Abstract

Dominant trends in distributed systems are increasing network connectivity, business-to-business integration and an evolution from desktop-oriented software to service-oriented software. The size and complexity of networked services has furthermore broadened from intra-organization to cross-organizational service provisioning. In the last decade many coordination architectures have been proposed to coordinate the provisioning of services across organizational boundaries. These approaches involve various types of service contracts and policies, and various coordination activities including negotiation, validation, enactment, monitoring and enforcement of service policies and contracts. In this paper, we propose a reference model for such coordination architectures in order to support the following goals: (i) to facilitate the analysis, comparison and discussion of different coordination architectures and (ii) to allow for constructive proposals about improving the existing coordination architectures.

1. Introduction

Organizations need to integrate their business processes in order to be able to operate and survive in a market. When multiple independent organizations interact with each other, they have to coordinate their activities in order to ensure that their interaction leads to an added-value result that is satisfactory to all parties. Coordination, which can also be defined as "the managing of dependencies between agents in order to foster harmonious interaction between them" [21], is indispensable for effective cooperation between autonomous organizations, as well as for safe competition between them[25].

Coordination involves a certain agreement, i.e., a set of rules of engagement, that must be complied with by all participating organizations for the business relationship to be exploited in a safe and harmonious way. The agreement is to be specified using an agreement language. An important assumption is that the organizations involved might not trust each other. So an important requirement for cross-organizational coordination is that it should enforce the regulated interactions (as encoded in the agreement) between two or more mutually distrusting and autonomous organizations.

Information technology is used to speed-up coordination. In this paper a coordination middleware is defined as a distributed management system [23] that provides technological support for the efficient organization of providing and consuming services across organizational boundaries [35]. By studying different coordination architectures (i.e., the software architecture of existing coordination middlewares), we can derive reference models [6] for cross-organizational coordination. A reference model is defined as a standard decomposition of a problem into parts that cooperatively solve the problem. Reference models arise mostly from experience and as a result are a characteristic of mature domains [6]. In our opinion, the understanding and acceptance of a common reference model for cross-organizational coordination is an important prerequisite in order to begin with the definition of a standardized approach for cross-organizational coordination.

The goal of this paper is to propose a reference model for cross-organizational coordination. The scope of the targeted reference model is focussed towards the technological and infrastructural aspects of cross-organizational coordination. We defer the reader, interested in the socio-organizational aspect of coordination, to reference models defined by researchers from the University of St. Gallen [33, 34].

Figure 1 shows a very simple reference model for cross-organizational coordination. Firstly, agreements must be represented digitally by means of an agreement language that offers the necessary concepts for describing and executing agreements. Second, coordination middleware must be
developed in order to establish agreements dynamically, and to enforce the agreements or detect violations against these. Although this simple reference model already presents a commonly accepted decomposition of the major problems involved, it is not detailed enough in order to be of any use. An effective reference model should instead enable us to analyze, compare and classify existing coordination architectures and, secondly, it should also enable us to make constructive proposals about improving the existing coordination architectures. To achieve this effectively, we seek a reference model with a finer-grained decomposition that unravels the different concerns of importance such that it becomes possible to assess these concerns in isolation.

The process of creating the reference model was based on the following approach. First we have performed a literature study of existing cross-organizational coordination middlewares. Secondly, based on the literature we have identified commonly recurring concepts and problems. We have performed this process iteratively and incrementally: a prior version of the reference model is validated by mapping it to a new coordination middleware that was not yet included in the literature study. Any found incompletenesses or inconsisties in the current version of the reference model are then amended, leading to a new version of the reference model. Overall, the process of creating the reference model was mostly based on generalization and abstraction, and not on rigor deduction. The set of coordination middlewares that were included in the literature study have been listed in [40].

