Representing Agent Contracts with Exceptions and Business Process Descriptions

Terrence C. Poon

Bachelor of Science in Computer Science and Engineering
Bachelor of Science in Economics
Massachusetts Institute of Technology (2001)

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 24, 2002

Author ____________ ______________
Department of Electrical Engineering and Computer Science
May 24, 2002

Certified by [illegible] 5/24/2002
Benjamin N. Grosof
Thesis Supervisor

Accepted by _________
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Representing Agent Contracts with Exceptions and Business Process Descriptions

Terrence C. Poon

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 24, 2002

Abstract

A key challenge in e-commerce is to specify the terms of the deal between buyers and sellers, e.g., pricing and description of goods/services. Previous work developed an approach that automates such business contracts by representing and communicating them as modular logic-program rules. This thesis presents the SweetDeal automated contracting system, which extends the rule-based contract representation with process knowledge descriptions (e.g. processes, exceptions, handlers) drawn from the MIT Process Handbook, a large previously-existing repository. This enables more complex contracts with behavioral provisions, especially for handling exception conditions that might arise during the execution of the contract.

For example, a contract can first identify possible exceptions like late delivery, nonpayment, and decommitment. Next, it can specify handlers to find or fix these exceptions, such as contingency payments, escrow services, detectors, and notifications.

Our rule-based representation allows software agents in an electronic marketplace to create, evaluate, negotiate, and execute such complex contracts with substantial automation. This thesis defines a software agent that creates contract proposals in a semi-automated manner by combining modular contract provisions from a contract repository with process knowledge from a process repository. Another novel aspect is that SweetDeal encodes the contract provisions using an emerging Semantic Web (XML) standard for knowledge representation of rules (RuleML). Yet another novel aspect is that SweetDeal combines this with an emerging Semantic Web representation for ontologies (DAML+OIL).

Thesis advisor: Benjamin N. Grosof
Title: Assistant Professor of Information Technology, MIT Sloan School of Management

Representing Agent Contracts with Exceptions and Business Process Descriptions

Terrence C. Poon

Contents

Acknowledgements

A thesis is a long journey. The seventeen thousand words in this thesis would be incomplete if I did not include a few more to recognize the many people who helped make it possible.
First I want to thank my thesis advisor, Benjamin Grosof. From inspiring me with initial research ideas, to introducing me to relevant researchers, to making time in his busy schedule to help me refine my thesis, he has given me invaluable guidance and encouragement, even through complicating circumstances.

I would not have found this thesis topic without some wonderful UROP experiences at the MIT Media Lab, first at the Junior Summit project and then at the Software Agents group. These were enjoyable learning opportunities, immersing me in leading areas of research. Indeed, I was first exposed to the concept of agent marketplaces while working for David Wang at the Agents group. One day last year, I was talking to David about potential thesis advisors, and he asked Professor Pattie Maes, the (former) head of the Agents group, for some suggestions. My current thesis advisor is one of the professors she recommended.

For the thesis itself, I am grateful to several researchers at Sloan's Center for eBusiness who helped me tremendously, particularly Mark Klein, John Quimby, George Herman, Chris Dellarocas, and Tom Malone. My research is essentially the intersection of my advisor's work on rule-based contracting with Malone et al's work on the MIT Process Handbook, extended by Klein et al's work on exception handling. I had many fruitful conversations with John Quimby, who demonstrated the Process Handbook software to me and gave me many insights into the details of representing process knowledge. Thanks to my advisor and his RA, Youssef Kabbaj, for their work on SweetRules, which enabled my prototype to output contracts in RuleML. Finally, thanks to Daniel Reeves and Michael Wellman at the University of Michigan for their work on CLP-based contracts, which served as the basis for the contracting language in this thesis.

Thanks to Ana S. Li '02 for being a special person in my life, always understanding and encouraging me throughout this journey. Thanks most of all to my parents, who have supported me since my childhood in Hong Kong, to New Jersey, Houston, and Boston. They have given me the freedom to discover my own passions and the opportunities to pursue them.

A part of the work described here appears in the following paper:

Benjamin N. Grosof and Terrence C. Poon, “Representing Agent Contracts With Exceptions using XML Rules, Ontologies, and Process Descriptions.” To appear in *Proc. International Workshop on Rule Markup Languages for Business Rules on the Semantic Web*, Sardinia, Italy, June 2002.

1 Introduction

Electronic markets, where software agents represent buyers, sellers, and intermediaries, provide automated discovery, negotiation, and execution in the contracting process for goods and services [2]. Most current agent marketplaces only support a simple knowledge representation (KR) for contract terms, specifying fixed values for a set of attributes like price, quantity, or delivery date. Previous work [1][5] introduced a more expressive contract representation using modular logic-program rules that is capable of expressing conditional relationships, where the value of one attribute is inferred from the values of other attributes.

This thesis introduces the SweetDeal system for automated contracting, part of the larger SWEET (Semantic WEb Enabling Technology) effort. SweetDeal extends the rule-based contract representation with process knowledge descriptions drawn from the MIT Process Handbook (PH) [6]. This includes Klein et al [7]'s extension to PH with knowledge about *exceptions*, which are violations of the inter-agent commitments specified in a contract. SweetDeal's contract representation enables more complex contracts with *behavioral* provisions, in particular *exception handling* provisions that manage the exceptions that might arise during the execution of a contract. Exception handling provisions declare possible exceptions for a contract, such as late delivery, nonpayment, fraudulent credit payment, and decommitment, and define corresponding exception handlers, including credit checks, contingency payments, risk payments, escrow services, detectors, and notifications.

Exception handling is particularly important for electronic markets. According to a recent government report [37], top categories of Internet fraud complaints in the United States in 2001 included Internet auction fraud at 42.8%, non-deliverable merchandise and payment at 20.3%, and credit card fraud at 9.4%, with a total reported loss of $17.8 million. The principal advantage of electronic markets over traditional markets is that they can rapidly bring disparate buyers and sellers together to form contracts on an as-needed basis. However, this means that the buyer and seller negotiating a deal often have little or no prior contact with each other. Each party could benefit from contractual protections against misbehavior or outright fraud by the other party. In SweetDeal, these exception handling provisions can be defined in the electronic contract itself and negotiated in the same way as other contract provisions.

Overall, the contracting process consists of three major stages [1]:

1. Discovery: Agents find potential contracting partners.
2. Negotiation: Agents exchange proposals and counterproposals with each other to determine mutually acceptable contract terms. This may involve iterative modification of the terms.
3. Execution: Agents execute the transactions defined by the contract provisions.

SweetDeal automates the contracting process along several dimensions [15]:

1. **Automated contract communication**. Logic program rules provide an expressive, machine-understandable format for representing contracts. Unlike human languages, logic program rules have well-defined, unambiguous semantics. This allows rule-based contracts to be exchanged between software agents (e.g. during negotiation) or Web services with a consistent yet implementation-independent interpretation [39].
2. **Automated contract creation**. This project defines software components called the *process repository* and *contract repository* that store and retrieve business knowledge in machine-understandable formats. SweetDeal enables software agents to create contracts in a semi-automated fashion by combining general knowledge about business processes and exceptions from the process repository with modular provisions from the contract

repository. The system guides the human user through the contract creation process, suggesting relevant contract provisions. (See Section 7.3.)

3. **Automated contract execution**. A software agent executes an automated contract by simply inferencing with the rule-based provisions. In particular, an agent can use inference to execute the exception handlers included in a contract, allowing it to detect and manage exceptions. Inference also generates implied obligations of the contract for particular situations (e.g. a particular penalty amount for a late delivery). In addition, exception handlers can invoke general business processes through attached procedures (see Section 2.3.1).

In addition, this thesis provides a foundation for the following dimensions of automation that are not directly addressed:

4. **Automated contract evaluation.** Since a software agent can identify the logical implications of rule-based contract provisions through inferencing, it can evaluate the utility of a contract according to some set of preferences (i.e. a utility function). Evaluation is especially important during the negotiation stage, where agents repeatedly compare alternate contract proposals.
5. **Automated contract negotiation.** The primary challenge for automated negotiation is in the generation of counterproposals. Automated contract evaluation provides a basis for a simple generate-and-test approach, where an agent considers alternate counterproposal terms and picks the one with highest utility. More sophisticated approaches could search the contract space in a more intelligent manner and take strategic considerations (i.e. reactions from the other party) into account.
6. **Automated discovery.** In addition to contract provisions, the rule-based representation may be used to specify market agent profiles. This facilitates the matchmaking process in which market agents look for contracting partners with the desired characteristics.

This thesis makes novel contributions in several areas:

- Represents process knowledge from the MIT Process Handbook using an emerging Semantic Web ontology KR (DAML+OIL) and defines a *process repository* that allows agents to query the process knowledge.
- Demonstrates a limitation of DAML+OIL in representing inheritance overrides.
- Extends an existing approach to rule-based representation of contracts with the ability to reference process knowledge and include exception handling mechanisms.
- Defines a corresponding contract representation based on an emerging Semantic Web rule KR (RuleML).
- This is, to our knowledge, the first time that RuleML has been combined with DAML+OIL for a substantial business application or domain situation/purpose.
- Designed an implementation of a mechanism for rule-based contracts in RuleML to be built from reusable modular provisions, called *contract fragments*, that are retrieved from *contract repositories*.
- Designed an implementation of the mechanisms of a *market agent* that largely automates the creation of such contracts as part of a negotiation process, in support of a human user.
- Provides an overall interaction architecture for an agent marketplace with such rule-based contracts.

Here we give an overview of the rest of this thesis. Chapter 2 describes some background research for this project. Chapter 3 gives an overview of the major contracting concepts and system components in SweetDeal. In Chapter 4, we use DAML+OIL to encode the PH process knowledge and demonstrate its inability to represent inheritance overrides. In addition, we describe the process repository interface for querying the process knowledge. Chapter 5 introduces our rule-based contracting language, capable of referencing process knowledge and defining exception handling provisions. Chapter 6 extends the contracting language with modular provisions called contract fragments. In addition, it describes contract repositories, which store and retrieve contract fragments. Chapter 7 explains the mechanisms of a market agent that creates contract proposals in a semi-automated fashion. In Chapter 8, we sketch some details in the implementation of SweetDeal. Finally, in Chapter 9, we summarize the contributions of this thesis and suggest some future research directions.

2 Background

This project builds upon several areas of prior research:

- Agent-based marketplaces
- Modular contract provisions
- Business rules and rule-based contracts
- Business process knowledge
- Semantic Web technologies: RuleML and DAML+OIL

2.1 Agent-Based Marketplaces

One established mechanism for automated electronic commerce involves intelligent software agents that represent the desires and resources of buyers and sellers [2]. These agents meet in electronic marketplaces and form contracts for goods and services [1]. Most current agent marketplaces only support a relatively simple KR for contract terms, specifying values for a predefined set of attributes such as price, quantity, and delivery date (see [2] for a review). Many real-world contracts require more complex terms [1] involving conditional relationships, where the value of one attribute in the contract depends on the values of other attributes. These conditional relationships can be conveniently expressed as business rules.

2.2 Modular Contract Provisions

Modular contract provisions are provisions specified in a generalized form so that they may be re-used in multiple contracts. Modular contract provisions are similar to boilerplate terms, except that they are simply *referenced* by the contract rather than copied into the body of the contract. CommonAccord [27] is a recently formed lawyer-world organization that advocates the use of modular contract provisions called *c-Terms* ("common terms") for *non-automated* contracts.

They are written in HTML-formatted legalese and hosted on the CommonAccord website. A contract incorporates *c-Terms* by referencing their URLs.

This thesis introduces a syntax and mechanism for modular contract provisions in *automated* contracts, called *contract fragments* (see Chapter 6).

2.3 Business Rules and Rule-Based Contracts

2.3.1 Business Rules and Situated Courteous Logic Programs

A *business rule* is an if-then rule used to describe some piece of business logic. Formally, it is an implication from an antecedent (IF clause) to a conclusion (THEN clause) in which the antecedent may contain multiple conjoined (AND'ed) conditions [1]. A rule with only a conclusion but no antecedent is called a *fact*. Rules can be used to describe terms and conditions such as volume discounts, service provisions for refunds and other exceptional conditions, and requirements for surrounding business processes like the lead time to place an order. Consider the following example of volume discounting:

- (Rule A) If the buyer purchases between 50 and 100 units and accepts delivery in 5 to 10 days, then the price is $10 per unit.
- (Rule B) If the buyer purchases between 80 and 150 units and accepts delivery in 8 to 15 days, then the price is $8 per unit.
- (Rule C) If the buyer is a preferred customer, then the price is $7 per unit, regardless of the quantity or delivery date.
- (Priority Rule 1) If both A and B apply, then Rule B 'wins', i.e. the price is $8.
- (Priority Rule 2) If both A and C apply, then Rule C wins, so the price is $7.

In addition to their expressiveness to human readers, contracts specified with business rules can be automatically evaluated, modified, and executed by software agents.

One language for encoding business rules is called Courteous Logic Programs (CLP) [1]. CLP is an extension of Ordinary Logic Programs, a well-established language in artificial intelligence for knowledge representation [3]. CLP provides the additional mechanism of prioritized conflict handling, in which conflicting rules are resolved through pairwise mutual exclusion (mutex) statements and priorities between rules. This mechanism allows one rule to be overridden by another rule with higher priority. Rules may be given higher priority because they specify special cases, come from higher-authority sources, or have been updated more recently. In particular, contract terms can be modified during negotiation by adding higher-priority rules.

CLP rules may be encoded in several formats. The SCLPfile format is a straightforward text format for CLP. "<-" stands for implication (i.e. "if"), "?" indicates a logical variable, and ";" ends a rule statement. "<...>" encloses a rule label, and "//" prefixes a comment line. The following is an SCLPfile encoding of the previous example:

```
<A> price(?Order,10) <-
  quantity(?Order,?Q) AND greaterThanOrEquals(?Q,50) AND lessThanOrEquals(?Q,100) AND
  deliveryDate(?Order,?D) AND greaterThanOrEquals(?D,5) AND lessThanOrEquals(?D,10) ;

<B> price(?Order,8) <-
  quantity(?Order,?Q) AND greaterThanOrEquals(?Q,80) AND lessThanOrEquals(?Q,150) AND
  deliveryDate(?Order,?D) AND greaterThanOrEquals(?D,8) AND lessThanOrEquals(?D,15) ;

<C> price(?Order,7) <- buyer(?Order,?Buyer) AND seller(?Order,?Seller) AND
  customerType(?Buyer,?Seller,preferred) ;

<priority1> overrides(B,A) ;
<priority2> overrides(C,A) ;

MUTEX price(?Order,?X) AND price(?Order,?Y)
  GIVEN notEquals(?X,?Y) ;
```

Figure 2-1: Pricing rules

The `MUTEX` statement says that there can only be one price for every order. If rules A and B both apply during execution, then the priority rule `overrides(B,A)` is used to decide whether to set the price to 10 or 8. Since rule B overrides rule A, the price would be set to 8.

An important extension to CLP is the ability to express procedural attachments, resulting in *situated* courteous logic programs (SCLP) [1][12]. This allows belief expressions in the rule system to be associated with procedure calls in a programming language like Java. There are two kinds of procedural attachments. *Sensors* test for antecedent conditions using an attached procedure. For example, if the `customerType` predicate is associated with the Java method `CustomerManager.getCustomerType`, then the rule engine will call that method to obtain a value for `customerType`. *Effectors* use an attached procedure to perform actions when a consequent condition is concluded. For example, if the `price` predicate is associated with the Java method `Order.setPrice`, then `Order.setPrice` will be executed when the rule engine infers a conclusion for *price.*

XML representation facilitates knowledge interchange on the Web. Previously, the Business Rules Markup Language (BRML) [1] provided an XML encoding for SCLP rules. However, the emerging RuleML language [10], which is partly based on the design approach and criteria of BRML, is now the preferred XML embodiment for SCLP rules [43]. RuleML is described in Section 2.5.1.

The IBM CommonRules rule engine [13] supports inferencing with SCLP rules. SCLP has been used in several major applications, including EECOMS, a three-year industry consortium effort led by IBM that focused on supply chain integration for manufacturing [4]. The project used SCLP to encode rules for supply chain processes, such as ordering lead time.

2.3.2 Rule-Based Contracting Using SCLP

ContractBot [5] presents a SCLP-based contracting language that automates the negotiation of business contracts. The language is used to represent fully-specified executable contracts as well as partial contracts, or contract templates, that are in the process of being negotiated. ContractBot creates a complete, executable contract by combining the rules in the contract template with negotiated values for contract parameters such as price, quantity, and delivery date. The Michigan Internet AuctionBot applies this contracting language toward automated auctions, using SCLP rules to define auction structures, auction parameters, and the domain-specific constraints, preferences, and capabilities of buyers and sellers. AuctionBot has been used in the semi-realistic domain of a Trading Agent Competition about travel packages [34].

2.4 Business Process Knowledge

2.4.1 MIT Process Handbook

The MIT Process Handbook [6] is a knowledge repository that describes and classifies major business processes using the organizational concepts of *decomposition*, *dependencies*, and *specialization.* The Handbook models each *process* as a collection of activities that can be decomposed into sub-activities, which may themselves be processes. In turn, coordination is modeled as the management of *dependencies* that represent flows of control, data, or material between activities. Each dependency is managed by a *coordination mechanism*, which is the process that controls its resource flow. Finally, processes are arranged into a taxonomy, with generic processes at the top and increasingly specialized processes underneath. Each specialization automatically inherits the properties of its parents. However, it may use *property overrides* to explicitly remove an inherited property. Figure 2-2 shows a part of the taxonomy with some specializations for the “Sell” process.

