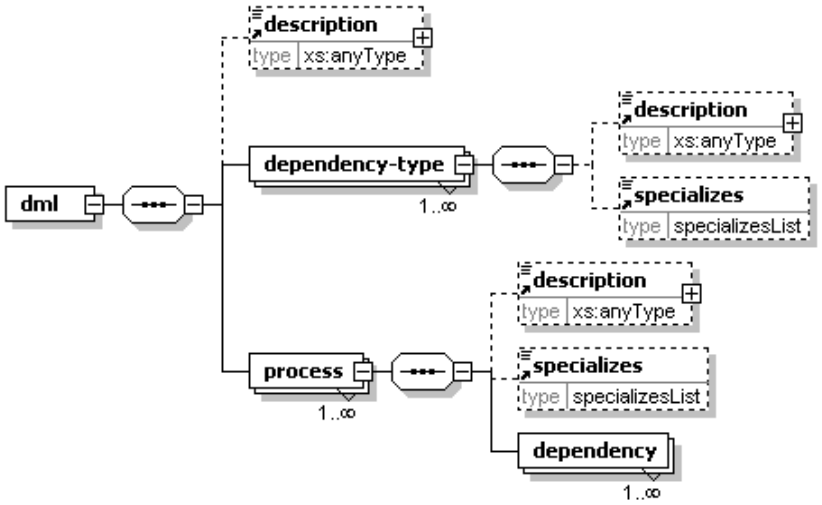

Fig. 1. The structure of DML

5 Processing DML

Given a description of a composite service with DML, there are several ways to process it. We foresee a *coordination environment* for DML that contains several services that perform functionalities as described in the next subsections.

5.1 Coordinating Web Services

The dependency-type specifications in DML carry no information about how the dependencies are managed. As seen in 3.1, there are several ways to manage dependencies resulting in different execution orders of services.

The coordination environment for DML contains *coordination services* that *bind* themselves to dependency types they are able to manage. Based on the information given by the DML specification, a coordination service generates a specific service service flows relative to the specification languages for composite services mentioned in the beginning.

To ensure openness and interoperability of these coordination services, a dependency ontology will be necessary. Currently, it is built by the specialization hierarchy within DML. The next step would be the design of an open dependency ontology on the basis of DAML+OIL which would be closely related to approaches like DAML-S. Figure 4 shows the DML notation for the dependency types mentioned in section 3.1.

5.2 Discovering Web Services with DML

In the coordination environment *dependency matchers* are able to determine whether a control flow is a specialization of a process. A concrete control flow is nothing than a

```
[...]
<xs:attributeGroup name="identification">
 <xs:attribute name="id" type="xs:ID" use="required"/>
 <xs:attribute name="name" type="xs:string"/>
</xs:attributeGroup>
<xs:attributeGroup name="specialization">
 <xs:attribute name="specializes" type="xs:anyURI" use="optional"/>
</xs:attributeGroup>
<xs:simpleType name="event">
 <xs:restriction base="xs:string">
  <xs:enumeration value="end"/>
  <xs:enumeration value="start"/>
 </xs:restriction>
</xs:simpleType>
<xs:attribute name="servicereference" type="xs:anyURI"/>
<xs:simpleType name="specializesList">
 <xs:list itemType="xs:anyURI"/>
</xs:simpleType>
<xs:element name="specializes" type="specializesList"/>
[...]
<xs:element name="dependency-type" maxOccurs="unbounded">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="description" minOccurs="0"/>
   <xs:element ref="specializes" minOccurs="0"/>
  </xs:sequence>
  <xs:attributeGroup ref="identification"/>
  <xs:attributeGroup ref="specialization"/>
 </xs:complexType>
</xs:element>
[...]
<xs:element name="process" maxOccurs="unbounded">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="description" minOccurs="0"/>
   <xs:element ref="specializes" minOccurs="0"/>
   <xs:element name="dependency" maxOccurs="unbounded">
    <xs:complexType>
     <xs:attribute name="name" type="xs:string"/>
     <xs:attribute name="type" type="xs:anyURI" use="required"/>
     <xs:attribute name="from" type="xs:anyURI"/>
     <xs:attribute name="from-event" type="event" default="end"/>
     <xs:attribute name="to" type="xs:anyURI"/>
     <xs:attribute name="to-event" type="event" default="start"/>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
  <xs:attributeGroup ref="identification"/>
  <xs:attributeGroup ref="specialization"/>
 </xs:complexType>
</xs:element>
[...]
```