The rest of this paper is structured as follows. Section 2 introduces and motivates the reference model for cross-organization coordination. Then section 3 compares seven well-known coordination architectures by mapping these to the reference model. Finally section 4 draws important conclusions from these mappings. In particular we identify important gaps in required coordination functionality that is not bridged by current coordination architectures. This allows us to propose some senseful directions for future work on cross-organizational coordination architectures.

2. Reference model

There is a vast heterogeneous body of research on cross-organizational coordination. Various types of agreements are supported and different kinds of language and middleware technology are being used, different coordination middlewares vary in their multitude of architectural styles such as Peer-2-Peer and Client-Server. However, to solve the problem of enterprise interoperability, it is of paramount importance that, within each business domains, reference architectures are defined with standard interfaces for establishing and executing agreements. A reference model that defines a standard decomposition of the coordination problem is a first step to obtain such reference architecture. A reference architecture for a particular business domain is derived by mapping the reference model for cross-organizational coordination onto software components using an architectural style [6]. The definition of such a domain-specific reference architecture is out of the scope of this paper. The paper only presents a reference model with the afterthought that a mapping to a reference architecture is one of its possible applications.

As pointed out in section 1, the reference model has been obtained by refining the simplified model in Figure 1 with additional dimensions. The simplified model distinguishes between three dimensions: (i) the type of agreements that are established, (ii) the language for describing agreements, and (iii) the middleware for establishing and executing the agreements. The extended reference model (see Figure 2) is then obtained by refining the two last dimensions into 4 and 2 sub-dimensions, respectively. The resulting reference model can thus be represented as a tree that has 7 subdimensions as its leaves:

1. The type of agreements (**Dim. 1**) that are established.
2. The language for describing agreements (**Dim. 2**) where we distinguish between four sub-dimensions:
   (a) the conceptual model of the language for describing agreements, (**Dim. 2.a**).
   (b) the computational model of the language for making agreements implementable (**Dim. 2.b**)
   (c) the management concepts of the language for managing large sets of agreements and dealing with their potential conflicts (**Dim. 2.c**).
   (d) the implementation technology of the language so that computers can process it (**Dim. 2.d**).
3. The middleware for establishing and executing agreements (Dim. 3) where we distinguish between two sub-dimensions:

(a) the functionality that the underlying coordination middleware offers for establishing and executing agreements (Dim. 3.a).

(b) the distributed systems architecture of the coordination middleware (Dim. 3.c).

2.1. Types of agreements

We distinguish between four types of agreements in the literature:

- Type 1: Agreements about the flow of business domain interactions; How is the cross-organizational relationship going to be exploited? What messages will be exchanged between the service consumer and service provider organizations and which data types are involved?

- Type 2: Agreements about modal constraints, i.e. authorizations, obligations, prohibitions, timings[11] that organizations must agree to and must respect as long as their business relationship holds.

- Type 3: Service-Level Agreements (SLAs) about the deployment and configuration of non-functional requirements of provided services, e.g. must the interactions be performed in a transactional way or not, must certain Quality-of-Service (QoS) requirements be enforced during the interactions?

- Type 4: Agreements at the level of protocols and standards used for implementing the non-functional requirements. For example, what specific transaction protocol must be used?

2.2. Language models and technology

In order to regulate the interactions between two or more independent agents, one needs a language for describing how the interactions are to be done. Such a language must (a) offer a conceptual model for describing the regulations at a sufficient high-level of abstraction that is independent from the organizations internal processes and data, (b) offer a computational model that ensures that regulations are implementable by the underlying system, (c) offer additional modeling concepts for managing large sets of agreements and (d) be computer-interpretable.

2.2.1 Conceptual model

A conceptual model for specifying cross-organizational agreements provides the modeling concepts necessary to describe all information that is needed to establish and exploit cross-organizational service relationships. In order to be successful, the conceptual model must have a clear scope on what is the domain or area where the agreements apply. The domain may be a country, or a professional institute, or a specific market such as financial services. Broadly speaking, we can divide the work on modeling concepts for cross-organizational coordination into management policies (e.g., [25, 9, 11]), electronic contracts (e.g.,
Policies

Policies are a means of specifying and influencing management behavior within a distributed system[38]. Policies can be broadly classified in authorization policies and obligation policies, where the former captures access control requirements of the system and the latter is about requirements related to system behavior.