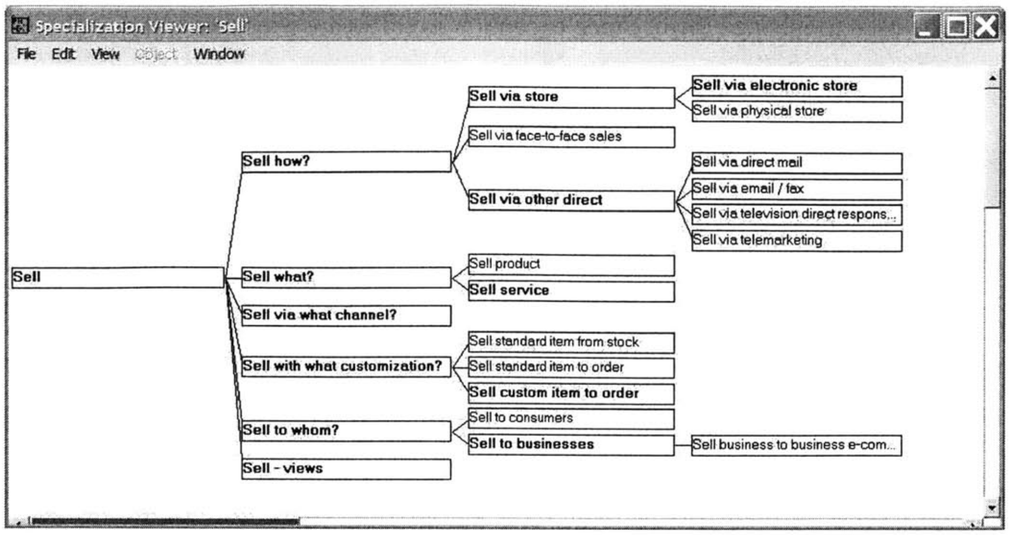

Specialization Viewer: 'Sell'
File Edit View Object Window
Sell
Sell how?
Sell via store
Sell via electronic store
Sell via physical store
Sell via face-to-face sales
Sell via other direct
Sell via direct mail
Sell via email / fax
Sell via television direct respons...
Sell via telemarketing
Sell what?
Sell product
Sell service
Sell via what channel?
Sell with what customization?
Sell standard item from stock
Sell standard item to order
Sell custom item to order
Sell to whom?
Sell to consumers
Sell to businesses
Sell business to business e-com...
Sell - views

Figure 2-2: Some specializations of "Sell" in the MIT Process Handbook

2.4.2 Exception Conditions

The terms of a contract establish a set of commitments between the parties involved for the execution of that contract. When a contract is executed, these commitments are sometimes violated. [7] considers these violations to be coordination failures, or *exceptions*, and introduces the concept of *exception handlers*, which are processes that manage particular exceptions.

Consider the following example. Company A agrees to pay \$100 per unit for 200 units of company B's product, and B agrees to ship within 15 days (commitments). However, due to unforeseen circumstances, B only manages to ship in 20 days (exception). B pays \$600 to A as compensation for the delay (exception handler).

[7] extends the MIT Process Handbook with an exception taxonomy. Every process is mapped (using a *hasException* link) to its exceptions, which are the characteristic ways in which its commitments may be violated. Like processes, exceptions are arranged in a specialization hierarchy, with generic exceptions on top and more specialized exceptions underneath (see Figure 2-3). Finally, each exception is mapped (using an *isHandledBy* link) to the processes (*exception*

handlers) that can be used to manage them. Since exception handlers are processes, they are arranged in a hierarchy (see Figure 2-4) and may have their own exceptions.

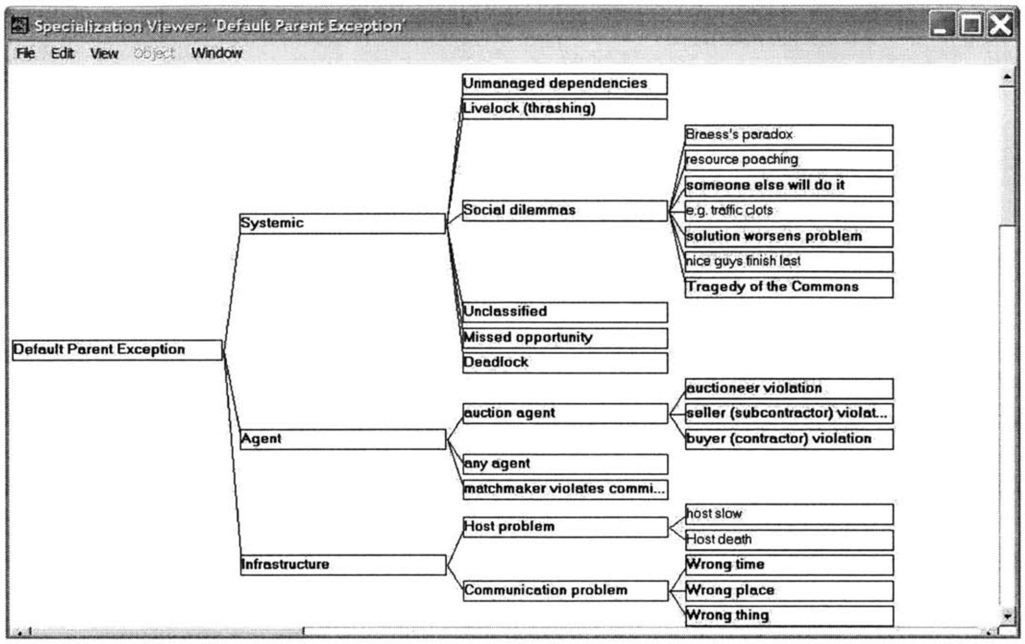


Figure 2-3: Some exceptions in the MIT Process Handbook

There are four kinds of exception handlers [7]. For an exception that has not occurred yet, we can use:

- Exception *anticipation* processes, which identify situations where the exception is likely to occur.
- Exception *avoidance* processes, which decrease or eliminate the likelihood of the exception.

For an exception that has already occurred, we can use:

- Exception *detection* processes, which detect when the exception has actually occurred.
- Exception *resolution* processes, which resolve the exception once it has occurred.

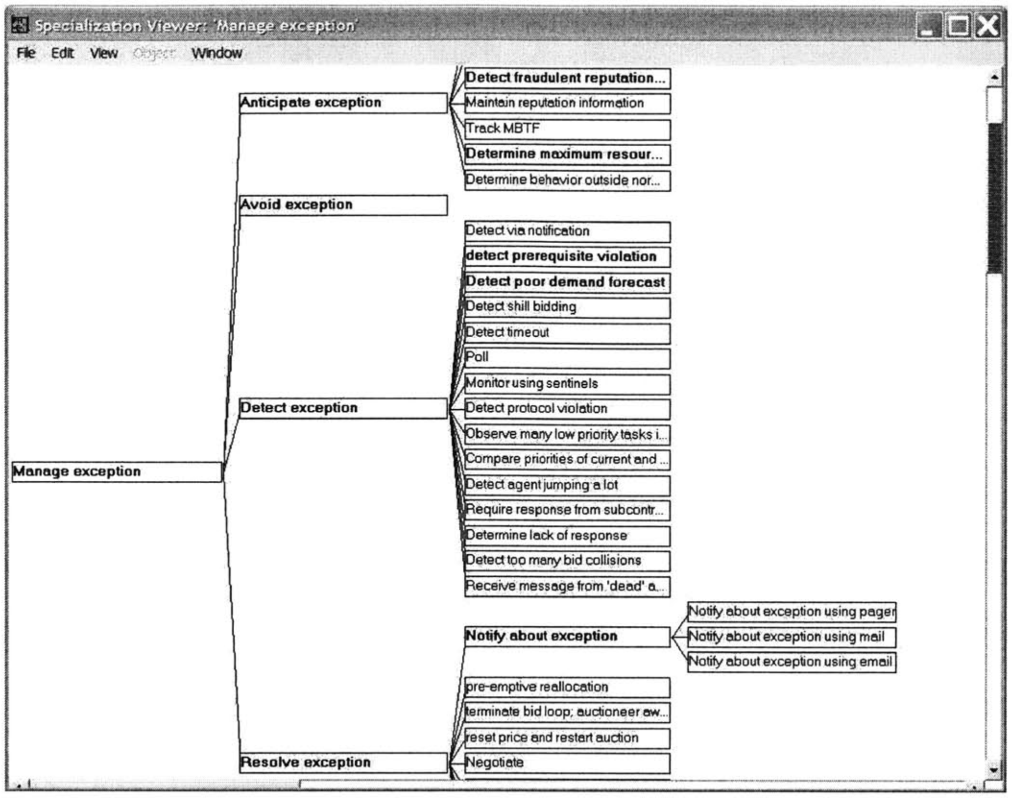

Specialization Viewer: 'Manage exception'
File Edit View Object Window
Manage exception
Anticipate exception
Avoid exception
Detect exception
Resolve exception
Detect fraudulent reputation...
Maintain reputation information
Track MBTF
Determine maximum resour...
Determine behavior outside nor...
Detect via notification
detect prerequisite violation
Detect poor demand forecast
Detect shill bidding
Detect timeout
Poll
Monitor using sentinels
Detect protocol violation
Observe many low priority tasks i...
Compare priorities of current and ...
Detect agent jumping a lot
Require response from subcontr...
Determine lack of response
Detect too many bid collisions
Receive message from 'dead' a...
Notify about exception
Notify about exception using pager
Notify about exception using mail
Notify about exception using email
pre-emptive reallocation
terminate bid loop; auctioneer aw...
reset price and restart auction
Negotiate

Figure 2-4: Some exception handlers in the MIT Process Handbook

Figure 2-5 summarizes the main concepts and relations in the MIT Process Handbook taxonomy. Note that the exceptions associated with a process are inherited by its specializations unless explicitly overridden. Similarly, the handlers for an exception are inherited by the specializations of that exception, unless the property is overridden.

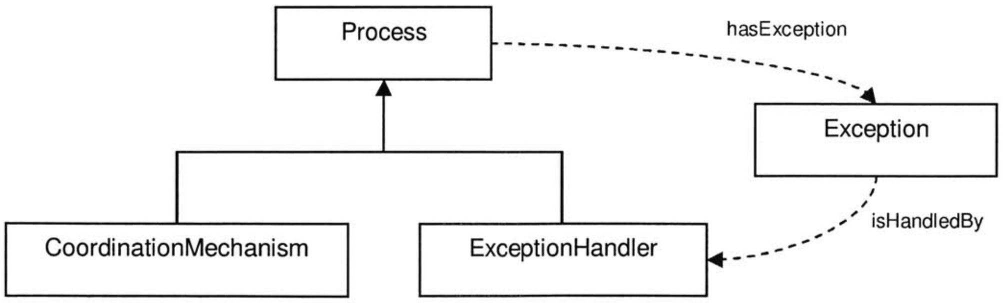


Figure 2-5: Entity relationship diagram for the augmented MIT Process Handbook taxonomy

2.4.3 Process Specialization Versus Object Specialization

The concept of process specialization used by the MIT Process Handbook is similar to the traditional concept of object specialization, used by object-oriented programming languages such as Java as well as many ontology languages. However, there is one apparent inconsistency. A process that overrides a property inherited from its parent could result in *property deletion*, which is prohibited in object specialization. Consider a scenario where handler H1 has a *hasException* link to exception E, handler H2 is a specialization of H1, and H2 overrides the *hasException(E)* property inherited from H1. Then H2 does not have a *hasException* link to E, so that property seems to have been deleted. [8] shows that this is entirely consistent with the notion of subtyping in object specialization. Subtyping restricts the type of a property, reducing its set of permissible values. Similarly, specialization by deletion removes some elements from the set of permissible values for a property. By overriding the *hasException(E)* property, H2 is deleting the value "E" from the set of permissible values for the *hasException* attribute. Therefore, the apparent property deletion in process specialization is conceptually compatible with object specialization. However, as we will see in Section 4.1.3, this difference does lead to some practical difficulties.

2.5 Semantic Web

The Semantic Web [30] is an effort to extend the current World Wide Web by describing the *meaning* of information in well-defined, machine-understandable formats that facilitate program-to-program communication with high-level shared semantics. Two Semantic Web technologies relevant to this thesis are RuleML, an XML language for logic program rules, and DAML+OIL, an XML language for representing ontologies.

2.5.1 RuleML

RuleML (Rule Markup Language) is an early-phase initiative to create a standard language for exchanging rules in XML [10]. RuleML is based on ordinary logic programs (i.e. Horn logic programs extended with negation [3]), extended by the prioritized conflict handling and procedural attachment features of SCLP [43] as well as other expressive features like equivalences, equations, and rewriting. Notably, RuleML allows URIs[1] to be used as names for local vocabulary and knowledge subsets, such as predicates, functions, and rules. This facilitates integration with emerging standards for ontologies on the Web, such as RDF/RDFS and DAML+OIL. As previously mentioned, we expect RuleML to become the preferred XML encoding for SCLP rules.

```
<imp>
   <head>
      <atom>
         <_opr><rel>price</rel></_opr>
         <var>Order</var>
         <ind>10</ind>
      </atom>
   </head>
   <body>
      <and>
         <atom>
            <_opr><rel>quantity</rel></_opr>
            <var>Order</var>
            <var>Q</var>
         </atom>
         <atom>
            <_opr><rel>greaterThanOrEquals</rel></_opr>
            <var>Q</var>
            <ind>50</ind>
         </atom>
         <atom>
            <_opr><rel>lessThanOrEquals</rel></_opr>
            <var>Q</var>
            <ind>100</ind>
         </atom>
         <atom>
            <_opr><rel>deliveryDate</rel></_opr>
            <var>Order</var>
            <var>D</var>
         </atom>
         <atom>
            <_opr><rel>greaterThanOrEquals</rel></_opr>
            <var>D</var>
            <ind>5</ind>
         </atom>
         <atom>
            <_opr><rel>lessThanOrEquals</rel></_opr>
            <var>D</var>
            <ind>10</ind>
         </atom>
      </and>
   </body>
</imp>
```

Figure 2-6: RuleML encoding of the first rule from Figure 2-1

In Figure 2-6, we encode the first rule from Figure 2-1 using the Version 0.8 schema of RuleML. As shown, the RuleML format is quite verbose. This is the typical tradeoff made by XML, favoring self-describing capability and interoperability over compactness.

[1] Uniform Resource Identifiers [31], a standard for naming and addressing Web resources

2.5.2 DAML+OIL (Web Ontology Language)

An ontology "defines the terms used to describe and represent an area of knowledge" [14]. Each ontology consists of classes that represent general concepts in the domains of interest, the relationships that can exist among these classes, and the properties that these classes may have. Ontologies are used extensively in knowledge management. They can represent the semantics of documents in a well-defined format that may be used by web applications and intelligent software agents.

DAML+OIL [18] is a language for creating ontologies and marking up information in a machine readable and understandable format. It originated from two related efforts, DARPA Agent Markup Language (DAML) [16] and Ontology Inferencing Language (OIL) [17]. DAML+OIL is based on RDF, an XML language that represents metadata about Web resources [20].

In August 2001, the World Wide Web Consortium created a working group to define a standard Web ontology language [19]. DAML+OIL is the main technical point of departure for this work.

3 System Overview

The SweetDeal system automates the contracting process for rule-based contracts with process knowledge. In addition to defining a contract representation that is used throughout the negotiation and execution stages of contracting, this thesis introduces mechanisms that automate the creation of contract proposals. In this chapter, we first introduce the simple one-buyer/one-seller negotiation process that is considered in this thesis. Next, we present the concepts used in defining SweetDeal contracts. Finally, we illustrate the overall architecture of SweetDeal and describe the major system components, including the process repository, the contract repository, and the market agent.

3.1 Typical Negotiation Process

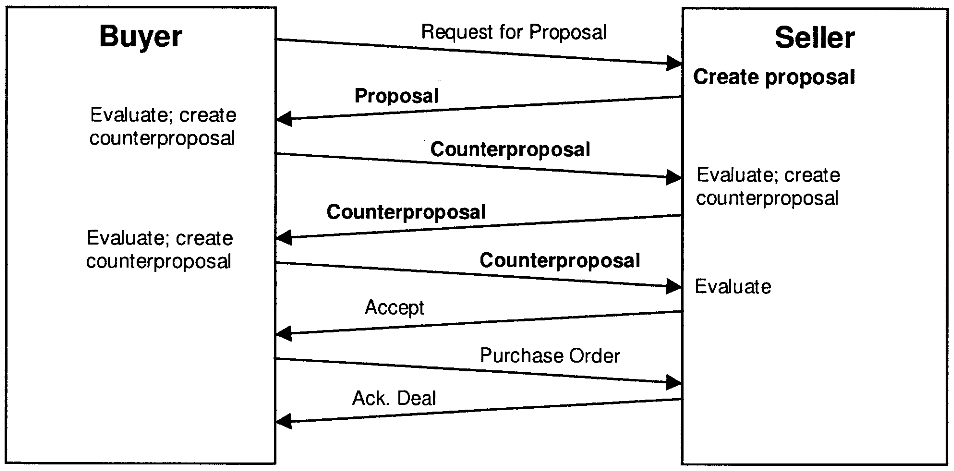


Figure 3-1: Typical negotiation process: *BuyUsingBilateralNegotiation*

We consider a typical negotiation between one buyer and one seller [15], as shown in Figure 3-1. The buyer initiates the process by sending a Request For Proposal (RFP) to the seller. The seller responds with an initial proposal. If the buyer is unsatisfied with the terms in the proposal, it may add some modifications and send back a counterproposal. The seller may respond to this with another counterproposal. In general, this sequence of counterproposals continues until one party responds with an "accept" or "reject" message. (Alternatively, the process may end if it exceeds the time constraints defined by the negotiation protocol.) If the proposal is accepted, it becomes a contract, and the buyer sends a Purchase Order. Finally, the seller responds with an acknowledgement of the deal, and the negotiation phase is complete. In the execution phase, the parties carry out the provisions specified in the contract. In particular, if any exceptions occur, the parties will react according to the exception handling provisions in the contract.

SweetDeal gives an approach for a SCLP-based contracting language that is used throughout the negotiation and execution stages of the contracting process. During negotiation, this language is used to represent all the stages of the contract, including the initial proposal, the intermediate counterproposals, and the final agreement. During execution, each market agent carries out the contract by inferencing on the rules in the final agreement. In addition to the contracting language, SweetDeal specifies mechanisms that automate the creation of such contract proposals by combining process knowledge from the process repository with modular provisions from the contract repository.

3.2 Contracting Concepts

Here we introduce some concepts used in defining rule-based contracts with process knowledge. Many of these contracting concepts build upon entities from the MIT Process Handbook (PH). Every PH entity is a *class* – it refers to an abstract type or category rather than a specific object or individual. For example, *SubcontractorIsLate* is a category that encompasses all the exception conditions where the subcontractor (i.e. seller) is late. Many contracting concepts draw upon *instances* of these classes. For example, an exception instance refers to a *possible* occurrence of the exception class for a particular contract.

A *contract* is an agreement between two or more parties to act according to a set of provisions. We can view a contract as a specification for one or more processes, detailing which party does what when [9]. These processes are instances of process classes from the MIT Process Handbook. In this thesis, we focus on the simple scenario where each contract specifies a *single* process that is an instance of *BuyUsingBilateralNegotiation*, the process class illustrated in Section 3.1.

In addition to attributes like product, price, quantity, and delivery date, contract terms may include exception handling provisions. First, a contract can declare a number of *exception instances* that could potentially occur during its execution. These are instances of various PH exception classes, such as *SubcontractorIsLate*. Next, a contract can include *exception handler instances* that handle particular exception instances. An exception handler instance consists of rules that implement the functionality of some exception handler class, such as *PenalizeForContingency*.

Finally, a *contract template* is a partial contract whose provisions serve as an incomplete specification of a process. To create a contract, a market agent could start with a contract template that contains some basic provisions and add custom provisions that are specific to the purchase being considered.