Fig. 2. An excerpt from the DML schema

```
[...]
 <process id="restaurantVisit" name="Visit to a restaurant">
  <description>
   An abstract description of a restaurant visit where only
   cooked food is eaten.
  </description>
  <dependency type="looseSequence" from="cook" to="serve"/>
 </process>
 <process id="freshlyCooked" specializes="restaurantVisit">
  <description>
   An abstract description where things are cooked after an order.
    </description>
  <dependency type="looseSequence" from="takeOrder" to="cook"/>
 </process>
 <process id="fullService" specializes="freshlyCooked">
  <dependency type="strictSequence" from="takeOrder" to="cook"/>
  <dependency type="strictSequence" from="cook" to="serve"/>
  <dependency type="strictSequence" from="serve" to="collect"/>
 </process>
 <process id="fastFood" specializes="restaurantVisit">
  <dependency type="strictSequence" from="cook" to="takeOrder"/>
  <dependency type="strictSequence" from="takeOrder" to="collect"/>
  <dependency type="strictSequence" from="collect" to="serve"/>
 </process>
 <process id="buffet" specializes="restaurantVisit">
  <dependency type="strictSequence" from="cook" to="takeOrder"/>
  <dependency type="strictSequence" from="takeOrder" to="serve"/>
  <dependency type="strictSequence" from="serve" to="collect"/>
 </process>
 <process id="churchSupper" specializes="freshlyCooked">
  <dependency type="strictSequence" from="collect" to="takeOrder"/>
  <dependency type="strictSequence" from="takeOrder" to="cook"/>
  <dependency type="strictSequence" from="cook" to="serve"/>
 </process>
[...]
```

Fig. 3. The restaurant example in DML

dependency specification, only at a less abstract level. Since coordination mechanisms are able to generate such a specialization by applying coordination mechanism, they can also determine whether they can generate a given process (they might not be able to falsify this) as a specialization of a certain DML specification. [WL95] contains a basic set of the necessary relations to do that.

With that, one can implement an enhanced discovery of Web Services that take into account dependencies in addition to any functional interfaces. Such a service would resolve the problems mentioned in the introduction.

5.3 Classifying processes

In DML the specialization hierarchy has to be specified explicitly. There seem to be more options for an automatic detection of such relations and an automatic classification

```
[...]
<dependency-type id="any"/>
<dependency-type id="mit-dependency" specializes="any"/>
<dependency-type id="sharedResources" specializes="mit-dependency"/>
<dependency-type id="producerConsumer" specializes="mit-dependency"/>
<dependency-type id="prerequisite" specializes="producerConsumer"/>
<dependency-type id="transfer" specializes="producerConsumer"/>
<dependency-type id="usability" specializes="producerConsumer"/>
<dependency-type id="simultaneity" specializes="mit-dependency"/>
<dependency-type id="taskSubtask" specializes="mit-dependency"/>
<dependency-type id="unizh-dependency" specializes="any"/>
<dependency-type id="temporal" specializes="unizh-dependency"/>
<dependency-type id="strictSequence">
 <specializes>temporal looseSequence</specializes>
</dependency-type>
<dependency-type id="realTime" specializes="temporal"/>
<dependency-type id="causal" specializes="unizh-dependency"/>
<dependency-type id="looseSequence" specializes="causal"/>
<dependency-type id="dataDependency" specializes="causal"/>
<dependency-type id="resourceDependency" specializes="causal"/>
<dependency-type id="abstraction" specializes="unizh-dependency"/>
<dependency-type id="generalization" specializes="abstraction"/>
<dependency-type id="refinement" specializes="abstraction"/>
<dependency-type id="aggregation" specializes="abstraction"/>
[...]
```

Fig. 4. Dependencies from tables 3 and 4 in DML

of composite services on that basis. Works like [Nie95,MHK98,ZG00] have studied such kind of typing of processes.

For our example, it can be detected that fullService contains the dependency that is specified for freshlyCooked since the dependency from takeOrder to cook exists. The type of the dependency in fullService is strictSequence which is a specialization of looseSequence according to our hierarchy of dependencies. From that, one could infer that fullService specializes freshlyCooked.

However, there are limits to such a classification. The dependencies still capture structural properties of processes. The processes under consideration implement some functionality. Even if two processes share no structural characteristics, they can still implement the same functionality and thus both specialize the same abstraction which might even be empty of dependencies.

6 Outlook

The DML specification is currently being finalized. On this basis, a coordination environment is to be build and tested. This includes the implementation of a set of coordination mechanisms, their binding to dependencies and the generation of control flows from DML. This also includes the implementation of dependency matchers as described.

The implementation of the mentioned services of the coordination environment seems to be straightforward. The coordination services transform DML specifications into formats for service composition and might even be implemented in XSL. Service discovery requires a repository of DML descriptions and an appropriate and simple lookup algorithm. A classification service needs the implementation of some more complex algorithms as mentioned in the above description.

The main obstacle to set up such a coordination environment is to make it rich in expressibility of dependencies. The dependency ontology has to be enlarged by further studies [Tol00]. It has be checked how notions of dependencies can be related by specialization to lead to a unified hierarchy. It has to be tested how the mentioned mechanisms for automatic classification of processes can be applied for DML specifications and whether they lead to useful results. Furthermore, the currently not considered multiparty dependencies have to be explored.