Although policies are originally envisaged to be specified and enforced within a single organization, policies are also useful for regulating inter-organizational interactions. In this context, policies typically support agreement types 1 and 2. For example consider the following four policies from the retail business domain that regulate the interactions between a purchaser and a seller organization:

1. **Authorization Policy 1** "The seller has the right to inspect a purchaser’s order at any time...."

2. **Prohibition Policy 2** "...except when the order is destined for a person with VIP status."

3. **Obligation Policy 3** "The purchaser is obliged to pay the price of a product no later than 10 days after the date of ordering."

4. **Obligation Policy 4** "A seller is required to ship an ordered item within 10 days to a Purchaser after the Purchaser has paid the order."

Electronic contracts

An electronic contract is an agreement between two or more organizations that stipulates how the involved parties are expected to behave. A contract generally consists of a structured set of entries, called clauses.

Contract models exist in the literature for all four agreement types as presented in section 2.1. Electronic contracts for agreement types 1 and 2 are typically electronic versions of conventional contracts on paper between business organizations. Electronic contracts for agreement types 3 and 4 typically correspond with SLA’s and QoS contracts that eventually have to be mapped to available computing resources.

2.2.2 Computational model

Agreements (either contracts or policies) may be specified at the conceptual level (as goals) and then need to be refined into a implementable actions that can be enforced upon the contracted application services.

To achieve implementable actions, the coordination language must also offer a computational model that supports cross-organizational service provisioning across independent organizations. Generally speaking, this computational model offers behavioral concepts that enable coordination architectures (see section 2.3) to observe service behavior by registering for events and control states that relate to the progress with which the service execution fulfills the agreement, and to control the service behavior by allowing, forbidding, or enforcing actions to take place at the service consumer or service provider site.

For example, we will show what behavioral elements are needed such that the above Obligation Policy 4 can be refined into an implementable policy specification:

\[\text{onEvent(OrderPaidByPurchaser):}\]
\[\{\text{action(OrderedItemSentToPurchaser) within(10 days)}\}\]

We can distinguish the following behavioral elements in the policy specification: (1) the subject Seller to which the obligation applies, (2) the action of shipping an ordered item, (3) the target Purchaser (4) an additional time constraint of 10 days, and (5) an activation condition stating that the policy should become active after the OrderPaid-ByPurchaser event has occurred.

All these elements must be mappable to implementable objects and operations available in the underlying software system of the service provider and consumer.

Note that the definition of a policy specification can consist of multiple levels of abstractions. For example, suppose OrderPaidByPurchaser corresponds to a high-level business event that is not necessarily directly represented as a message exchange between purchaser and seller. In order to implement or detect such a business event, it must be refined into a pattern of lower-level events that directly come from the internal processes of the seller and/or purchaser organization. How this is implemented, needs to be specified at a lower level of abstraction.

Event-based architectural description languages already support some of these mechanisms. For example, Rapide [19, 18] provides maps that correlate a complex pattern of multiple event occurrence into a single higher-level business event. A similar mechanism is provided by Finesse [8, 32]. Milosevic [24] and Yildiz [42] both point out the usefulness of these existing complex event processing languages for cross-organizational coordination.

2.2.3 Management concepts

Management concepts are necessary for facilitating agreement specification for large-scale systems with millions of objects. This typically involves a concept that provides a
way of grouping objects to which agreements apply [11]. Also concepts are required for hierarchical grouping of agreements in order to help manage the large set of agreements that apply for a given organization.