3.3 Overall Architecture

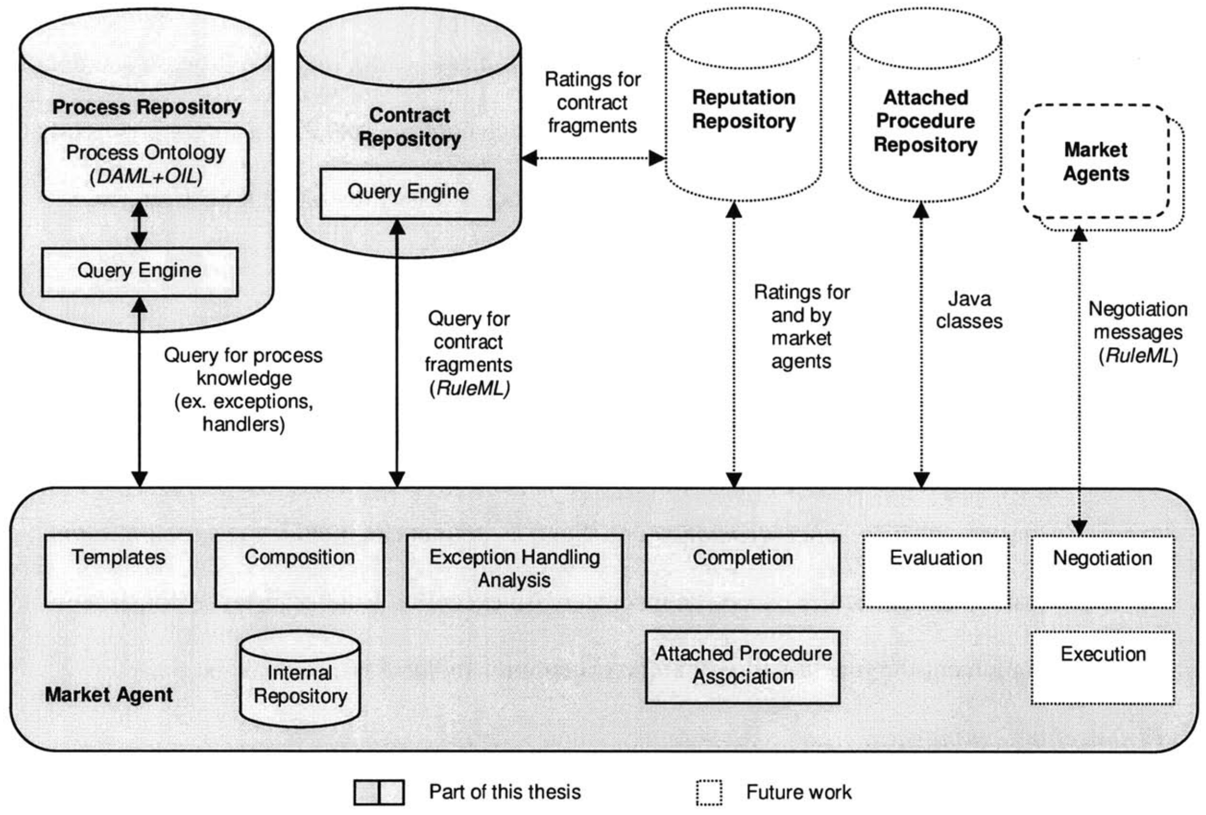

Process Repository
Process Ontology (DAML+OIL)
Query Engine
Contract Repository
Query Engine
Ratings for contract fragments
Reputation Repository
Attached Procedure Repository
Market Agents
Query for process knowledge (ex. exceptions, handlers)
Query for contract fragments (RuleML)
Ratings for and by market agents
Java classes
Negotiation messages (RuleML)
Templates
Composition
Exception Handling Analysis
Completion
Evaluation
Negotiation
Internal Repository
Attached Procedure Association
Execution
Market Agent
Part of this thesis
Future work

Figure 3-2: Overall architecture of the SweetDeal system

Figure 3-2 shows the components of the SweetDeal system. The shaded portions are part of this thesis and are described in later chapters. The unshaded portions with dotted outlines represent future work that can build upon the foundations defined by this thesis.

Each *market agent* represents a buyer, seller, or intermediary (such as an auctioneer) in the marketplace. During the negotiation phase, these agents exchange negotiation messages with contract proposals and counterproposals encoded in RuleML. Chapter 1 describes the components inside each market agent. Market agents make use of several repositories of business knowledge. A *process repository* provides knowledge about processes, exceptions, and exception handlers, as well as the relationships between them. A *contract repository* provides contract fragments, which are modular contract provisions.

Two system components in the diagram are outside the scope of this thesis. A *reputation repository* could provide reputation ratings for market agents as well as contract fragments. This helps an agent make intelligent choices when creating a contract. An *attached procedure repository* provides Java classes whose methods implement attached procedures (i.e. sensors and effectors). This allows a market agent to download implementations of the attached procedures that are used in a contract.

In our current prototype implementation, the system components communicate with each other via local Java calls. However, our design makes it relatively straightforward to extend the system to communicate using SOAP[1] messages over HTTP instead (see Section 8.4 for more details). This would allow the system components to be maintained separately (e.g. by different organizations) and located anywhere on the Internet. It addition, this would make it possible for other software systems to remotely access the SweetDeal system components, for example to retrieve process knowledge from the process repository.

[1] Simple Object Access Protocol [36] is an emerging Web Services standard for exchanging XML-based messages, including remote procedure calls.

4 Process Repository

We view a contract as the *specification* for a process and contract design as the *configuration* of that process [9]. As such, process knowledge plays a critical role in contract creation.

A process repository maintains process knowledge and provides an interface for agents to query this knowledge. In Section 4.1, we describe an approach for representing the process knowledge in the MIT Process Handbook (PH) as a DAML+OIL ontology. In Section 4.2, we describe the query interface of the process repository.

4.1 Representing Process Knowledge

We use the DAML+OIL language to define the ontology for the process knowledge in the MIT Process Handbook. There is a natural mapping between PH concepts and DAML+OIL classes.

4.1.1 Advantages of Using DAML+OIL

Before describing the representation, we consider the choice of DAML+OIL as the language for expressing the ontology of process knowledge. DAML+OIL provides several benefits:

- There are many tools in varying stages of development to parse, manipulate, and query DAML+OIL ontologies [22][23][24]. This helps us avoid creating custom tools for standard tasks. For example, we use the Jena toolkit from HP Labs [23] to parse DAML+OIL ontologies, programmatically traverse their resources, and make simple inferences (see Chapter 8).
- DAML+OIL is thoroughly Web-enabled. In particular, each DAML+OIL resource (e.g. class or property) is named by a globally-unique URI[1]. As a result, the resource can be

[1] There are *anonymous* resources in DAML+OIL, such as Restriction elements, which do not have a URI. They only serve to modify named resources and do not need to be referenced externally.

referenced unambiguously from other XML documents on the Web, which is a key semantic capability. As we will see in Chapter 5, this allows contracts to refer to classes (e.g. processes, exceptions, handlers) in the process ontology.

- DAML+OIL also supports ontology sharing. This allows an ontology to define its vocabulary terms by extending terms from other ontologies. In Chapter 5, we define a contract ontology that extends the process ontology presented in this chapter.
- Most importantly, DAML+OIL is the basis for the Web Ontology Language Working Group at the World Wide Web Consortium, which is expected to define *the* standard for representing ontologies on the web. By using DAML+OIL, we position ourselves to take advantage of future research and tools in this area.

4.1.2 Representing the MIT Process Handbook Ontology in DAML+OIL

In this section, we show how to represent some of the process knowledge in the MIT Process Handbook as a DAML+OIL ontology at `http://xmlcontracting.org/pr.daml`, where "pr" stands for process. (The ontology file is actually stored locally in our current prototype. However, our research group has registered the `xmlcontracting.org` domain and intend to make it a public resource for contracting-related ontologies as well as contract fragments – see Chapter 6.)

Before proceeding, we include some DAML+OIL header statements[1]:

```
<?xml version="1.0" ?>
<rdf:RDF
  xmlns:rdf ="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
  xmlns:xsd ="http://www.w3.org/2000/10/XMLSchema#"
  xmlns     ="http://xmlcontracting.org/pr.daml#"
>

<daml:Ontology rdf:about="">
  <rdfs:comment>
An ontology of some process knowledge from the MIT Process Handbook.
  </rdfs:comment>
  <daml:imports rdf:resource="http://www.daml.org/2001/03/daml+oil"/>
</daml:Ontology>
```

[1] Here `rdf:about=""` means that the `Ontology` element describes the current document.

We define the main concepts as top-level classes (recall Figure 2-5):

```
<daml:Class rdf:ID="Process">
  <rdfs:comment>A process</rdfs:comment>
</daml:Class>

<daml:Class rdf:ID="CoordinationMechanism">
  <rdfs:comment>A process that manages activities between multiple agents</rdfs:comment>
</daml:Class>

<daml:Class rdf:ID="Exception">
  <rdfs:comment>A violation of an inter-agent commitment</rdfs:comment>
</daml:Class>

<daml:Class rdf:ID="ExceptionHandler">
  <rdfs:subClassOf rdf:resource="#Process"/>
  <rdfs:comment>A process that deals with a particular exception</rdfs:comment>
</daml:Class>
```

Next we define the relations between concepts as object properties:

```
<daml:ObjectProperty rdf:ID="hasException">
  <rdfs:domain rdf:resource="#Process" />
  <rdfs:range rdf:resource="#Exception" />
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="isHandledBy">
  <rdfs:domain rdf:resource="#Exception" />
  <rdfs:range rdf:resource="#ExceptionHandler" />
</daml:ObjectProperty>
```

Specializations are expressed as subclasses:

```
<daml:Class rdf:ID="SystemCommitmentViolation">
  <rdfs:subClassOf rdf:resource="#Exception"/>
  <rdfs:comment>
Violations of commitments made by the system operator to create an
environment well-suited to the task at hand.
  </rdfs:comment>
</daml:Class>

<daml:Class rdf:ID="AgentCommitmentViolation">
  <rdfs:subClassOf rdf:resource="#Exception"/>
  <rdfs:comment>
Violations of commitments agents make to each other.
  </rdfs:comment>
</daml:Class>

...
```

This is a natural representation. The Process Handbook expects each specialization to inherit the properties of its parent, and subclasses provide this automatically.

We represent a particular relationship between concepts as a `daml:hasClass` restriction. Consider the following fragment, which defines the exception *ContractorDoesNotPay* and specifies an *isHandledBy* link to the exception handler *ProvideSafeExchangeProtocols*.

```
<daml:Class rdf:ID="ContractorDoesNotPay">
  <rdfs:subClassOf rdf:resource="#ContractorViolation"/>
  <rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#isHandledBy"/>
      <daml:hasClass rdf:resource="#ProvideSafeExchangeProtocols"/>
    </daml:Restriction>
  </rdfs:subClassOf>
</daml:Class>
```

Formally, the `daml:Restriction` element here defines an anonymous class consisting of all things whose *isHandledBy* property could have values of class *ProvideSafeExchangeProtocols*. As a subclass of both *ContractorViolation* and this restriction, *ContractorDoesNotPay* is a specialization of *ContractorViolation* that could be handled by the class *ProvideSafeExchangeProtocols*.

Notice we use a `daml:hasClass` restriction here rather than a `daml:toClass` restriction[1] [18]. `daml:toClass` would require that the value of the *isHandledBy* property for all instances of *ContractorDoesNotPay must* be of the class *ProvideSafeExchangeProtocols*. In other words, it would exclude the possibility for any other exception handler class to handle a *ContractorDoesNotPay* exception. In contrast, `daml:hasClass` leaves open this possibility. This matches the semantics of the *isHandledBy* link in the MIT Process Handbook, which is that some instances of the *ContractorDoesNotPay* exception are known to be aptly handled by some instances of the *ProvideSafeExchangeProtocols* handler. The Handbook takes the approach (which we endorse) that it is typically desirable to treat a process repository as potentially extensible, i.e. open. Indeed, it is often unrealistic to expect a repository to provide an exhaustive listing of all the handlers for a given exception.

Now consider *FradulentCreditPayment*, a specialization of *ContractorDoesNotPay*:

```
<daml:Class rdf:ID="FradulentCreditPayment">
  <rdfs:subClassOf rdf:resource="#ContractorDoesNotPay"/>
</daml:Class>
```

---

[1] The `daml:toClass` restriction is analogous to the universal (for-all) quantifier of predicate logic, while the `daml:hasClass` restriction is analogous to the existential (there-exists) quantifier of predicate logic.

As a subclass, *FradulentCreditPayment* automatically inherits the properties of *ContractorDoesNotPay*, so we conclude that *FradulentCreditPayment* could also be handled by *ProvideSafeExchangeProtocols*. Now consider that the *ProvideSafeExchangeProtocols* handler has *ProvideEscrowService* as a specialization:

```
<daml:Class rdf:ID="ProvideEscrowService">
  <rdfs:subClassOf rdf:resource="#ProvideSafeExchangeProtocols"/>
</daml:Class>
```

We can then infer that exception instances of either *ContractorDoesNotPay* or *FradulentCreditPayment* could be handled by instances of *ProvideSafeExchangeProtocols* as well as instances of *ProvideEscrowService.* This demonstrates the representational power of specialization for organizing process knowledge.

4.1.3 Issue: Property Inheritance Overrides

PH allows *overrides*, where a specialization explicitly omits a property inherited from its generalization (see Section 2.4.3). This is impossible to express directly in DAML+OIL, due to the logical monotonicity of its class system. Consider the following naïve attempt at omitting the *isHandledBy ProvideSafeExchangeProtocols* property from *SpecialNonpayment*, a fictitious specialization of *ContractorDoesNotPay*:

```
<daml:Class rdf:ID="SpecialNonPaymentViolation">
  <rdfs:subClassOf rdf:resource="#ContractorDoesNotPay"/>
  <rdfs:subClassOf>
    <daml:Class>
      <daml:complementOf>
        <daml:Restriction>
          <daml:onProperty rdf:resource="#isHandledBy"/>
          <daml:hasClass rdf:resource="#ProvideSafeExchangeProtocols"/>
        </daml:Restriction>
      </daml:complementOf>
    </daml:Class>
  </rdfs:subClassOf>
</daml:Class>
```

Since *SpecialNonpayment* is a subclass of *ContractorDoesNotPay*, we would infer that it is handled by *ProvideSafeExchangeProtocols*. However, since it is a subclass of the complement of all things that could be handled by *ProvideSafeExchangeProtocols*, we would infer that it could not have the handler *ProvideSafeExchangeProtocols*. These two statements contradict each other, implying that the *SpecialNonpayment* class must be the empty set. If we defined an instance of

SpecialNonpayment, a logical inconsistency would result, since it is impossible for such an instance to exist.

Pragmatically, this is not a critical problem, since there are as yet only a few places in the current content of the MIT Process Handbook where *overrides* is used. Coping with this inability of DAML+OIL to express overrides is an area of future work. See Section 9.1.

4.2 Querying for Process Knowledge

The process repository interface allows agents to query and retrieve process knowledge remotely. Although each agent could simply download the entire DAML+OIL ontology file and query it locally, this would prove too costly in terms of bandwidth and time, since real-world ontologies may be very large. For example, the MIT Process Handbook currently has more than 10,000 entities. Moreover, since ontologies are likely to be updated often, each agent would have to periodically download new versions of the ontology. Therefore, it makes sense for the process repository to provide a remote query interface.

Note that there are other early-stage efforts [41] for querying DAML+OIL ontologies. Future work could relate our remote query interface to these efforts.

An agent can perform the following actions at a process repository:

1. Query for the subclass relationships of classes in the process ontology. This query takes a `closed` argument to specify whether to generate the transitive closure of the subclass relationship – i.e. whether subclasses of a subclass should be returned. For example, a query for the subclasses of *ContractorViolation* without closure would return *ContractorCancelsTask*, *ContractorDoesNotPay*, etc. The results with closure would also include *FradulentCreditPayment*, as it is a subclass of *ContractorDoesNotPay*.

2. Confirm a given subclass or superclass relationship, with or without closure. For example, a query asking whether *FradulentCreditPayment* is a subclass of *ContractorCancelsTask* would return false.
3. Query for the values of the *hasException* property of a process, with or without subclasses (specified using the `withSubclasses` argument). For example, a query for the exceptions of *BuyUsingBilateralNegotiation* would return *ContractorDoesNotPay*, *SubcontractorChangesCost*, *SubcontractorDropsTask*, *SubcontractorIsLate*, etc. Results with subclasses would include *FradulentCreditPayment* as well.
4. Query for the *isHandledBy* properties of an exception, with or without subclasses. For example, a query for the handlers of *FradulentCreditPayment* would return *ProvideSafeExchangeProtocols*. If `withSubclasses` is on, then *ProvideEscrowService* would be returned as well, since it is a subclass of *ProvideSafeExchangeProtocols*.

See Chapter 8 for implementation details of the process repository interface.

5 Contracting Language

In this chapter, we introduce our rule-based contracting language, which allows RuleML provisions to reference process knowledge in a DAML+OIL ontology. We explain in detail how to use this contracting language to specify various types of provisions, including exception handling provisions. The Appendix gives a listing of all the predicates used in SweetDeal.

Although we intend for rules to be exchanged in RuleML format, RuleML is quite verbose (recall Figure 2-6). For ease of human-readability and to save space, this thesis shows all example rules in the SCLPfile text format (see Section 2.3.1), which maps to RuleML in a straightforward manner. The SweetRules software component, which is also part of the SWEET effort, performs this translation automatically.

5.1 Contract Ontology

Before we can define contract rules, we need to extend the process ontology with an ontology for contract concepts at `http://xmlcontracting.org/sd.daml`, where "sd" stands for SweetDeal. (Recall from Section 4.1.2 that both the process ontology and the contract ontology are stored locally in the current prototype. We intend to set up `xmlcontracting.org` in the near future.)

Again we begin with some DAML+OIL header statements. Notice that this ontology imports the process ontology `http://xmlcontracting.org/pr.daml`:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<rdf:RDF
  xmlns:rdf ="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
  xmlns:xsd ="http://www.w3.org/2000/10/XMLSchema#"
  xmlns     ="http://xmlcontracting.org/sd.daml#" >
<daml:Ontology rdf:about="">
<daml:imports rdf:resource="http://www.daml.org/2001/03/daml+oil"/>
 <daml:imports rdf:resource="http://xmlcontracting.org/pr.daml"/>
</daml:Ontology>
```

As described in Section 3.2, we view a contract as a specification for one or more processes. We define the *Contract* class and a *specFor* relation that links each contract to its process(es):

```
<daml:Class rdf:ID="Contract">
  <rdfs:subClassOf>
    <daml:Restriction daml:minCardinality="1">
      <daml:onProperty rdf:resource="#specFor"/>
    </daml:Restriction>
  </rdfs:subClassOf>
</daml:Class>

<daml:ObjectProperty rdf:ID="specFor">
  <rdfs:domain rdf:resource="#Contract" />
  <rdfs:range rdf:resource="http://xmlcontracting.org/pr.daml#Process" />
</daml:ObjectProperty>
```

For the special case of contracts that specify a single process, we define *ContractForOneProcess*, using a `daml:cardinality` restriction to limit the *specFor* relation to exactly one process:

```
<daml:Class rdf:ID="ContractForOneProcess">
  <rdfs:subClassOf rdf:resouce="#Contract"/>
  <rdfs:subClassOf>
    <daml:Restriction daml:cardinality="1">
      <daml:onProperty rdf:resource="#specFor"/>
    </daml:Restriction>
  </rdfs:subClassOf>
</daml:Class>
```

A contract represents the conditions that the parties agreed upon during negotiation. We define another concept, *ContractResult*, to describe the state of how the contract is actually carried out. For example, *ContractResult* could describe the actual shipping date, the quality of the received goods, the amount of payment received, etc.