References

[ABH+02] Anupriya Ankolekar, Mark Burstein, Jerry R. Hobbs, Ora Lassila, David Martin, Drew McDermott, Sheila A. McIlraith, Srini Narayanan, Massimo Paolucci andTerry Payne, and Katia Sycara. DAML-S: Web Service Description for the Semantic Web. In I. Horrocks and J. Hendler, editors, *The Semantic Web - ISWC 2002*, volume 2342 of *LNCS*, pages 348–363. Springer-Verlag, 2002.

[Agr01] Ashish Agrawal, editor. *Business Process Modeling Language (BPML) Specification*. Business Process Management Initiative, 2001.

[AMS02] Selim Aissi, Pallavi Malu, and Krishnamurthy Srinivasan. E-Business Process Modeling: The Next Big Step. *IEEE Computer*, 35(5):55–62, May 2002.

[BBB+02] Arindam Banerji, Claudio Bartolini, Dorothea Beringer, Venkatesh Chopella, Kannan Govindarajan, Alan Karp, Harumi Kuno, Mike Lemon, Gregory Pogossiants, Shamik Sharma, and Scott Williams. Web Services Conversation Language (WSCL) 1.0. W3c note, World Wide Web Consortium, 2002. http://www.w3.org/TR/wscl10/.

[Bus01] Business Process Team. ebXML Business Process Specification Schema. Technical report, UN/CEFACT and OASIS, 2001. http://www.ebxml.org/specs/ebBPSS.pdf.

[Cro91] Kevin Ghen Crowston. *Towards a Coordination Cookbook: Recipes for Multi-Agent Action*. PhD thesis, Sloan School of Management, MIT, 1991. CCS TR# 128.

[Del96] Chrysantos Nicholas Dellarocas. *A Coordination Perspective on Software Architecture: Towards a Design Handbook for Integrating Software Components*. PhD thesis, Massachusetts Institute of Technology, 1996.

[FB02] D. Fensel and C. Bussler. The Web Service Modeling Framework WSMF. Technical report, Vrije Universiteit Amsterdam, 2002.

[Ley01] Frank Leymann. Web Services Flow Language Web Services Flow Language Web Services Flow Language Web Services Flow Language (WSFL 1.0). Technical report, IBM Software Group, 5 2001.

[MC94] Thomas W. Malone and Kevin Crowston. The Interdisciplinary Study of Coordination. *ACM Computing Surveys*, 26(1):87–119, March 1994.

[MCL+99] Thomas W. Malone, Kevin Crowston, Jintae Lee, Brian Pentland, Chrysanthos Dellarocas, George Wyner, John Quimby, Charles S. Osborn, Abraham Bernstein, George Herman, Mark Klein, and Elissa O Donnell. Tools for Inventing Organizations: Toward a Handbook of Organizational Processes. *Management Science*, 45(3):425–443, 3 1999.

[MHK98] Max Mühlhäuser, Ralf Hauber, and Theodorich Kopetzky. Typing Concepts for the Web as a Basis for Re-use. In Anne-Marie Vercoustre, Maria Milosavljevic, and Ross Wilkinson, editors, *Proceedings of the Workshop on the Reuse of Webbased Information*, Report Number CMIS 98-111, pages 79–89. CSIRO Mathematical and Information Sciences, 1998.

[Nie95] Oscar Nierstrasz. Regular Types for Active Objects. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, chapter 4, pages 99–121. Prentice Hall, 1995.

[RG00] J. Ryser and M. Glinz. Using Dependency Charts to ImproveScenario-Based Testing. In *Proceedings of the 17th International Conference on Testing Computer Software (TCS2000)*, Washington D.C., 6 2000.

[RG01] J. Ryser and M. Glinz. Dependency Charts as a Means to Model Inter-Scenario Dependencies. In G. Engels, A. Oberweis, and A. Zündorf, editors, *Modellierung 2001*, volume P-1 of *GI-Edition - Lecture Notes in Informatics*, pages 71–80, 2001.

[Tha01] Satish Thatte. XLANG. Web Services for Business Process Design. Technical report, Microsoft Corporation, 2001.

[Tol00] Robert Tolksdorf. Models of Coordination. In Andrea Omicini, Robert Tolksdorf, and Franco Zambonelli, editors, *Engineering Societies in the Agent World First International Workshop, ESAW 2000, Berlin, Germany, August 21, 2000*, number LNAI 1972, pages 78–92. Springer Verlag, 2000.

[WL95] George M. Wyner and Jintae Lee. Applying Specialization to Process Models. In *Conference on Organizational Computing Systems*, Tools, pages 290–301, 1995.

[ZC02] Mladen A. Vouk Zhengang Chang, Munindar P. Singh. Composition Constraints for Semantic Web Services. In *Proceedings of the International Workshop Real World RDF and Semantic Web Applications 2002*, 2002. http://www.cs.rutgers.edu/~shklar/www11/.

[ZG00] Michael Zapf and Kurt Geihs. What Type Is It? A Type System for Mobile Agents. In Robert Trappl, editor, *Proceedings of the 15th European Meeting on Cybernetics and Systems Research*, pages 585–590. Austrian Society for Cybernetic Studies, April 2000.