Another important aspect of agreement management is the detection and resolution of conflicts between multiple agreements. Existing solutions for dealing with conflicts between policies originate from the work of Lupu and Sloman. They distinguish between two kinds of conflicts: modality conflicts and application-specific conflicts. A modality conflict arises for instance when two policies associate both an authorization and a prohibition to the same subject, action, target tuple. Modality conflicts are domain-independent.

Application-specific conflicts are conflicts that are related to the specific application for which the policies have been defined. Application-specific conflicts are further classified into conflict of duties, conflict of interests, conflict of priorities for resources, multiple manager conflict and self-management conflict. A conflict of duty, for example, will arise when the same subject is permitted to perform actions that, in the context of the application, are defined to be conflicting. For example, in the context of a financial application, the same subject should not have the right to sign a payment check and authorize that payment at the same time. Lupu and Sloman have worked on various solutions for detecting both kinds of conflicts.

2.2.4 Language technology

The modeling and behavioral concepts offered must eventually be expressed by means of a language format that can be processed by a computer. While it is possible to develop a computer language from scratch for this (e.g. [24]), most coordination architectures employ an existing language technology or specification standard. In particular the following language technologies have already been used: XML – e.g. [9], WS-Policy and WS-PolicyAttachment [12] – e.g. [41], Object-oriented programming languages [29] – e.g. [11], the Meta Object Facility (MOF) [30] – e.g. [37], Finite state machines – e.g. [28], Logic programming languages such as Prolog – e.g. [25], reflection as a complementary implementation technology – e.g. [25].

2.3. Coordination middleware functionalities

Besides a language for specifying agreements, coordination middleware is needed to support exploitation of the cross-organizational service provisioning across independent organizations. Broadly speaking, we distinguish between two main responsibilities for cross-organizational coordination middleware [23, 15]: (i) establishment of agreements between a service consumer and service provider organization and (ii) execution of the contracted services in conformance with the agreement.

Based on [23], we decompose the broad responsibilities of the establishment and execution layers into more specific functionalities.

2.3.1 Establishment layer

Establishment involves negotiation and validation of agreements.

Negotiation Organizations aim to use matchmaking technology for speeding up agreement negotiation such that contracted business relationships can be achieved dynamically without engaging into lengthy and costly human-to-human negotiations. With this end in view, agreement templates are often used. An agreement template is a common form that is used as a standard within the agreement domain. Examples of agreement templates are: sales and order contracts, car and house sale contracts, airline tickets. An agreement template is similar to a regular agreement with predefined properties that need to be filled in the course of agreement negotiation [15]. In most agreement templates, the agreeing parties can negotiate the data to be written in the blanks of the template, but not the clauses.

Validation Agreement validation corresponds with validating whether there are no ambiguities in the clauses of an established agreement. An agreement template may help with validation as it may provide rules that constrain how the agreement template may be instantiated such that an agreement is considered valid.

2.3.2 Execution

Execution involves enactment, monitoring and enforcement.

Enactment When agreements will be negotiated dynamically, exploitation of the agreements is preferably also performed on the fly. This requires the ability to defer resource allocation to the latest time, such that an organization is freed from the risk of having to pre-allocate resources for long periods of time [15]. Enactment is the process that dynamically creates the appropriate infrastructure to the point where the exploitation of the business relationship is imminent. Some systems also offer concepts for automating the mapping of an agreement to necessary resources; this

Note that conflicts between the clauses in the same agreement are considered as ambiguities that should be dealt with as part of validating the agreement.
is however limited to coordination middleware supporting agreement types 3 and 4 involving non-functional requirements and QoS. The execution layer in these middleware platforms relies on lower-level middleware such as deployment containers with pre-fabricated common services (e.g., transactions, persistence, security), QoS-enabled application servers [13] and dynamic distributed resource management systems (such as e.g., [5, 43]).

**Monitoring** Although organizations are expected to implement the agreements themselves, one may not trust that organizations will never cheat or deviate from the established agreement. Hence most architectures support **monitoring** the execution of the internal processes and data of the organizations. When violations occur, notifications are sent to the appropriate parties.