```
<daml:Class rdf:ID="ContractResult"/>

<daml:ObjectProperty rdf:ID="result">
  <rdfs:domain rdf:resource="#Contract" />
  <rdfs:range rdf:resource="#ContractResult" />
</daml:ObjectProperty>
```

The process ontology provides the *hasException* property to indicate that a process could have a particular exception. To denote that an exception actually happened during contract execution, we define a new *exceptionOccurred* property on *ContractResult*:

```
<daml:ObjectProperty rdf:ID="exceptionOccurred">
  <daml:domain rdf:resource="http://xmlcontracting.org/pr.daml#ContractResult"/>
  <daml:range rdf:resource="http://xmlcontracting.org/pr.daml#Exception"/>
</daml:ObjectProperty>
```

Similarly, we introduce the *exceptionLikely* property. It indicates that, during contract execution, an *anticipate* handler identified the current situation as one in which the specified exception is more likely to occur (but has not actually happened yet). See Section 2.4.2.

```
<daml:ObjectProperty rdf:ID="exceptionLikely">
  <daml:domain rdf:resource="http://xmlcontracting.org/pr.daml#ContractResult"/>
  <daml:range rdf:resource="http://xmlcontracting.org/pr.daml#Exception"/>
</daml:ObjectProperty>
```

Finally, we introduce some relations to specify the purpose of an exception handler. A *DetectException* handler detects certain exception classes, an *AnticipateException* handler anticipates certain exception classes, etc. Notice that the *range* is *Class*, because we want to identify exception classes, not exception instances. Ideally, we would limit the range to subclasses of *Exception*, but DAML+OIL does not currently provide this capability.

```
<daml:ObjectProperty rdf:ID="detectsException">
  <daml:domain rdf:resource="http://xmlcontracting.org/pr.daml#DetectException"/>
  <daml:range rdf:resource="http://www.daml.org/2001/03/daml+oil#Class"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="anticipatesException">
  <daml:domain rdf:resource="http://xmlcontracting.org/pr.daml#AnticipateException"/>
  <daml:range rdf:resource="http://www.daml.org/2001/03/daml+oil#Class"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="avoidsException">
  <daml:domain rdf:resource="http://xmlcontracting.org/pr.daml#AvoidException"/>
  <daml:range rdf:resource="http://www.daml.org/2001/03/daml+oil#Class"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="resolvesException">
  <daml:domain rdf:resource="http://xmlcontracting.org/pr.daml#ResolveException"/>
  <daml:range rdf:resource="http://www.daml.org/2001/03/daml+oil#Class"/>
</daml:ObjectProperty>
```

5.2 Referencing Process Knowledge

Recall from Section 3.2 that contract entities like exception instances and exception handler instances are related to concepts from the DAML+OIL process ontology. Here we describe how to express these relationships in the RuleML contracting language. To our knowledge, this is the first published description and example of such integration of DAML+OIL into RuleML.

Consider an example where a process instance `p123` has a possible exception `e1` that is an instance of the *SubcontractorIsLate* class. We would represent this as two SCLP facts:

```
http://xmlcontracting.org/pr.daml#hasException(p123, e1);
http://xmlcontracting.org/pr.daml#SubcontractorIsLate(e1);
```

The first fact declares that `p123` has an exception `e1`, according to the notion of *hasException* defined in the MIT Process Handbook ontology. In the second fact, we identify `e1` as an instance of the DAML+OIL class *SubcontractorIsLate*. Notably, the system recognizes and interprets `http://xmlcontracting.org/pr.daml#hasException` and `http://xmlcontracting.org/pr.daml#SubcontractorIsLate` as URIs, not simple tokens, so it knows that it can find a definition for these terms at `http://xmlcontracting.org/pr.daml`. This is apparent in the RuleML representation, which specifies each URI as the corresponding element's `href` attribute rather than its content:

```
<fact>
  <_head>
    <atom>
      <_opr><rel href="http://xmlcontracting.org/pr.daml#hasException"/></_opr>
      <ind>p123</ind>
      <ind>e1</ind>
    </atom>
  </_head>
</fact>
<fact>
  <_head>
    <atom>
      <_opr><rel href="http://xmlcontracting.org/pr.daml#SubcontractorIsLate"/> </_opr>
      <ind>e1</ind>
    </atom>
  </_head>
</fact>
```

5.3 Defining Contract Provisions

This section uses some examples to sketch the basics of defining contract provisions that make use of both the process ontology and the contract ontology. Sections 5.4 and 5.5 describe the structure and syntax of contracts in greater detail.

5.3.1 Exception handler instance

Exception handler instances are implementations of handler classes defined in the process ontology. We first consider `detectLateDelivery`, a *detect* handler instance that identifies late deliveries. Figure 5-1 shows a listing of its rules.

```
http://xmlcontracting.org/pr.daml#DetectPrerequisiteViolation(detectLateDelivery);
http://xmlcontracting.org/sd.daml#detectsException(detectLateDelivery,
  http://xmlcontracting.org/pr.daml#SubcontractorIsLate);

<detectLateDelivery_def> http://xmlcontracting.org/sd.daml#exceptionOccurred(?R, ?EI) <-
  http://xmlcontracting.org/pr.daml#specFor(?CO,?PI) AND
  http://xmlcontracting.org/pr.daml#hasException(?PI,?EI) AND
  http://xmlcontracting.org/pr.daml#isHandledBy(?EI,detectLateDelivery) AND
  http://xmlcontracting.org/sd.daml#result(?CO,?R) AND
  shippingDate(?CO,?COD) AND shippingDate(?R,?RD) AND
  greaterThan(?RD,?COD) ;
```

Figure 5-1: detectLateDelivery

Here we give a rule-by-rule description of this handler instance. Since `detectLateDelivery` detects violations of the shipping date requirement specified in the contract, it is an instance of the *DetectPrerequisiteViolation* exception handler:

```
http://xmlcontracting.org/pr.daml#DetectPrerequisiteViolation(detectLateDelivery);
```

Late delivery is a situation where the subcontractor (i.e. seller) is late. As such, we add a rule declaring that this handler instance detects the *SubcontractorIsLate* exception class.

```
http://xmlcontracting.org/sd.daml#detectsException(detectLateDelivery,
  http://xmlcontracting.org/pr.daml#SubcontractorIsLate);
```

This illustrates one of the practical complications in using a process ontology. We would like to say that this handler instance detects the "late delivery" exception, but there is no such exception in the process ontology. As such, we use the more general *SubcontractorIsLate* class instead.

Finally, we define late delivery as a situation where the actual shipping date `?RD` (i.e. as defined in the *ContractResult*) is later than the shipping date `?COD` specified in the contract. When this condition is met, then the exception instance `?EI` that is handled by `detectLateDelivery` has occurred. This logic is captured in the following rule:

```
<detectLateDelivery_def> http://xmlcontracting.org/sd.daml#exceptionOccurred(?R, ?EI) <-
  http://xmlcontracting.org/pr.daml#specFor(?CO,?PI) AND
  http://xmlcontracting.org/pr.daml#hasException(?PI,?EI) AND
  http://xmlcontracing.com/pr.daml#SubcontractorIsLate(?EI) AND
  http://xmlcontracting.org/pr.daml#isHandledBy(?EI,detectLateDelivery) AND
  http://xmlcontracting.org/sd.daml#result(?CO,?R) AND
  shippingDate(?CO,?COD) AND shippingDate(?R,?RD) AND
  greaterThan(?RD,?COD) ;
```

Notice that this rule is *generalized*, as it does not refer to any particular contract or process instance. As expressed by the highlighted conditions, the rule applies to any contract whose process has an exception that is handled by `detectLateDelivery`. To use `detectLateDelivery` in another contract, we can simply copy the rules into that contract and use them verbatim. In contrast, an ungeneralized version of the rule (see Figure 5-2) would have to be modified for use in another contract – i.e. by replacing `p123` with the new process instance and `co123` with the new contract. We will see in Chapter 6 that generalized rules are essential for modular provisions called *contract fragments*.

```
<detectLateDelivery_def> http://xmlcontracting.org/sd.daml#exceptionOccurred(?R, ?EI) <-
  http://xmlcontracting.org/pr.daml#hasException(p123,?EI) AND
  http://xmlcontracting.org/pr.daml#isHandledBy(?EI,detectLateDelivery) AND
  http://xmlcontracting.org/sd.daml#result(co123,?R) AND
  shippingDate(co123,?COD) AND shippingDate(?R,?RD) AND
  greaterThan(?RD,?COD) ;
```

Figure 5-2: Ungeneralized version of detectLateDelivery_def

Next we consider a more complex *avoid* handler instance, `lateDeliveryPenalty`, that charges the seller a penalty fee for late delivery.

```
http://xmlcontracting.org/pr.daml#PenalizeForContingency(lateDeliveryPenalty);
http://xmlcontracting.org/sd.daml#avoidsException(lateDeliveryPenalty,
  http://xmlcontracting.org/pr.daml#SubcontractorIsLate);

// penalty = - overdueDays * 50
<lateDeliveryPenalty_def> payment(?R, contingentPenalty, ?Penalty) <-
  http://xmlcontracting.org/pr.daml#specFor(?CO,?PI) AND
  http://xmlcontracting.org/pr.daml#hasException(?PI,?EI) AND
  http://xmlcontracting.org/pr.daml#isHandledBy(?EI,lateDeliveryPenalty) AND
  http://xmlcontracting.org/sd.daml#result(?CO,?R) AND
  http://xmlcontracting.org/sd.daml#exceptionOccurred(?R,?EI) AND
  shippingDate(co123,?CODate) AND shippingDate(?R,?RDate) AND
  subtract(?RDate,?CODate,?OverdueDays) AND
  multiply(?OverdueDays, 50, ?Res1) AND multiply(?Res1, -1, ?Penalty) ;
```

Figure 5-3: lateDeliveryPenalty

Since it imposes a penalty in the event that a late delivery occurs, `lateDeliveryPenalty` is an instance of *PenalizeForContingency.* It helps avoid the *SubcontractorIsLate* exception by giving an incentive to the seller to ensure that it does not deliver late.

```
http://xmlcontracting.org/pr.daml#PenalizeForContingency(lateDeliveryPenalty);
http://xmlcontracting.org/sd.daml#avoidsException(lateDeliveryPenalty,
  http://xmlcontracting.org/pr.daml#SubcontractorIsLate);
```

Once an exception instance that is handled by `lateDeliveryPenalty` has occurred, we calculate the appropriate penalty. The penalty is specified as a payment of type `contingentPenalty`. It is negative, since we define all payments to be from the buyer to the seller. The penalty formula we use in this example is to charge $50 per day late. We calculate this penalty by multiplying the number of overdue days, which is the difference between the contract shipping date and the actual shipping date, by -50. More complex formulas can be incorporated in a similar manner.

```
// penalty = - overdueDays * 50
<lateDeliveryPenalty_def> payment(?R, contingentPenalty, ?Penalty) <-
  http://xmlcontracting.org/pr.daml#specFor(?CO,?PI) AND
  http://xmlcontracting.org/pr.daml#hasException(?PI,?EI) AND
  http://xmlcontracting.org/pr.daml#isHandledBy(?EI,lateDeliveryPenalty) AND
  http://xmlcontracting.org/sd.daml#result(?CO,?R) AND
  http://xmlcontracting.org/sd.daml#exceptionOccurred(?R,?EI) AND
  shippingDate(co123,?CODate) AND shippingDate(?R,?RDate) AND
  subtract(?RDate,?CODate,?OverdueDays) AND
  multiply(?OverdueDays, 50, ?Res1) AND multiply(?Res1, -1, ?Penalty) ;
```

5.3.2 Contract

Now we show how to define a contract `co123` that uses some exception handlers, as shown in Figure 5-4.

```
http://xmlcontracting.org/sd.daml#Contract(co123);
http://xmlcontracting.org/sd.daml#specFor(co123,p123);
http://xmlcontracting.org/sd.daml#BuyWithBilateralNegotiation(p123);

http://xmlcontracting.org/pr.daml#hasException(p123,e1);
http://xmlcontracting.org/pr.daml#SubcontractorIsLate(e1);
<e1_detect> http://xmlcontracting.org/pr.daml#isHandledBy(e1,detectLateDelivery);
<e1_avoid> http://xmlcontracting.org/pr.daml#isHandledBy(e1,lateDeliveryRiskPayment);
```

Figure 5-4: Provisions in contract co123

`co123` is an instance of the *Contract* class from the contract ontology. It is the specification for `p123`, an instance of the *BuyWithBilateralNegotiation* process from the process ontology:

```
http://xmlcontracting.org/sd.daml#Contract(co123);
http://xmlcontracting.org/sd.daml#specFor(co123,p123);
http://xmlcontracting.org/sd.daml#BuyWithBilateralNegotiation(p123);
```

The process ontology lists *SubcontractorIsLate* as one of the exceptions for *BuyWithBilateralNegotation*. This suggests that our process instance has a potential exception instance `e1` that is an instance of *SubcontractorIsLate*. We declare this with two rules:

```
http://xmlcontracting.org/pr.daml#hasException(p123,e1);
http://xmlcontracting.org/pr.daml#SubcontractorIsLate(e1);
```

Finally, we set up a mechanism to manage this possible exception by associating `e1` with `detectLateDelivery` as a *detect* handler and `lateDeliveryRiskPayment` as an *avoid* handler:

```
<e1_detect> http://xmlcontracting.org/pr.daml#isHandledBy(e1,detectLateDelivery);
<e1_avoid> http://xmlcontracting.org/pr.daml#isHandledBy(e1,lateDeliveryRiskPayment);
```

5.3.3 Modifying Provisions

The prioritized overrides mechanism of SCLP provides a convenient way to modify contract provisions by simply adding new rules, without removing or modifying any existing rules. This is particularly useful for creating counterproposals during negotiation.

For example, how do we augment the previous provisions so that `e1` is avoided by `lateDeliveryPenalty` instead of `lateDeliveryRiskPayment`? We can add the following facts:

```
<e1_avoid2> http://xmlcontracting.org/pr.daml#isHandledBy(e1,lateDeliveryPenalty);
MUTEX
  http://xmlcontracting.org/pr.daml#isHandledBy(e1,lateDeliveryRiskPayment) AND
  http://xmlcontracting.org/pr.daml#isHandledBy(e1,lateDeliveryPenalty);
overrides(e1_avoid2,e1_avoid);
```

The MUTEX statement says that `e1` cannot be handled by *both* `lateDeliveryRiskPayment` and `lateDeliveryPenalty`. We use an *overrides* fact to choose between the two. Since we declare the new handler declaration `e1_avoid2` to take precedence over the existing declaration `el_avoid`, only `el_avoid2` takes effect, so that `el` is now handled by `detectLateDelivery` and `lateDeliveryPenalty`, but not `lateDeliveryRiskPayment`.

5.4 Contract Structure

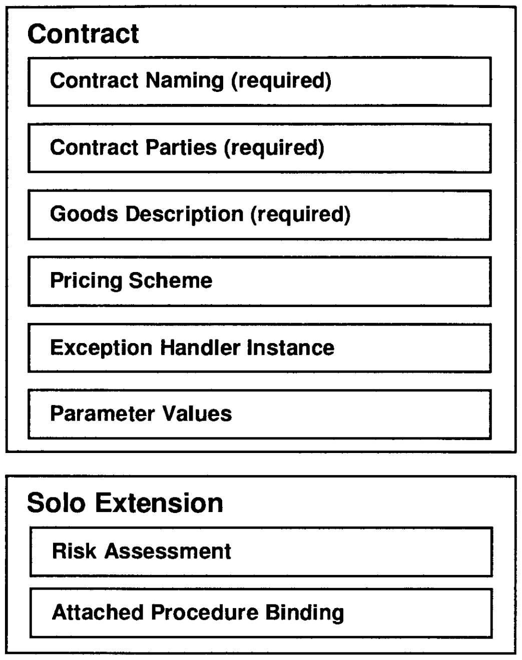

Contract
Contract Naming (required)
Contract Parties (required)
Goods Description (required)
Pricing Scheme
Exception Handler Instance
Parameter Values
Solo Extension
Risk Assessment
Attached Procedure Binding

Figure 5-5: Typical contract structure

The rules in a contract are conceptually grouped into a number of *contract sections*, which describe different provisions (Figure 5-5). The *contract* consists of a set of contract sections that is exchanged with the other parties in the negotiation. In addition, each market agent may associate with the contract a number of provisions that it does not share with the other parties (i.e. provisions that are private to the agent itself), called the *solo extension*. An agent could use the solo extension for proprietary knowledge that it does not want to disclose, such as its own valuation or business rules. In addition, the solo extension could be used for rules that do not apply to other parties, such as attached procedure bindings that refer to Java classes and methods which may not be available to all the parties in the negotiation (see Section 5.5.8).

5.5 Types of Contract Sections

In this section, we describe the different types of contract sections in detail. Only *contract naming*, *contract parties*, and *goods description* are required for every contract; the other types of contract sections are optional. We illustrate each type using examples from a scenario where Acme, a small firm in Mexico, is looking to purchase plastic product #425 from Plastics Etc., a large firm in the United States. All the example rules outside of *contract naming*, *contract parties*, and *goods description* are generalized – i.e. they do not refer to specific contracts or process instances. (Recall from Section 5.3.1 that only generalized rules can be part of a contract fragment.)

5.5.1 Contract Naming

(required)

This section names the contract, the process associated with the contract, and the contract result. In this example, contract `co123` is a specification for process `co123_process`, an instance of *BuyWithBilateralNegotiation*. `co123_res` is the *ContractResult* of `co123`.

```
http://xmlcontracting.org/sd.daml#Contrac(co123);
http://xmlcontracting.org/sd.daml#specFor(co123,co123_process);
http://xmlcontracting.org/sd.daml#BuyWithBilateralNegotiation(co123_process);
http://xmlcontracting.org/sd.daml#result(co123,co123_res);
```

5.5.2 Contract Parties

(required)

Here we name the parties in the contract and use rules to describe their attributes. In this thesis, we consider a simple scenario with one buyer and one seller.

```
buyer(co123,acme);
seller(co123,plastics_etc);
country(acme,Mexico);
country(plastics_etc,USA);
firmSize(acme,small);
firmSize(plastics_etc,large);
```

5.5.3 Goods Description

(required)

Similarly, *goods description* names the products or services for this contract and describes them using rules. This thesis considers a simple scenario where each contract is for a single product.

```
product(co123,plastic425);
```

5.5.4 Pricing Scheme

The pricing scheme is a set of rules that are used to determine the price. The price could be specified as a function of attributes like quantity, quality, or shipping date.

The example pricing scheme `stdVolPricing` defines two possible prices as a function of quantity and shipping date, where the shipping date is defined relative to the order date (e.g. "5" means "5 days after the order date"). The standard price is $50, but if an order is large enough and the shipping date required is not too urgent, then it is eligible for the lower volume price of $45. Notice that we use priority overrides to naturally express the conditions for the two prices.

```
/**
<standard> p = 50 if 100<=Q<=1000 and 10<=D<=20
<volume>   p = 45 if 700<=Q<=1000 and 15<=D<=20
**/
<standard> price(?CO, 50) <-
  quantity(?CO, ?Q) AND
  greaterThanOrEquals(?Q, 100) AND equalsOrLessThan(?Q, 1000) AND
  shippingDate(?CO, ?D) AND
  greaterThanOrEquals(?D, 10) AND equalsOrLessThan(?D, 20) ;
<volume> price(?CO, 45) <-
  quantity(?CO, ?Q) AND
  greaterThanOrEquals(?Q, 700) AND equalsOrLessThan(?Q, 1000) AND
  shippingDate(?CO, ?D) AND
  greaterThanOrEquals(?D, 15) AND equalsOrLessThan(?D, 20) ;
overrides(volume, standard) ;
```

Figure 5-6: stdVolPricing

5.5.5 Risk Assessment

When creating a contract, how do we decide which exceptions are most pertinent? The process ontology gives the exceptions that are associated with a process, but it does not indicate which ones are more likely to occur for a given situation (i.e. process instance). A *risk assessment* is a set of rules that estimate the expected conditional probability of a particular exception given a contract's attributes, such as characteristics of the buyer, seller, or product. Naturally, this probability ranges from 0 to 1, where 1 is the greatest risk. It might, for example, be calculated as a statistical average over contracts with the given attribute values, if such information were available. Each risk assessment includes a `riskFor` statement to indicate the exception that it calculates the risk for.

The following example, `nonpaymentRisk`, specifies the conditional risk for *ContractorDoesNotPay* as an average of two risk components: the size of the buyer firm and whether the purchase is international. The risk of nonpayment is higher if the buyer firm is small or the purchase is international. For example, a contract with a small buyer and an international purchase would be estimated to have (0.08+0.24)/2 = 16% risk of nonpayment, while one with a large firm and a domestic purchase would have a risk of (0.02+0.04)/2 = 3%. This is clearly a very simplistic approach. More sophisticated approaches for estimating risk probabilities could employ Bayesian networks [40].

```
riskFor(nonpaymentRisk, http://xmlcontracting.org/pr.daml#ContractorDoesNotPay) ;

// *** Conditional risk components:
// whether the buyer is a small company
<rc> riskComponent(?C,http://xmlcontracting.org/pr.daml#ContractorDoesNotPay,
  businessSize,0.08) <-
  buyer(?C,?Buyer) AND firmSize(?Buyer,small);
<rc_default> riskComponent(?C,http://xmlcontracting.org/pr.daml#ContractorDoesNotPay,
  businessSize,0.02) <- http://xmlcontracting.org/sd.daml#Contract(?C);

// whether the purchase is international
<rc> riskComponent(?C,http://xmlcontracting.org/pr.daml#ContractorDoesNotPay,
  internationality,0.24) <-
  buyer(?C,?Buyer) AND seller(?C,?Seller) AND
  country(?Buyer,?BCountry) AND country(?Seller,?SCountry) AND
  notEquals(?BCountry,?SCountry);
<rc_default> riskComponent(?C,http://xmlcontracting.org/pr.daml#ContractorDoesNotPay,
  internationality,0.04) <- http://xmlcontracting.org/sd.daml#Contract(?C);

overrides(rc,rc_default);

// *** Conditional risk function
// Calculate the risk score as the average of the risk components
<risk_cond> risk(?C,http://xmlcontracting.org/pr.daml#ContractorDoesNotPay,?Risk) <-
  riskComponent(?C,http://xmlcontracting.org/pr.daml#ContractorDoesNotPay,
  businessSize,?Size) AND
  riskComponent(?C,http://xmlcontracting.org/pr.daml#ContractorDoesNotPay,
  internationality,?Intl) AND
  add(?Size,?Intl,?Total) AND divide(?Total,2.0,?Risk) ;
```

Figure 5-7: nonpaymentRisk

5.5.6 Exception Handler Instance

Each exception handler instance is an implementation of an exception handler class from the process ontology. As previously described, a contract uses the `isHandledBy` predicate to associate an exception instance with a handler instance. The following fact declares that the exception instance `e1` is handled by `detectLateDelivery`:

```
http://xmlcontracting.org/pr.daml#isHandledBy(e1,detectLateDelivery);
```

The following sections illustrate the four types of handlers instances through examples.

Detect handler

See `detectLateDelivery` in Section 5.3.1.

Anticipate handler

One anticipate handler class in the process ontology is *CheckCreditLine*. Figure 5-8 presents an instance that anticipates *ContractorDoesNotPay* exceptions by checking whether the buyer's credit rating is low. If so, then the handler instance declares that the exception is likely. The credit rating is actually obtained using a *sensor* attached procedure called `creditRating`. Notice that we use the `sensable` predicate to declare to SweetDeal that we intend for `creditRating` to be bound to a sensor. See Section 5.5.8.

```
http://xmlcontracting.org/pr.daml#CheckCreditLine(checkCreditRating);
http://xmlcontracting.org/sd.daml#anticipatesException(
  checkCreditRating, http://xmlcontracting.org/pr.daml#ContractorDoesNotPay);

http://xmlcontracting.org/sd.daml#exceptionLikely(?R, ?EI) <-
  http://xmlcontracting.org/pr.daml#specFor(?CO,?PI) AND
  http://xmlcontracting.org/pr.daml#hasException(?PI,?EI) AND
  http://xmlcontracting.org/pr.daml#isHandledBy(?EI,checkCreditRating) AND
  http://xmlcontracting.org/sd.daml#result(?CO,?R) AND
  buyer(?CO,?Buyer) AND creditRating(?Buyer,low);

sensable(creditRating);
```

Figure 5-8: checkCreditRating

Avoid handler

Here we present an augmented version of `lateDeliveryPenalty` from Section 5.3.1. Recall that the original `lateDeliveryPenalty` imposed a penalty of $50 per day late. What if we want to use this handler instance in multiple contracts with different penalty amounts? The modified rules in Figure 5-9 introduce a `lateDeliveryPerUnitPerDayPenalty` parameter that allows each contract to configure its own penalty amount. See Section 5.5.7 for a discussion of parameters and parameter values.

```
http://xmlcontracting.org/pr.daml#PenalizeForContingency(lateDeliveryPenalty);
http://xmlcontracting.org/sd.daml#avoidsException(lateDeliveryPenalty,
  http://xmlcontracting.org/pr.daml#SubcontractorIsLate);

// penalty = - overdueDays * perDayPenalty
<lateDeliveryPenalty_def> payment(?R, contingentPenalty, ?Penalty) <-
  http://xmlcontracting.org/pr.daml#specFor(?CO,?PI) AND
  http://xmlcontracting.org/pr.daml#hasException(?PI,?EI) AND
  http://xmlcontracting.org/pr.daml#isHandledBy(?EI,lateDeliveryPenalty) AND
  http://xmlcontracting.org/sd.daml#result(?CO,?R) AND
  http://xmlcontracting.org/sd.daml#exceptionOccurred(?R,?EI) AND
  shippingDate(co123,?CODate) AND shippingDate(?R,?RDate) AND
  subtract(?RDate,?CODate,?OverdueDays) AND
  parameterValue(?CO,lateDeliveryPerDayPenalty,?PerDayPenalty) AND
  multiply(?OverdueDays, ?PerDayPenalty, ?Res1) AND multiply(?Res1, -1, ?Penalty) ;

parameter(?CO, lateDeliveryPerDayPenalty) <-
  http://xmlcontracting.org/sd.daml#Contract(?CO);
```

Figure 5-9: lateDeliveryPenalty with parameter

Resolve handler

A simple way to resolve an exception is to notify someone in the firm who is responsible for the process. `emailNotify`, an instance of the *NotifyAboutExceptionUsingEmail* handler class, sends an email notification when an exception occurs. Since `emailNotify` can notify about exceptions of any class, it declares that it resolves the *Exception* class. The email sending itself is accomplished using an *effector* attached procedure called `notifyByEmail`. We use the `effectable` predicate to declare that we intend for it to be bound to an effector. See Section 5.5.8.

```
http://xmlcontracting.org/pr.daml#NotifyAboutExceptionUsingEmail(emailNotify);
http://xmlcontracting.org/sd.daml#resolvesException(emailNotify,
  http://xmlcontracting.org/pr.daml#Exception);

notifyByEmail(?R,?EI) <-
  http://xmlcontracting.org/pr.daml#specFor(?CO,?PI) AND
  http://xmlcontracting.org/pr.daml#hasException(?PI,?EI) AND
  http://xmlcontracting.org/pr.daml#isHandledBy(?EI,emailNotify) AND
  http://xmlcontracting.org/sd.daml#result(?CO, ?R) AND
  http://xmlcontracting.org/sd.daml#exceptionOccurred(?R,?EI);

effectable(notifyByEmail);
```

Figure 5-10: emailNotify

5.5.7 Parameter Value

Any contract section can use the `parameter` predicate to declare a parameter. For example, the following rule from Figure 5-9 declares `lateDeliveryPerDayPenalty` as a parameter:

```
parameter(?CO, lateDeliveryPerDayPenalty) <-
  http://xmlcontracting.org/sd.daml#Contract(?CO);
```

The *parameter value* contract section consists of `parameterValue` facts that set the values for various parameters. For example, this fact sets the penalty for late delivery to be $32 per day:

```
parameterValue(co123,lateDeliveryPerDayPenalty,32);
```

Note that a contract proposal does not have to specify values for all the parameters. Parameters left unconfigured may be decided during negotiation. (Indeed, unconfigured parameters influence the nature of the negotiation.) All parameters must be configured in a completed contract.

5.5.8 Attached Procedure Binding

Recall from Section 2.2 that attached procedures allow SCLP rules to access other software systems. *Sensors* test for certain conditions, while *effectors* perform actions. The SCLP mechanism allows rules to use an attached procedure independently from that procedure's binding to a Java method. For example, `checkCreditRating.clp` (Figure 5-8) uses the `creditRating` sensor without specifying how it is implemented. We use the `sensable` and `effectable` predicates as controlled hints of which predicates are intended to be bound to attached procedures. These hints may be used by an editor (see Section 7.3) to prompt the user to select the appropriate bindings. They have no semantic impact, however.

The actual binding from a predicate to a Java method is defined in an *attached procedure binding*, which consists of a number of SCLP binding statements. The following example binding associates the `creditRating` predicate with the `getCreditRating` method of the Java class `sweet.deal.aprocs.OrderManager`.

```
  Sensor: creditRating
  Class: OrderManager
  Method: getCreditRating
  BindingRequirement: (BOUND,FREE)
  path: "sweet.deal.aprocs" ;
```

Figure 5-11: creditRatingSensor

6 Contract Fragments

One of the advantages of using a declarative, rule-based representation for contracts is that we can easily compose a contract from reusable pieces. In this chapter, we introduce *contract fragments* – self-contained and reusable modules of contract provisions. Contract fragments are contract sections that have been separated from a contract, cast in general terms, and named by a URI, allowing them to be used by multiple contracts. As suggested by Figure 6-1, this allows us to construct a contract by specifying which contract fragments to include as provisions. Instead of copying the rules of the contract fragments into the contract, we simply add `_includes` statements that reference their URIs. This is called *incorporation by reference*.

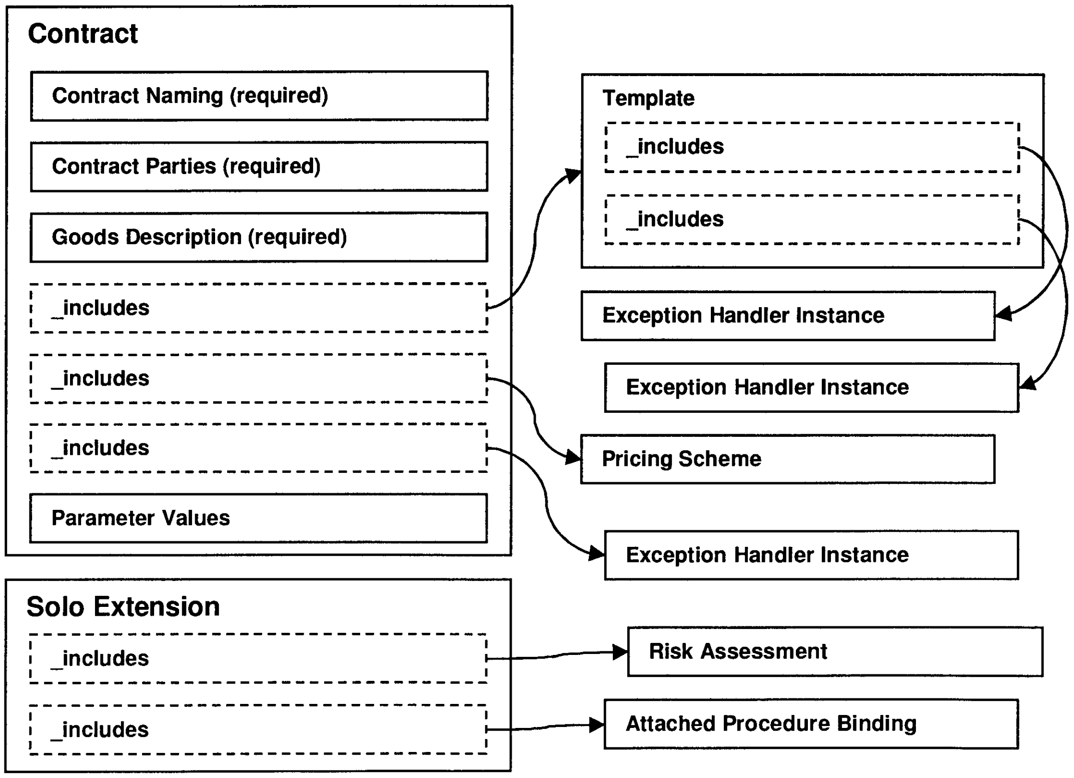

Contract
Contract Naming (required)
Contract Parties (required)
Goods Description (required)
_includes
_includes
_includes
Parameter Values
Solo Extension
_includes
_includes
Template
_includes
_includes
Exception Handler Instance
Exception Handler Instance
Pricing Scheme
Exception Handler Instance
Risk Assessment
Attached Procedure Binding

Figure 6-1: Typical structure of a contract with included contract fragments.

In this chapter, we first explain the mechanism for processing contracts with included contract fragments. Next, we describe the syntax of contract fragments. After that, we introduce contract repositories, which provide a storehouse for contract fragments and an interface to query for them based on specific attributes. Finally, we discuss some benefits and challenges of using contract fragments.

6.1 Module-Inclusion Mechanism (_includes)

A contract uses the special `_includes` predicate to incorporate a contract fragment as part of its provisions. For example, if a contract wants to include a contract fragment named by the URI `http://www.xmlcontracting.org/cf/detectLateDelivery.clp`, it would have the following fact:

```
_includes(http://www.xmlcontracting.org/cf/detectLateDelivery.clp);
```

How does SweetDeal evaluate and execute a contract that includes contract fragments? For every `_includes` statement that it finds in the contract provisions, the system downloads the rules from the associated URI and adds them to the *active ruleset* of rules to be evaluated for this contract. This mechanism is recursive – i.e. if an included contract fragment contains `_includes` statements, then those contract fragments will be downloaded and added to the active ruleset as well. In addition, it keeps track of which contract fragments it has included and will not include the same contract fragment twice, thus avoiding cycles.

With incorporation by reference, the included contract fragments are, by default, not sent along with the contract proposal. However, there are certain scenarios where sending along a *courtesy copy* of some of the included contract fragments may be helpful. A courtesy copy could save the other parties the trouble of downloading an infrequently-used contract fragment. In addition, it could provide a fallback option in case the contract repository becomes inaccessible, for example if the server or the network goes down.

Extending RuleML to support incorporation by reference is an area of future work. A possible approach is to use XML Inclusions (XInclude) [26], which was published by the World Wide Web Consortium as a candidate recommendation in February 2002. XInclude defines a general purpose mechanism for an XML document to include (i.e. be merged with) other XML documents. Notably, XInclude has a fallback mechanism that is similar to the notion of a courtesy copy, although its processing model only uses the fallback content if the included document is not available.

6.2 Syntax, Types, Creation

The syntax of contract fragments is largely similar to that of contract sections, but there are a few important differences. We illustrate this by converting the `detectLateDelivery` contract section from Figure 5-1 into the contract fragment in Figure 6-2.

```
contractFragment(http://xmlcontracting.org/cf/detectLateDelivery.clp, handler);
http://xmlcontracting.org/pr.daml#DetectPrerequisiteViolation(
  http://xmlcontracting.org/cf/detectLateDelivery.clp);
http://xmlcontracting.org/sd.daml#detectsException(
  http://xmlcontracting.org/cf/detectLateDelivery.clp,
  http://xmlcontracting.org/pr.daml#SubcontractorIsLate);

<detectLateDelivery_def> http://xmlcontracting.org/sd.daml#exceptionOccurred(?R, ?EI) <-
  http://xmlcontracting.org/pr.daml#specFor(?CO,?PI) AND
  http://xmlcontracting.org/pr.daml#hasException(?PI,?EI) AND
  http://xmlcontracting.org/pr.daml#isHandledBy(?EI,
    http://xmlcontracting.org/cf/detectLateDelivery.clp) AND
  http://xmlcontracting.org/sd.daml#result(co123,?R) AND
  shippingDate(?CO,?COD) AND shippingDate(?R,?RD) AND
  greaterThan(?RD,?COD) ;
```

Figure 6-2: http://xmlcontracting.org/cf/detectLateDelivery.clp

First, we ensure that all the rules are generalized and make no reference to any specific contract or process instance. This is already the case for `detectLateDelivery`. Second, we assign a URI name to the contract fragment, so that any contract can refer to it unambiguously. We use `http://xmlcontracting.org/cf/detectLateDelivery.clp` for this example. As we will see in Section 6.3, `xmlcontracting.org` is a contract repository that hosts many contract fragments. Finally, we add a `contractFragment` fact to indicate the type of this contract fragment (see Table 6-1). Our example is a contract fragment of type `handler`:

```
contractFragment(http://xmlcontracting.org/cf/detectLateDelivery.clp, handler);
```

contract fragment type	type name
exception handler instance	`handler`
risk assessment	`riskAssessment`
pricing scheme	`pricing`
attached procedure binding	`aprocBinding`
contract template	`template`

Table 6-1: Contract fragment types

6.2.1 Contract Template

Contract fragments introduce a new type of provision called *contract template*, which is simply a partial contract with some number of arbitrary provisions. For example, a template could declare some exception instances, specify their risk assessments, and specify their handler instances. Templates provide a "shortcut" to contract creation by providing some commonly-used provisions, allowing the user to focus on adding custom provisions that are specific to the situation. Similar to a contract, a template uses the `templateFor` predicate to indicate the process class that it specifies.

Figure 6-3 shows an example template for the *BuyUsingBilateralNegotiation* process. First, it declares a *SubcontractorIsLate* exception `e1`. Next, it incorporates two exception handler contract fragments, `http://xmlcontracting.org/cf/detectLateDelivery.clp` and `http://xmlcontracting.org/cf/lateDeliveryPenalty.clp`, and specifies them to be handlers for `e1`.