**Enforcement** The system may also trigger actions that aim at rectifying the deviations that cause an agreement violation. Molina-Jimenez et al. [28] state that every enforcement middleware must have two important properties: (1) safety, meaning that local policies of an organization should not be compromised by failures or misbehavior by other parties. (2) liveness, requiring that if all the parties are correct (not misbehaving), then agreed interactions should take place despite finite number of temporary omission failures in the network or computers.

**Dismantling** The execution layer must also offer support for terminating running agreements. This is a succinct process that takes a lot of additional coordination between the business parties involved. Typically, it means shutting down the service, or temporarily putting the overall system in a dual mode where ongoing transactions are completely monitored against the agreement, but new transactions are not subject anymore to the agreement.

### 2.4. Distributed architecture

A final interesting dimension is the distributed architectural decisions behind the design of the coordination middleware. As monitoring and enforcement are the most complex functionalities, these are often the most weighing factors in the distributed architecture of the coordination middleware.

Generically speaking, a monitoring and enforcement system consists of a **controller** which maintains state about the system in order to measure the progress of the execution of the system in relation to the contract. This state may be updated at run-time when the controller receives events. These events are filtered and dispatched by an **event interception** mechanism.

There are various alternative distributed architectural styles for coordination middleware:

**Centralized vs. Decentralized** Both controller and interception mechanism can be implemented in either a centralized or decentralized way. The latter implies that each organization has deployed its own local instance of the coordination middleware, or the middleware is deployed on a nearby node. The latter entails that a trusted third party system governs all interactions between the organizations.

**Reactively vs. Proactively** Monitoring and enforcement can be implemented either reactively or proactively [27]; the system may interact with the cross-organizational processes in two different ways: a **reactive** system intercepts messages exchanged between business processes and reacts by approving or disapproving them; a **proactive** system drives the cross-organizational interaction between the business processes by inviting the partners to send valid messages.

### 3. Mapping of existing coordination architectures

To illustrate the proposed reference model, we have mapped seven well-known coordination architectures to the reference model. These are Ponder [11], Law-Governed Interaction (LGI) [25], the Business Contract Language (BCL) and Architecture (BCA) [24], Crossflow [15, 14], The FSM-based approach from The Tapas project [28], GlueQoS [41] and T-BPEL [39]. As stated above, these mappings enable us to compare the different coordination architecture and also allow to make constructive proposals for improvements. Figure 3 gives a concise overview of the mapping of these coordination architectures to the elements of the reference model. The detailed descriptions of these mappings can be found elsewhere [40]. For the sake of completeness, a summary of the 7 mappings is given in appendix A.

### 4. Conclusions and Future work

This section aims drawing some general conclusions from the mapping of these seven coordination architectures to the reference model. The remainder of this summarizing section is structured as follows. We first discuss the main differences between policy-based and contract-based approaches. Secondly, future directions in the construction of cross-organizational coordination architectures are proposed.

#### 4.1. Differences between policies and electronic contracts

In our opinion three main differences between policies and electronic contracts have emerged from the mapping.
Firstly, policies are focussed on governing business-level interactions and modality constraints (agreement types 1 and 2), while electronic contracts not only support these but also agreements about QoS features and technical protocols (agreement types 3 and 4). Secondly, contract frameworks generally support much more coordination functionalities than policy-based architectures that all exclusively focus on enforcement. Thirdly, the problem of managing policies has been thoroughly studied, whereas management of contracts has been largely ignored in the work on electronic contracts. Especially, the problem of dealing with conflicts between policies has been thoroughly studied, whereas conflicts between electronic contracts have been largely ignored until now. Instead, the focus lies rather on contract validation that detects ambiguities between different clauses in a single contract.

4.2. Directions for future work

Based on the reference model and the mapping exercise, we are able to make some constructive protocols for improving the existing coordination architectures.