```
contractFragment(http://xmlcontracting.org/cf/bilateralBuy.clp, template);
templateFor(http://xmlcontracting.org/cf/bilateralBuy.clp,
  http://xmlcontracting.org/pr.daml#BuyUsingBilateralNegotiation) ;

http://xmlcontracting.org/pr.daml#hasException(?PI, e1) <-
  http://xmlcontracting.org/sd.daml#specFor(?CO,?PI);
http://xmlcontracting.org/pr.daml#SubcontractorIsLate(e1);

http://xmlcontracting.org/pr.daml#isHandledBy(e1,
  http://xmlcontracting.org/cf/detectLateDelivery.clp);
_include(http://xmlcontracting.org/cf/detectLateDelivery.clp);

http://xmlcontracting.org/pr.daml#isHandledBy(e1,
  http://xmlcontracting.org/cf/lateDeliveryPenalty.clp);
_include(http://xmlcontracting.org/cf/lateDeliveryPenalty.clp);
```

Figure 6-3: http://xmlcontracting.org/cf/bilateralBuy.clp

6.3 Contract Repository

Contract fragments are stored in *contract repositories*. A contract repository functions as a server for downloading contract fragments as well as a query interface for finding them. In general, a contract could include contract fragments from multiple contract repositories.

As a server, a contract repository provides a URI name for every contract fragment. For example, the contract fragments of a contract repository at `xmlcontracting.org` could have URIs of the form `http://xmlcontracting.org/cf/`*fragmentName*. This aspect of a contract repository could be provided by a standard HTTP server.

How do market agents find contract fragments at a contract repository? This functionality is provided by the query interface. Here we describe a simple indexing mechanism that allows agents to query for contract fragments based on pre-defined sets of attributes. An area of future research is to develop better index and query mechanisms.

For each type of contract fragment, we select an attribute that is used to index and query contract fragments of that type. Thus we can maintain a hash table where contract fragments are keyed by that attribute. Table 6-2 shows the index attribute for each contract fragment type.

contract fragment type	index attribute
exception handler instance	exception handler class
risk assessment	exception class
pricing scheme	*none*
attached procedure binding	attached procedure predicate
contract template	process class

Table 6-2: Index attribute for each contract fragment type.

In addition, we allow queries by (exception handler, exception) pairs for exception handler instances, since agents may only be interested in handler instances that deal with a certain exception. For these queries, we return handler instances that deal with the specified exception or any of its superclasses. If a handler instance handles a certain exception class, it should be able to handle all of its subclasses as well.

6.4 Discussion

Why should we use contract fragments instead of contract sections? What problems are introduced by contract fragments? CommonAccord's [27] non-automated *c-Terms* (see Section 2.2) present similar benefits and challenges as contract fragments in SweetDeal.

Contract fragments provide several benefits:

- **Modularity**: Contract fragments are self-contained and describe the terms that they implement. As a result, a complete contract may be constructed by simply including contract fragments that realize the desired provisions.
- **Reuse**: Contract fragments can substantially reduce the effort of creating a contract. Instead of trying to write the terms anew every time, we can simply include a contract fragment that has been known to work in the past.
- **Best practice**: There may be different ways to implement any given provision. Contract repositories provide an easy way to organize and retrieve contract fragments that implement similar terms. In addition, one could use a reputation repository that maintains a reputation score for each contract fragment. After every use of a contract fragment in the marketplace, its score could be updated by soliciting ratings from the agents involved. Using these scores, the system could suggest which contract fragments best implement a particular provision.
- **Human cognitive cost**: In addition to easing contract creation, the use of contract fragments may make it easier for people to understand a contract. A person could avoid reading the details of a particular provision if that provision is implemented by a contract

fragment that he is familiar with. This allows him to focus on the unique parts of the contract – i.e. those that differ from “standard practice”.

- **Legal implications**: As suggested by the CommonAccord project, incorporation by reference is a legally accepted practice. Therefore, a contract with fragments included by reference has equivalent legal power as one where those provisions are copied into the body of the contract.

Contract fragments also raise several challenges which could be basis for future work:

- **Content integrity**: How do we ensure that the contents of a contract fragment referenced in a contract will remain the same when other parties access it? This is related to the more general issue of URL persistence [29]. One solution is for contract repositories to ensure as an organizational commitment that the contract fragment at a particular URI will never change. Newer versions of that contract fragment will be assigned different URIs.
- **Access control**: Complex contract fragments may be valuable intellectual property. A company that develops an effective contract fragment may seek to limit the parties that can access it. There is a tension between this proprietary desire and the universal access encouraged by URIs. Contract repositories could restrict access based on the requesting client.

7 Market Agent

Market agents are software agents that represent the participants in the electronic marketplace. Although this thesis focuses on a simple scenario with one buyer and one seller, marketplaces could in general have multiple buyers and sellers, as well as intermediaries like auctioneers and insurance agencies. In this chapter, we first give an overview of how the components of the market agent can automate contract creation, negotiation, and execution. Next, we explain the behavior of each component in detail. Finally, we present the Proposal Creation Wizard, which combines the market agent components in a graphical interface for creating contract proposals.

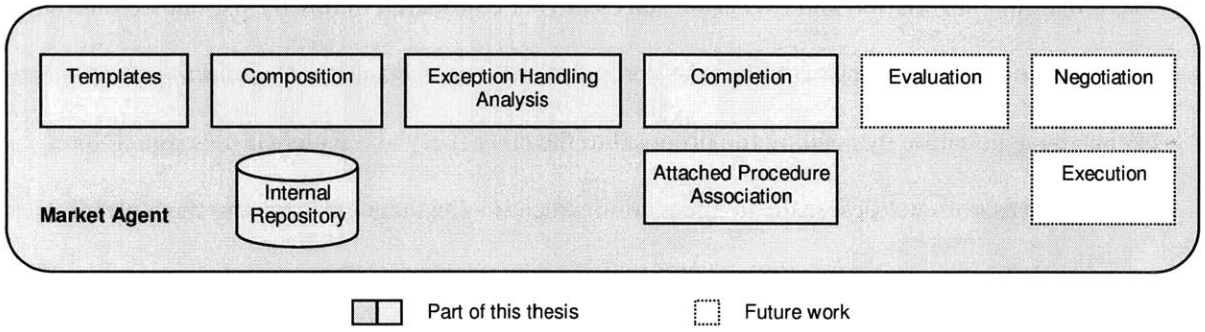


Figure 7-1: Components of a market agent.

7.1 Overall Workflow

As shown in Figure 7-1, a market agent uses several mechanisms to create, evaluate, negotiate, and execute contracts. This thesis specifies the core components for proposal creation. The *Evaluation*, *Negotiation*, and *Execution* components are areas for future work.

To create a contract proposal, a market agent first asks the user for the process class to be specified by the contract, which is added to the proposal as a *contract naming* contract section. Next, the agent lets the user specify the contract parties, the contract goods, and their attributes, which are added to the proposal as *contract parties* and *goods description* contract sections. In the following step, the *Templates* component lets the user choose a contract template for the selected process from a contract repository. The *Composition* component adds this contract fragment to the proposal. Then the agent prompts the user to select a pricing scheme, which is also added to the proposal using *Composition.* After that, the agent performs *Exception Handling Analysis* to identify the possible exceptions for this proposal and prompt the user to select risk assessments and exception handler instances for each exception. These contract fragments are then added to the proposal. In *Attached Procedure Association*, the agent asks the user to select bindings for the sensors and effectors in the proposal. The final step, *Completion*, allows the user to set the values for parameters in the proposal.

The remaining negotiation and execution stages are not considered in this thesis, but we outline them here to complete the workflow. After proposal creation, the agent's *Negotiation* component begins the negotiation by sending the proposal to the other party as a RuleML message. (Note that this thesis does not designate an agent communication language for carrying the RuleML proposals between agents. See [28] and [38] for work in this area.) After receiving the proposal, the other agent uses the *Evaluation* component to calculate a utility score for the proposal based on its preferences. If the user is not satisfied with these provisions (i.e. he believes that he can negotiate for a contract that achieves higher utility), he creates a counterproposal by changing parameter values or specifying different exception handlers. Finally, the agent uses the *Negotiation* component to send the counterproposal back to the other party.

On the other hand, if the user is satisfied with the proposal it receives from the other party, then he tells the agent to send back an *accept* message. The accepted proposal then becomes a contract, and the execution phase begins. Each party uses its *Execution* module to carry out the provisions of the contract through inference on the rules. When an exception condition occurs, the rules of its associated *detect exception* handler will fire and assert the presence of the exception. Then the rules of any associated *avoid exception* or *resolve exception* handlers will fire, executing the provisions for fixing this exception. Once execution completes, the contract has been fulfilled.

Future work could automate the contracting process to a greater degree. For example, once an *Evaluation* component is developed, it could help the agent choose a pricing scheme automatically, by comparing the expected utility scores from using different pricing schemes. A similar approach could be used to automatically choose advantageous parameter values. In general, the *Evaluation* component could be part of a strategy for automatically generating proposals and counterproposals. Such strategies are a current area of research [9][32][33].

7.2 Component Details

7.2.1 Internal Repository

The *internal repository* is a local contract repository whose contents are only available to the owner agent. An agent uses its internal repository to store proprietary contract fragments that it may not want to disclose and contract fragments that may not be meaningful to other agents, such as attached procedure bindings. Contract fragments from the internal repository are typically placed in the solo extension of a contract.

7.2.2 Templates

This component queries a contract repository for templates that specify the contract's process and allows the user to choose one to add to the proposal.

7.2.3 Composition

Composition is the adding of contract fragments to a contract. Contract fragments from contract repositories are incorporated by reference using the `_includes` predicate, while those from the internal repository are added by copying their rules into the contract.

7.2.4 Exception Handling Analysis

This component determines the possible exceptions for a contract, evaluates their risk, and adds associated exception handling provisions.

Exception Handling Analysis first queries the process repository for exceptions associated with the contract's process. It then adds facts to the proposal to declare corresponding exception instances for the contract. For example, if contract `co123` specifies an instance of the *BuyWithBilateralNegotiation* process, and the process repository returns *SubcontractorIsLate* and *ContractorDoesNotPay* as exceptions for *BuyWithBilateralNegotiation*, then *ExceptionHandlingAnalysis* will add the following rules to the proposal:

```
http://xmlcontracting.org/pr.daml#hasException(co123,co123_e1);
http://xmlcontracting.org/pr.daml#SubcontractorIsLate(co123_e1);
http://xmlcontracting.org/pr.daml#hasException(co123,co123_e2);
http://xmlcontracting.org/pr.daml#ContractorDoesNotPay(co123_e2);
```

The component automatically generates unique names (`co123_e1`, `co123_e2`) for these new exception instances.

Next, the component evaluates the risk of each exception for this contract. After the user selects a *risk assessment* for an exception, the component adds it to the proposal and evaluates it to obtain the exception's conditional risk valuation based on the attributes of the buyer, seller, or product.

Finally, *Exception Handling Analysis* configures the exception handler instances for this contract. For each exception, it queries the process repository for associated exception handlers. Then it queries the contract repository for corresponding exception handler instances that can handle the given exception. Finally, the user chooses the handler instances to use for each exception, and the component adds them to the proposal.

Continuing the example, *Exception Handling Analysis* finds in the process repository that *SubcontractorIsLate* has *DetectPrerequisiteViolation* as a handler. Next, it finds the contract fragment `http://xmlcontracting.org/cf/detectLateDelivery.clp` in the contract repository as an instance of *DetectPrerequisiteViolation* that handles *SubcontractorIsLate*. The user chooses to add this handler instance to the contract, so the component adds the following rules to the proposal:

```
http://xmlcontracting.org/pr.daml#isHandledBy(co123_e1,
  http://xmlcontracting.org/cf/detectLateDelivery.clp);
_includes(http://www.contracts.com/cf/detectLateDelivery.clp);
```

7.2.5 Attached Procedure Association

Recall from Section 5.5.6 that rules may use `sensable` and `effectable` to declare that certain predicates are intended to be bound to sensor or effector attached procedures. This component prompts the user to select *attached procedure bindings* for these predicates and adds them to the proposal.

7.2.6 Completion

Completion asks the user to specify the values for the parameters in a contract. This adds `parameterValue` facts to the proposal.

7.3 Proposal Creation Wizard

The Proposal Creation Wizard demonstrates how the market agent components allow the user to create proposals through a simple point-and-click interface. In this section, we again consider a scenario where Acme is looking to purchase a plastic product from Plastics Etc. When you run the SweetDeal demo using `java sweet.deal.Demo`, *Market Agent* windows appear to represent Plastics Etc. and Acme (Figure 7-2). We assume that Acme has just sent a Request for Proposal message to Plastics Etc, so Plastics Etc. is now ready to construct the initial proposal to Acme.

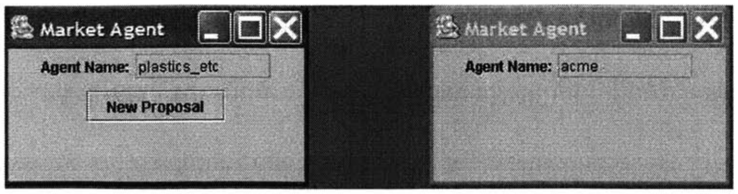
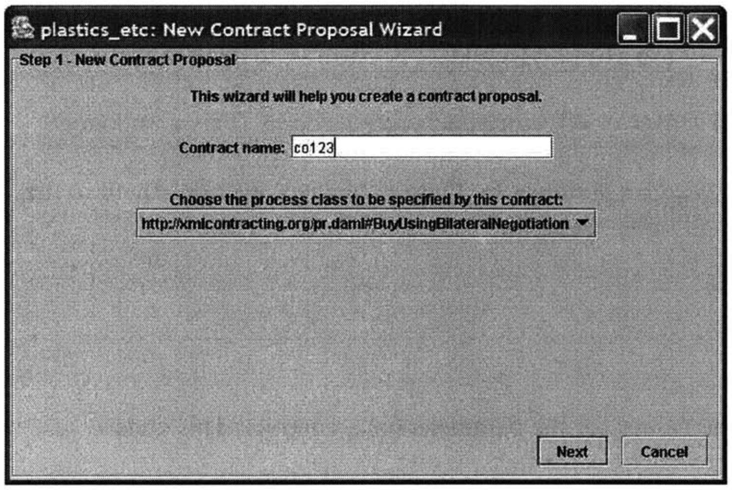


Figure 7-2: Market Agent windows

Click on "New Proposal" in the `plastics_etc` window to start the New Contract Proposal Wizard. Step 1 of the wizard will appear, allowing you to choose a name and a process class for the contract (Figure 7-3). *BuyUsingBilateralNegotiation* is the only process class available in this demonstration system.


plastics_etc: New Contract Proposal Wizard

Step 1 - New Contract Proposal

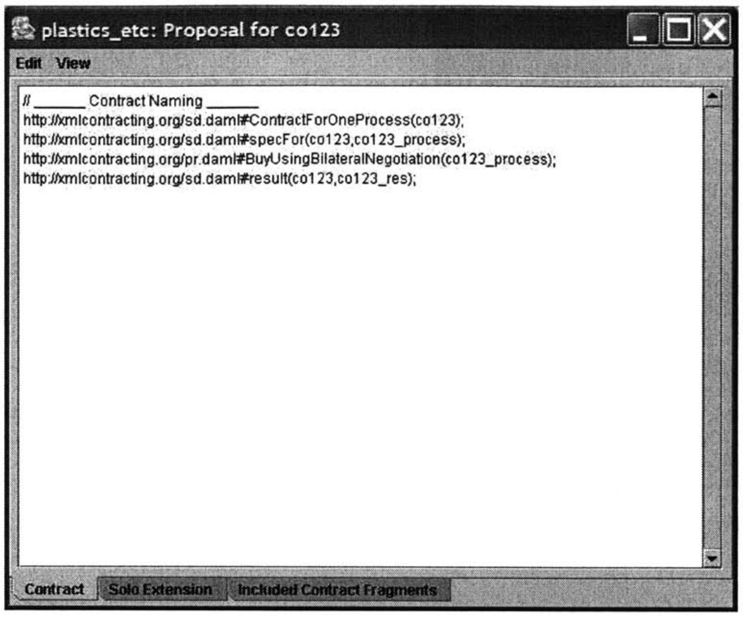
This wizard will help you create a contract proposal.

Contract name: co123

Choose the process class to be specified by this contract:
http://xmlcontracting.org/pr.daml#BuyUsingBilateralNegotiation

Figure 7-3: Step 1 - New Contract Proposal

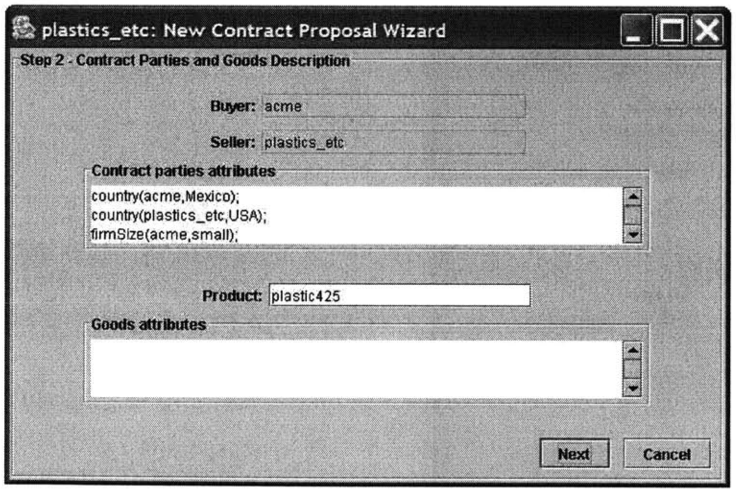
When you click Next, the *Proposal* window will appear, showing the current rules in the proposal (Figure 7-4). As you progress through the wizard, the rules that you add to the contract will appear in this window. The *Proposal* window has three tabs. The *Contract* tab shows the contract rules that will be exchanged with the other party in the negotiation (which does not include the contents of any included contract fragments). The *Solo Extension* tab shows this agent's solo extension to the contract. Finally, the *Included Contract Fragments* tab shows the contract fragments that have been included by either the contract or the solo extension.



```
// ______ Contract Naming ______
http://xmlcontracting.org/sd.daml#ContractForOneProcess(co123);
http://xmlcontracting.org/sd.daml#specFor(co123,co123_process);
http://xmlcontracting.org/pr.daml#BuyUsingBilateralNegotiation(co123_process);
http://xmlcontracting.org/sd.daml#result(co123,co123_res);
```

Figure 7-4: Contract Naming rules

In Step 2 of the wizard (Figure 7-5), you specify the buyer, seller, and product for this contract and add rules describing them. As future work, the system could be extended to acquire these attributes automatically by contacting some central repository or the agents themselves.


plastics_etc: New Contract Proposal Wizard
Step 2 - Contract Parties and Goods Description
Next
Cancel

Buyer: acme

Seller: plastics_etc

Contract parties attributes

```
country(acme,Mexico);
country(plastics_etc,USA);
firmSize(acme,small);
```

Product: plastic425

Goods attributes

Figure 7-5: Step 2 - Contract Parties and Goods Description

Step 3 (Figure 7-6) lets you choose a contract template among those that specify the process chosen in Step 1. Choosing a different contract repository, from the Contract Repository box, lets you see the templates available at that contract repository. The demo supports two contract repositories: `http://xmlcontracting.org` and the agent's internal repository.

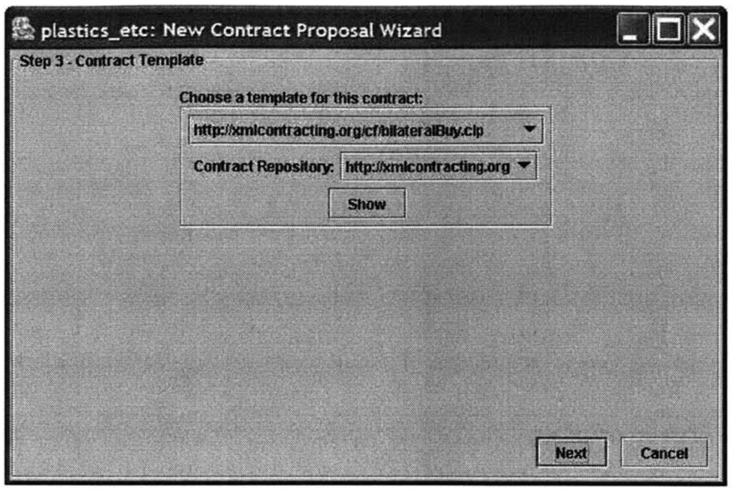
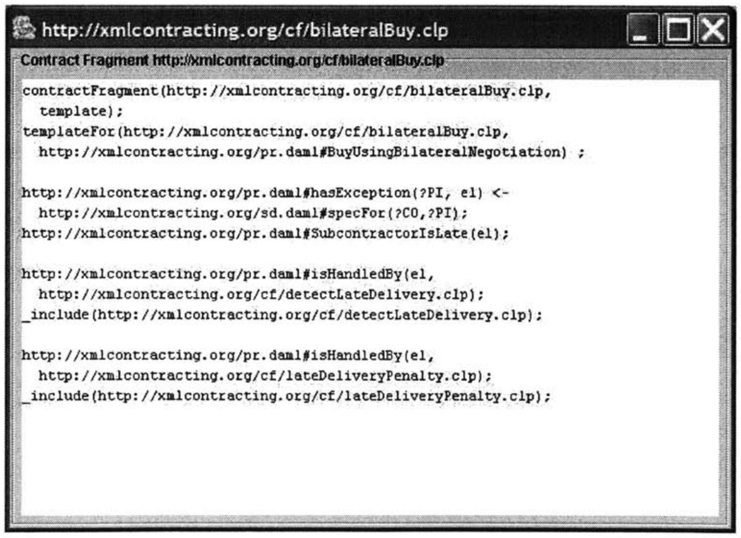

plastics_etc: New Contract Proposal Wizard
Step 3 - Contract Template
Choose a template for this contract:
http://xmlcontracting.org/cf/bilateralBuy.clp
Contract Repository: http://xmlcontracting.org
Show
Next
Cancel

Figure 7-6: Step 3 - Contract Template

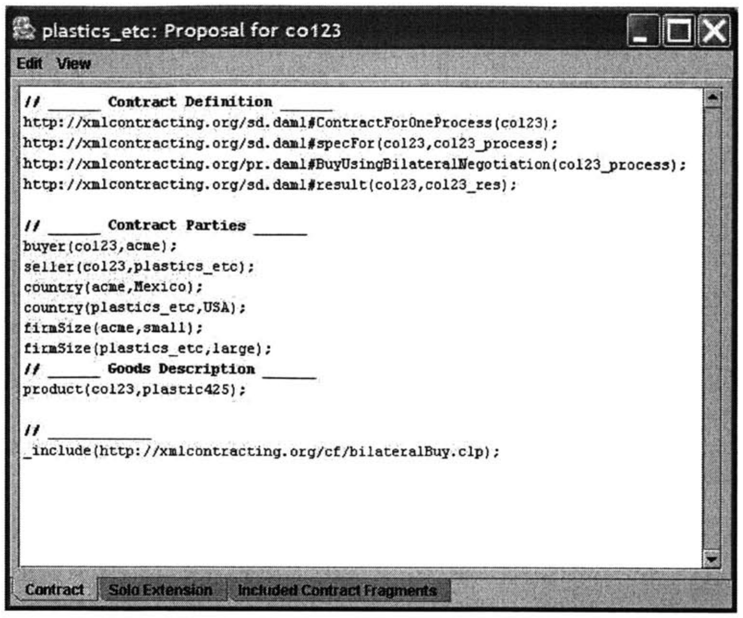
Choose the `http://xmlcontracting.org/cf/bilateralBuy.clp` template and click Show to see its rules (Figure 7-7). Notice that the template includes two exception handler contract fragments, `http://xmlcontracting.org/cf/detectLateDelivery.clp` and `http://xmlcontracting.org/cf/lateDeliveryPenalty.clp`.


http://xmlcontracting.org/cf/bilateralBuy.clp
Contract Fragment http://xmlcontracting.org/cf/bilateralBuy.clp

```
contractFragment(http://xmlcontracting.org/cf/bilateralBuy.clp,
  template);
templateFor(http://xmlcontracting.org/cf/bilateralBuy.clp,
  http://xmlcontracting.org/pr.daml#BuyUsingBilateralNegotiation) ;

http://xmlcontracting.org/pr.daml#hasException(?PI, el) <-
  http://xmlcontracting.org/sd.daml#specFor(?CO,?PI);
http://xmlcontracting.org/pr.daml#SubcontractorIsLate(el);

http://xmlcontracting.org/pr.daml#isHandledBy(el,
  http://xmlcontracting.org/cf/detectLateDelivery.clp);
_include(http://xmlcontracting.org/cf/detectLateDelivery.clp);

http://xmlcontracting.org/pr.daml#isHandledBy(el,
  http://xmlcontracting.org/cf/lateDeliveryPenalty.clp);
_include(http://xmlcontracting.org/cf/lateDeliveryPenalty.clp);
```

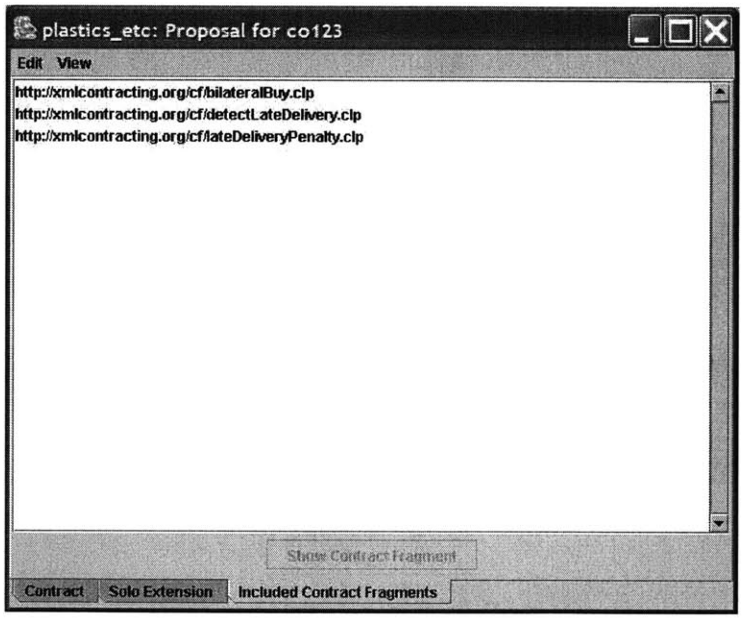
Figure 7-7: bilateralBuy.clp

Click Next to select this template. This adds an `_include` fact that incorporates this template by reference (Figure 7-8). Click on the *Included Contract Fragments* tab, and you will see the template and its two included exception handlers (Figure 7-9).



```
// ______ Contract Definition ______
http://xmlcontracting.org/sd.daml#ContractForOneProcess(col23);
http://xmlcontracting.org/sd.daml#specFor(col23,col23_process);
http://xmlcontracting.org/pr.daml#BuyUsingBilateralNegotiation(col23_process);
http://xmlcontracting.org/sd.daml#result(col23,col23_res);

// ______ Contract Parties ______
buyer(col23,acme);
seller(col23,plastics_etc);
country(acme,Mexico);
country(plastics_etc,USA);
firmSize(acme,small);
firmSize(plastics_etc,large);
// ______ Goods Description ______
product(col23,plastic425);

// ____________
_include(http://xmlcontracting.org/cf/bilateralBuy.clp);
```

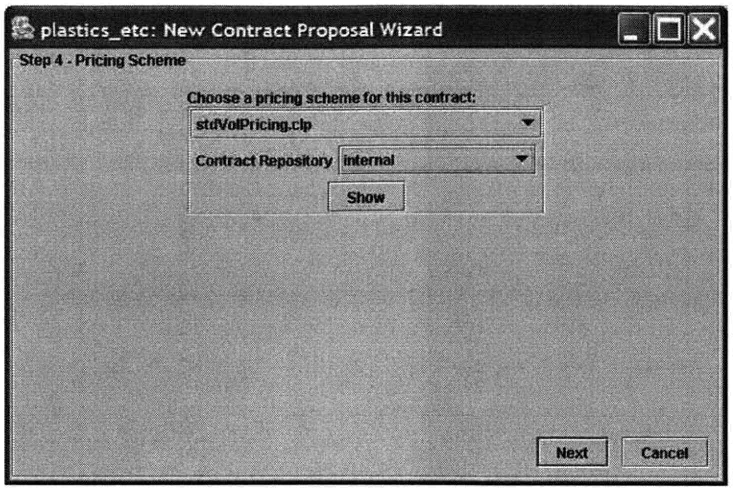
Figure 7-8: Contract that incorporates a template by reference



```
http://xmlcontracting.org/cf/bilateralBuy.clp
http://xmlcontracting.org/cf/detectLateDelivery.clp
http://xmlcontracting.org/cf/lateDeliveryPenalty.clp
```

Figure 7-9: Included contract fragments

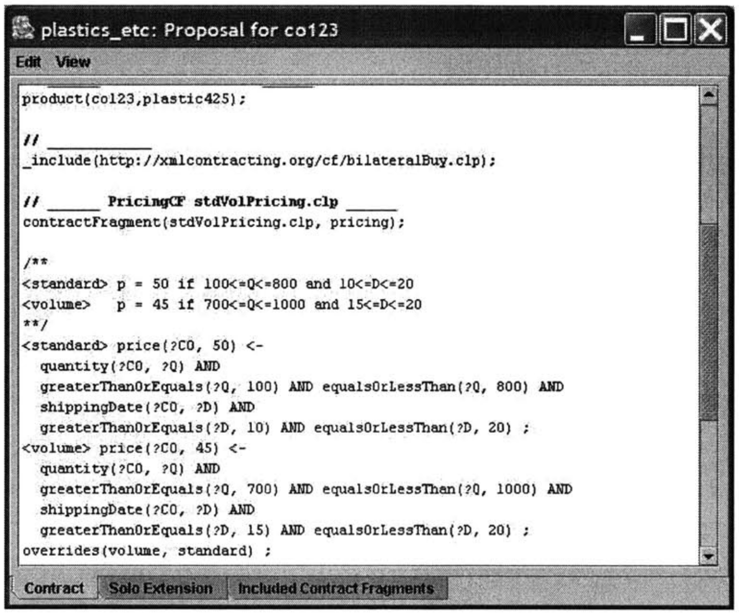
In Step 4 (Figure 7-10), you select a pricing scheme for this contract. Select the `stdVolPricing.clp` pricing scheme from the *internal repository* and click Next. Since it is from the internal repository, it is incorporated by copy and not by reference (Figure 7-11). Notice, however, that it is copied into the *shared* contract, since a pricing scheme is only useful if it is shared with the other party.



Choose a pricing scheme for this contract: stdVolPricing.clp

Contract Repository: internal

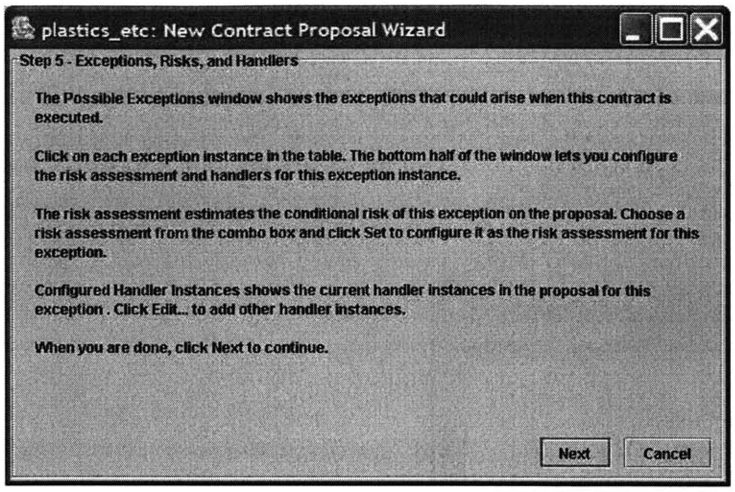
Figure 7-10: Step 4 - Pricing Scheme



```
product(col23,plastic425);

// ___________
_include(http://xmlcontracting.org/cf/bilateralBuy.clp);

// _____ PricingCF stdVolPricing.clp _____
contractFragment(stdVolPricing.clp, pricing);