Policy and contracts united  First of all, these above observations indicate that contracts and policies relate to each other in major/minor balance (at least when considering cross-organizational coordination). Contract frameworks offer the most complete set of concepts and mechanisms for cross-organizational coordination. Yet policy-driven systems are an important contribution as they offer complementary solutions. This complementarity shows in at least three different ways. First of all, the concepts used for specifying obligations and rights in contracts (Agreement type 2) are very similar to those of policies. As such policy-based enforcement can be leveraged for this type of agreement. Secondly, existing contract-based coordination architecture could be extended with policy-based enforcement. Consider for instance the situation where a policy is dynamically deployed to rectify a contract violation. Thirdly, the mechanisms for policy management can be adapted for management of contracts.

Enactment of agreements  Clearly for agreement types 1 and 2, enactment of agreements is the least understood area of research. As pointed out in the Crossflow project, the mapping of a contract to an existing infrastructure is beyond the current technological state-of-the-art.

Complete coordination architectures  Currently none of the 7 coordination architectures simultaneously support all four agreement types. Either support for agreement type 1 and 2, or 3 and 4 is offered, but never the combination of all four types. Furthermore, none of the these archi-
tectures support all six kinds of coordination middleware functionalities (negotiation, validation, enactment, monitoring, enforcement, dismantling). Especially the dismantling of agreements is not well supported in particular.

**Standardization** Orthogonal to the issue of improving existing coordination middleware, there is also the issue of standardization. As pointed out in section 2, the reference model could potentially be used as an instrument to derive commonly accepted architectural blueprints for coordination middleware. The reference model after all aims to offer a common vocabulary of important problems to be solved. This helps people in standardization bodies to discuss with less ambiguity about the required features for their particular coordination architectures. A highly related project is this vain is the European Virtual Laboratory for Enterprise based on the INTEROP project [1]. In this project an extensive “Interoperability” glossary and taxonomy has been defined. Obviously, the reference model presented in this paper should integrate and be compatible with this knowledge base.

References

A. Summary of mappings

Ponder


1. Agreement type: Ponder [11] allows to capture agreements about *modal constraints* (agreement type 2). Both obligations and authorization policies are supported.

2. Agreement language

(a) Conceptual model: The Ponder language is policy-based.

(b) Computational model: Ponder comes with an approach for deriving implementable policies from high-level goals [3, 4] based on the *event calculus* [17].

(c) Management concepts: Concepts for managing policies include *policy domains* for grouping objects to which the same policy applies and *hierarchical grouping* of related policies. Concepts for policy conflict management include support for dealing with *modality* and *application-specific conflicts*. Automatic detection of policy conflicts is also supported, again by using the event calculus as underlying formalism [4, 10].

(d) Language technology: Ponder is designed as an *object-oriented* language and is statically typed.

3. Coordination middleware

(a) Coordination middleware functionalities: With respect to coordination middleware functionalities, Ponder only supports *enforcement*, primarily enforcement of access control policies. It does not support establishment.

(b) Distributed architecture: Policies are enforced by so-called management agents that impose the policy by *re-actively controlling and adapting the behavior of the underlying application, mostly by intercepting messages*. Management agents [38] are injected into the application in a *decentralized way*: management agents for obligation policies are typically injected at the subject side, while authorizations are enforced by management agents at the target side.

**Law-Governed Interaction**

1. Agreement type: Law-governed Interaction (LGI) [25] can be used to regulate *business interactions* between autonomous organizations or agents (agreement type 1), but it also allows expressing *modal constraints* (agreement type 2).

2. Agreement language

(a) Conceptual model: The basic modeling concept of LGI is the *law*. Abstractly speaking, the concept of a law is in essence the same as a policy.
(b) Computational model: There is no real distinction between the conceptual and the computational model of LGI: A law is directly specified as a function that returns an implementable action for every possible event that might happen at a given software object. These events must directly refer to methods or changes of internal states of software objects. This makes that the law specifications are directly coupled to a specific software system and therefore are brittle to evolution of the software code.

(c) Management concepts: Similar to Ponder, LGI also supports agent grouping and hierarchical grouping of related laws [2].

(d) Language technology: Laws are implemented using Prolog due to its expressive power and its relatively widespread usage. Reflection is also used heavily as an implementation technique.

3. Coordination middleware

(a) Coordination middleware functionalities: LGI does not focus on establishment. With respect to execution, LGI focuses only at policy enforcement.

(b) Distributed architecture: A law is enforced by a set of trusted controllers. These controllers are reactive and for each agent there is a separate local controller (decentralized).

The Business Contract Language and Architecture

1. Agreement type: The Business Contract Language (BCL) [24] is targeted towards regulating business interactions between organizations and allows defining modal constraints (agreement types 1 and 2).

2. Agreement language

(a) Conceptual model: The basic modeling concept of BCL is the notion of community. Communities are in fact contract templates that already contain general contract behavior that is essential, and so cannot be varied, and those parts that can be expected to vary.

(b) Computational model: The behavioral concepts of BCL include event patterns (e.g., Sequence of Events, Disjunction of events and Event causality), internal states and their changes in response to the events, and event types to be created when certain conditions have been matched, e.g. creation of contract violation or contract fulfillment events.

(c) Management concepts: Except support for dealing with application-specific conflicts such as conflict of duty, BCL offers no further management concepts.

(d) Language technology: BCL is implemented using a proprietary language technology.

3. Coordination middleware

(a) Coordination middleware functionalities: As underlying contract framework, a generic Business Contract Architecture (BCA) is proposed. The core of this contract architecture supports contract execution only; contract establishment is delegated to additional components. Thus the core of BCA silently assumes that agreed upon contracts are stored in a contract repository. For contract execution, BCA supports contract monitoring and enforcement only. It does not provide an enactment component by means of which organizations can dynamically create the necessary infrastructure and allocate the necessary resources to exploit a contract.

(b) Distributed architecture: BCA is a flexible architecture; event interception and controller functions can be implemented both centralized or decentralized. The Contract Monitor component performs re-actively evaluations of contracts to determine whether parties’ obligations have been satisfied or whether there are violations to the contract.

Contract-driven creation of virtual enterprises in Crossflow

1. Agreement type: The Crossflow project [15, 14] presents a contract architecture that is mostly focussed on agreement type 1, i.e. regulating business-level interactions between organizations, especially in the context of cross-organizational workflow management.

2. Agreement language:

(a) Conceptual model: Crossflow views contracts as the central theme that runs throughout the enterprises’ life cycle. A contract specifies an abstract workflow for regulating the interactions between the business partners, and procedures for monitoring workflow execution and adjusting the workflow in case of deviations to the contract.

(b) Computational model: In order to turn these contracts into implementable actions, a blueprint for contract enactment (called the Internal Enactment Specification) is prepared, one blueprint for each participating organization.

(c) Management concepts: No additional management concepts are offered.

(d) Language technology: The language technology used in the Crossflow project is XML.
3. Coordination middleware

(a) Coordination middleware functionalities: The focus of Crossflow lies mostly on contract negotiation, contract enactment and contract dismantling. An unusual approach to contract monitoring and enforcement is taken however: contracts themselves may contain clauses for how the behavior of the service provider can be monitored by the service consumer (and vice versa), and what rectifying actions are possible.

(b) Distributed architecture: At the architectural level, the contract framework consists of two building blocks: (1) a Virtual Market (used to find business partners and negotiate contracts dynamically) with as central component a matchmaking engine and (2) an Enactment Infrastructure (needed to enact (i.e., set-up, execute, terminate) the contracted service. Both reactive and proactive contract execution is supported.

Electronic contracts based on FSM's

1. Agreement type: Work as part of The Tapas project [36] focusses on agreement types 1 and 2: business interactions and rights and obligations of the signed organizations.

2. Agreement language

   (a) Conceptual model: Contracts are modelled in Tapas as a set of finite state machines (FSMs), one for each contracted party [28].

   (b) Computational model: Similar to the above Law-Governed Interaction approach there is no real distinction between the conceptual model and the computational model of contracts in Tapas: FSM's are directly linked to implementable actions. The states of the FSM correspond with the progress of the execution or fulfillment of the contract; the input symbols are events that trigger the contract; and the output symbols correspond with the set of obligations and rights that the party is respectively subject to, as well as what actions the party has the right or obligation to perform.

   (c) Management concepts: No additional management concepts are supported.

   (d) Language technology: Contracts are directly specified using the FSM formalism.

3. Coordination middleware

   (a) Coordination middleware functionalities: The work covers the following coordination functionalities: (i) contract validation to detect and remove ambiguities between the clauses of a contract, and (ii) contract monitoring with the focus on capturing non-repudiable evidence of the actions performed by the business parties.

   (b) Distributed architecture: The contract framework is deployed in a de-centralized manner. Furthermore, the architecture is pro-active as the FSMs controls which operations are invoked on the contract objects.

GlueQoS

1. Agreement type: GlueQoS [41] is a middleware-based approach to match, interpret, and mediate QoS requirements of clients and servers. The quality of service requirements are mostly related to QoS features such as security, reliability and performance (agreement type 3).

2. Agreement language

   (a) Conceptual model: The basic modeling concept of GlueQoS is the QoS policy. The QoS policy is essentially a contract that describes what is the combination and configuration of QoS features (e.g. security) that is acceptable for a client and a server system.

   (b) Computational model: For implementing an agreed QoS policy, QoS features are directly implemented using aspects that intercept the necessary events and state changes by means of a generic join point model.

   (c) Management concepts: With respect to management concepts, GlueQoS allows mediating application-specific conflicts between a client and a server organization with respect to the QoS policies.

   (d) Language technology: Client and Server policies are specified using the WS-Policy standard [12].

3. Coordination middleware

   (a) Coordination middleware functionalities: GlueQoS focuses on contract negotiation and contract enactment. It does not focus on contract execution.

   (b) Distributed architecture: When the client requests a session to the server, a QoS policy for that session will be computed at the client-side, by means of a matchmaking engine (decentralized architecture). An agreed QoS policy is enacted by means of deploying aspects. As contract monitoring and enforcement is not the focus of this work, no controller or event interception is necessary.

T-BPEL: Transaction policies for service oriented computing

1. Agreement type: T-BPEL [39], which is related to GlueQoS, supports advertising and negotiating between
different transaction protocols (such as direct transaction processing, queued transaction processing and compensation-based transaction processing). The focus lies thus on agreement type 4.

2. Agreement language

(a) Conceptual model: T-BPEL introduces the concept of transaction coupling mode which is a contract about the used transaction protocol(s) between a web service provider and an orchestrating BPEL business process that invokes the web service. The preferred transaction protocols of the business process and the partner web services are expressed as by so-called policies that are directly attached to them.

(b) Computational model: Client and provider policies as well as transaction coupling modes can be gradually refined from abstract transactional modes to concrete implementable protocols (e.g., WS-ReliableMessaging specification). The BPEL process itself is then transformed into a Java implementation (exposed as a SOAP-based service) that uses the selected transaction protocols.

(c) Management concepts: No additional management concepts are provided, except that conflicts between different policy assertions can be mediated.

(d) Language technology: Client and provider policies are described using the WS-Policy language. [12].

3. Coordination middleware

(a) Coordination middleware functionalities: The underlying contract framework supports (i) validation of client policies, (ii) contract negotiation by means of policy matching, and (iii) contract enactment.

(b) Distributed architecture: The distributed architecture of T-BPEL is decentralized, and very similar to Glue-QoS, the distinction between pro-active or reactive is irrelevant.