/**
<standard> p = 50 if 100<=Q<=800 and 10<=D<=20
<volume>   p = 45 if 700<=Q<=1000 and 15<=D<=20
**/
<standard> price(?CO, 50) <-
  quantity(?CO, ?Q) AND
  greaterThanOrEquals(?Q, 100) AND equalsOrLessThan(?Q, 800) AND
  shippingDate(?CO, ?D) AND
  greaterThanOrEquals(?D, 10) AND equalsOrLessThan(?D, 20) ;
<volume> price(?CO, 45) <-
  quantity(?CO, ?Q) AND
  greaterThanOrEquals(?Q, 700) AND equalsOrLessThan(?Q, 1000) AND
  shippingDate(?CO, ?D) AND
  greaterThanOrEquals(?D, 15) AND equalsOrLessThan(?D, 20) ;
overrides(volume, standard) ;
```

Figure 7-11: Incorporating stdVolPricing.clp by copy

Step 5 (Figure 7-12) lets you manage the exceptions that could occur with this contract.


plastics_etc: New Contract Proposal Wizard

Step 5 - Exceptions, Risks, and Handlers

The Possible Exceptions window shows the exceptions that could arise when this contract is executed.

Click on each exception instance in the table. The bottom half of the window lets you configure the risk assessment and handlers for this exception instance.

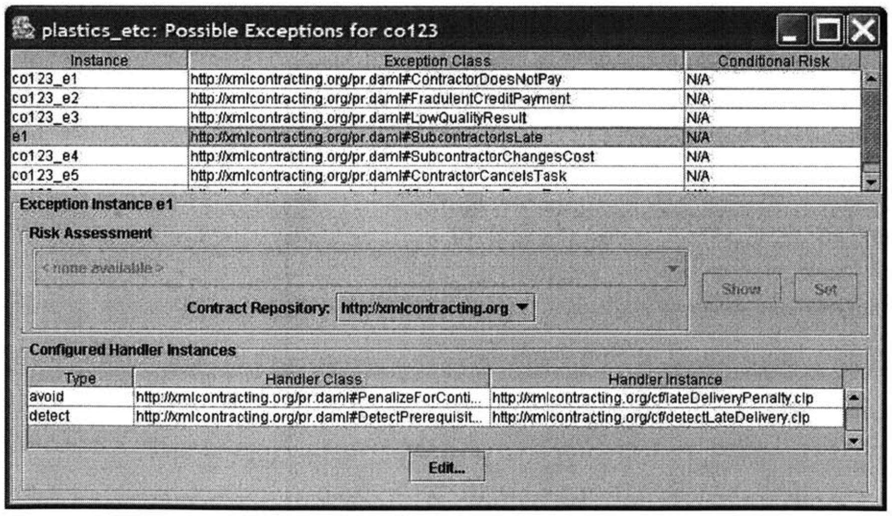
The risk assessment estimates the conditional risk of this exception on the proposal. Choose a risk assessment from the combo box and click Set to configure it as the risk assessment for this exception.

Configured Handler Instances shows the current handler instances in the proposal for this exception . Click Edit... to add other handler instances.

When you are done, click Next to continue.

Figure 7-12: Step 5 - Exceptions, Risks, and Handlers

The *Possible Exceptions* window lists the possible exceptions for this contract, based on the exception classes returned by the process repository for *BuyUsingBilateralNegotiation.* Selecting each exception instance lets you see its risk assessment and handler instances. Select `e1`, an instance of *SubcontractorIsLate*, and you will see that it already has the two handler instances specified by the template (Figure 7-13).


plastics_etc: Possible Exceptions for co123

Instance	Exception Class	Conditional Risk
co123_e1	http://xmlcontracting.org/pr.daml#ContractorDoesNotPay	N/A
co123_e2	http://xmlcontracting.org/pr.daml#FradulentCreditPayment	N/A
co123_e3	http://xmlcontracting.org/pr.daml#LowQualityResult	N/A
e1	http://xmlcontracting.org/pr.daml#SubcontractorIsLate	N/A
co123_e4	http://xmlcontracting.org/pr.daml#SubcontractorChangesCost	N/A
co123_e5	http://xmlcontracting.org/pr.daml#ContractorCancelsTask	N/A

Exception Instance e1

Risk Assessment

< none available >

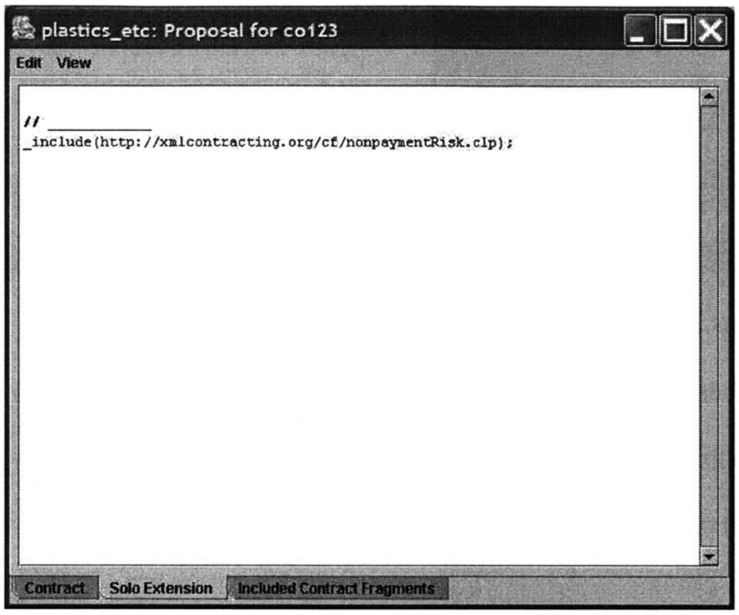
Contract Repository: http://xmlcontracting.org

Configured Handler Instances

Type	Handler Class	Handler Instance
avoid	http://xmlcontracting.org/pr.daml#PenalizeForConti...	http://xmlcontracting.org/cf/lateDeliveryPenalty.clp
detect	http://xmlcontracting.org/pr.daml#DetectPrerequisit...	http://xmlcontracting.org/cf/detectLateDelivery.clp

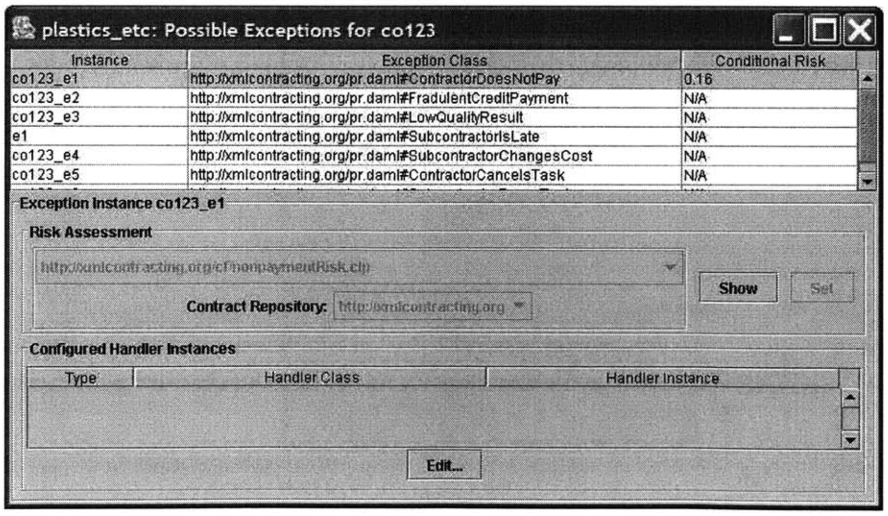
Figure 7-13: Possible Exceptions - e1

Now select `co123_e1`, an instance of *ContractorDoesNotPay*. We want to estimate the probability that the contract will encounter this exception, so we set `http://xmlcontracting.org/cf/nonpaymentRisk.clp` as its risk assessment. This adds the contract fragment to the solo extension[1] by reference (Figure 7-14) and estimates the conditional risk to be 16% (Figure 7-15). We expect high risk of nonpayment because the buyer (Acme) is a small firm and the purchase is international. (This is inferred from the contract parties attributes specified in Step 2 of the wizard.)



```
//_________
_include(http://xmlcontracting.org/cf/nonpaymentRisk.clp);
```

Figure 7-14: Risk assessment in the Solo Extension


plastics_etc: Possible Exceptions for co123
Exception Instance co123_e1
Risk Assessment
Show
Set
Configured Handler Instances
Edit...

Instance	Exception Class	Conditional Risk
co123_e1	http://xmlcontracting.org/pr.daml#ContractorDoesNotPay	0.16
co123_e2	http://xmlcontracting.org/pr.daml#FradulentCreditPayment	N/A
co123_e3	http://xmlcontracting.org/pr.daml#LowQualityResult	N/A
e1	http://xmlcontracting.org/pr.daml#SubcontractorIsLate	N/A
co123_e4	http://xmlcontracting.org/pr.daml#SubcontractorChangesCost	N/A
co123_e5	http://xmlcontracting.org/pr.daml#ContractorCancelsTask	N/A

Risk Assessment: http://xmlcontracting.org/cf/nonpaymentRisk.clp

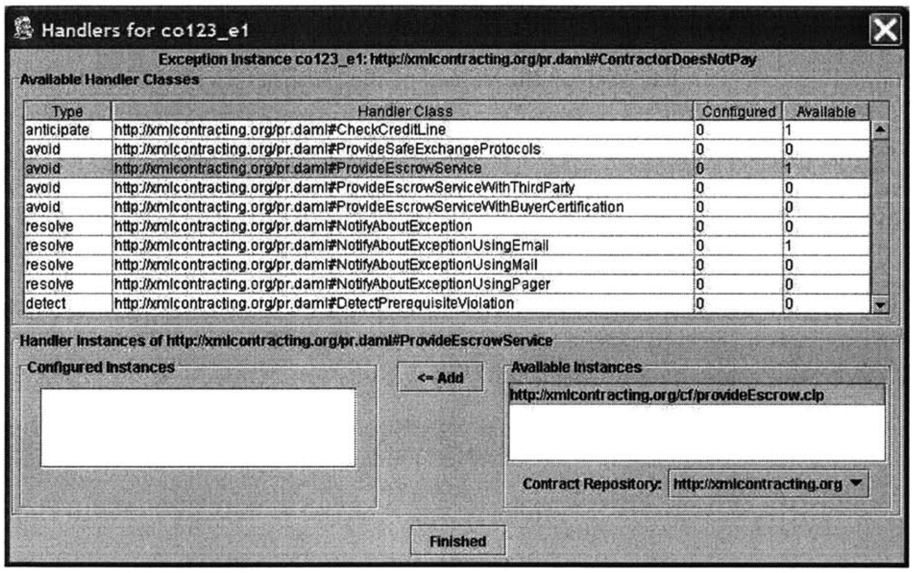
Contract Repository: http://xmlcontracting.org

Type	Handler Class	Handler Instance

Figure 7-15: Possible Exceptions - co123_e1

[1] It is added to the solo extension because Plastics Etc. may not want Acme to know which risk assessment it is using to estimate the risk.

Since `co123_e1` has such a high risk, we want to specify a handler instance to guard against it. Click Edit... to open the *Handlers* window (Figure 7-16). Notice that *CheckCreditLine*, *ProvideEscrowService* and *NotifyAboutExceptionUsingEmail* have available instances. Select *ProvideEscrowService*, and the *Available Instances* box will update to show `http://xmlcontracting.org/cf/provideEscrow.clp`. This handler instance uses a `provideEscrow` effector to establish an escrow service for the purchase. Select it and click Add. This adds rules to the contract to incorporate this handler instance by reference and specify it as a handler for `co123_e1`.



Exception Instance co123_e1: http://xmlcontracting.org/pr.daml#ContractorDoesNotPay

Available Handler Classes

Type	Handler Class	Configured	Available
anticipate	http://xmlcontracting.org/pr.daml#CheckCreditLine	0	1
avoid	http://xmlcontracting.org/pr.daml#ProvideSafeExchangeProtocols	0	0
avoid	http://xmlcontracting.org/pr.daml#ProvideEscrowService	0	1
avoid	http://xmlcontracting.org/pr.daml#ProvideEscrowServiceWithThirdParty	0	0
avoid	http://xmlcontracting.org/pr.daml#ProvideEscrowServiceWithBuyerCertification	0	0
resolve	http://xmlcontracting.org/pr.daml#NotifyAboutException	0	0
resolve	http://xmlcontracting.org/pr.daml#NotifyAboutExceptionUsingEmail	0	1
resolve	http://xmlcontracting.org/pr.daml#NotifyAboutExceptionUsingMail	0	0
resolve	http://xmlcontracting.org/pr.daml#NotifyAboutExceptionUsingPager	0	0
detect	http://xmlcontracting.org/pr.daml#DetectPrerequisiteViolation	0	0

Handler Instances of http://xmlcontracting.org/pr.daml#ProvideEscrowService

Configured Instances

<= Add

Available Instances

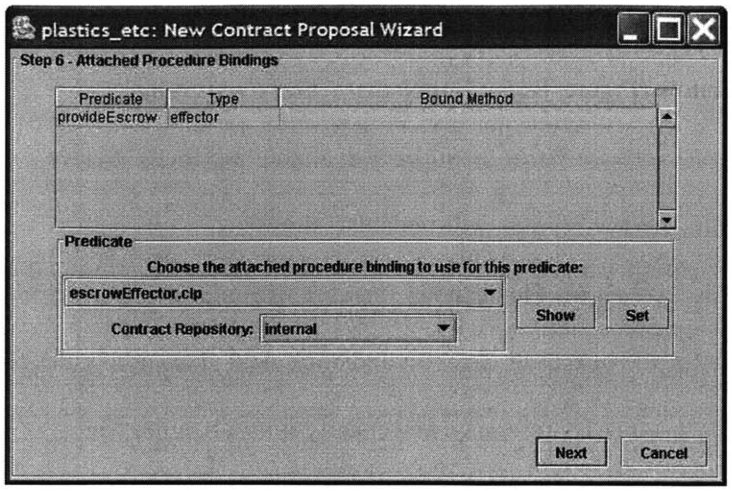
http://xmlcontracting.org/cf/provideEscrow.clp

Contract Repository: http://xmlcontracting.org

Finished

Figure 7-16: Handlers for co123_e1

Step 6 (Figure 7-17) lets you configure the bindings for any attached procedures in the contract. Choose `escrowEffector.clp` from the internal repository as the binding for `provideEscrow`. The system will ask whether to add the binding to the Contract or the Solo Extension (Figure 7-18). We choose to add it to the Contract because both parties have to execute this effector in order to set up the escrow service.

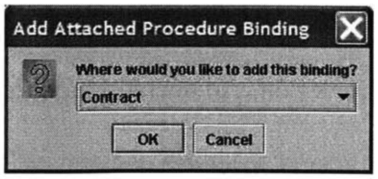

plastics_etc: New Contract Proposal Wizard
Step 6 - Attached Procedure Bindings
Predicate
Choose the attached procedure binding to use for this predicate:
Contract Repository:
Show
Set
Next
Cancel

Predicate	Type	Bound Method
provideEscrow	effector	

escrowEffector.clp

Contract Repository: internal

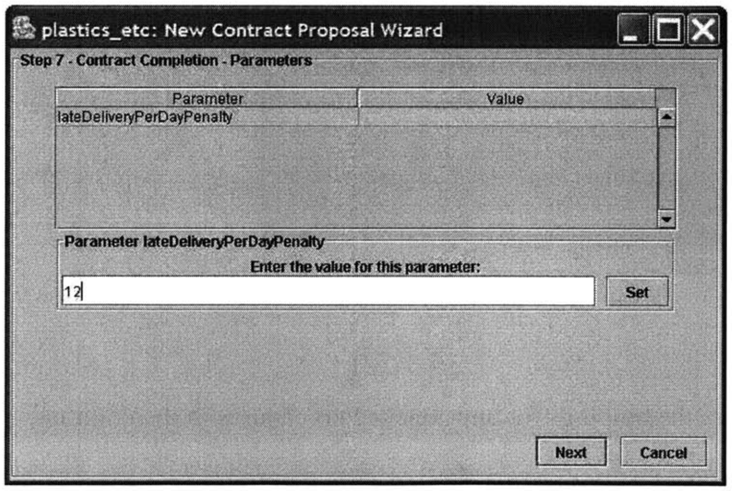
Figure 7-17: Step 6 - Attached Procedure Binding


Add Attached Procedure Binding
Where would you like to add this binding?
OK
Cancel

Contract

Figure 7-18: Adding an attached procedure binding

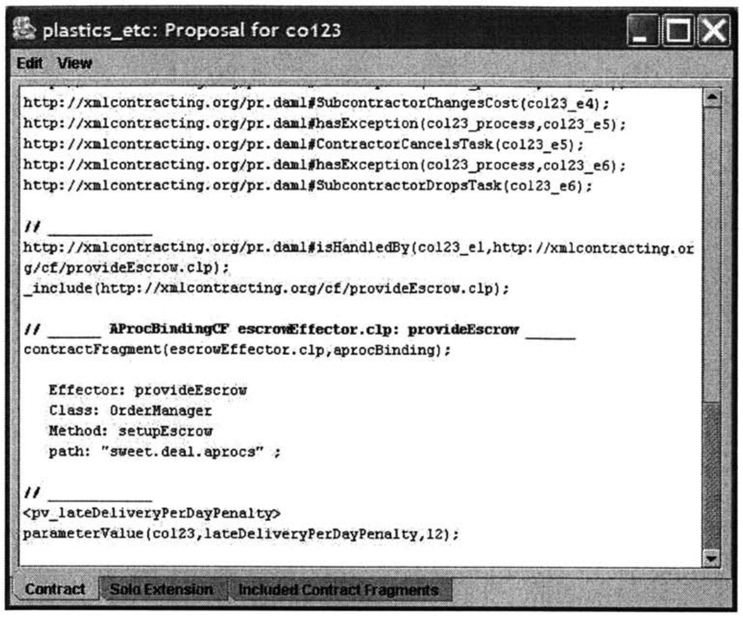
The last step, *Completion* (Figure 7-19), lets you set the values of the parameters in the contract. Recall that `http://xmlcontracting.org/cf/lateDeliveryPenalty.clp` introduced the `lateDeliveryPerDayPenalty` parameter. Here we set the penalty to $12 per day, which adds the corresponding `parameterValue` statement to the contract (Figure 7-20).


plastics_etc: New Contract Proposal Wizard
Step 7 - Contract Completion - Parameters
Parameter lateDeliveryPerDayPenalty
Enter the value for this parameter:
Set
Next
Cancel

Parameter	Value
lateDeliveryPerDayPenalty	

12

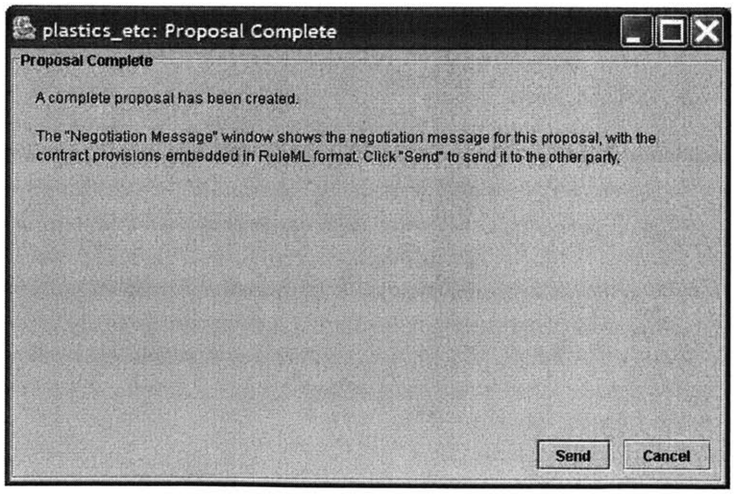
Figure 7-19: Step 7 - Contract Completion



```
http://xmlcontracting.org/pr.daml#SubcontractorChangesCost(col23_e4);
http://xmlcontracting.org/pr.daml#hasException(col23_process,col23_e5);
http://xmlcontracting.org/pr.daml#ContractorCancelsTask(col23_e5);
http://xmlcontracting.org/pr.daml#hasException(col23_process,col23_e6);
http://xmlcontracting.org/pr.daml#SubcontractorDropsTask(col23_e6);

// ____________
http://xmlcontracting.org/pr.daml#isHandledBy(col23_e1,http://xmlcontracting.or
g/cf/provideEscrow.clp);
_include(http://xmlcontracting.org/cf/provideEscrow.clp);

// ______ AProcBindingCF escrowEffector.clp: provideEscrow ______
contractFragment(escrowEffector.clp,aprocBinding);

   Effector: provideEscrow
   Class: OrderManager
   Method: setupEscrow
   path: "sweet.deal.aprocs" ;

// ____________
<pv_lateDeliveryPerDayPenalty>
parameterValue(col23,lateDeliveryPerDayPenalty,12);
```

Figure 7-20: Contract - parameterValue

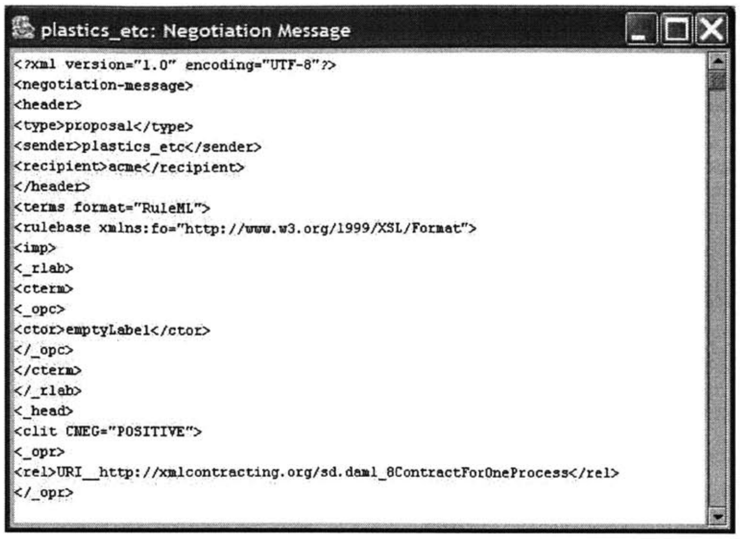
Now the proposal is complete (Figure 7-21). The *Negotiation Message* window appears, showing the XML message that will be sent to the other party (Figure 7-22). This message includes the contract terms encoded in RuleML format. Internally, the system uses IBM CommonRules to translate the terms from the SCLPfile format to BRML and an early version of SweetRules to translate from BRML to RuleML.



A complete proposal has been created.

The "Negotiation Message" window shows the negotiation message for this proposal, with the contract provisions embedded in RuleML format. Click "Send" to send it to the other party.

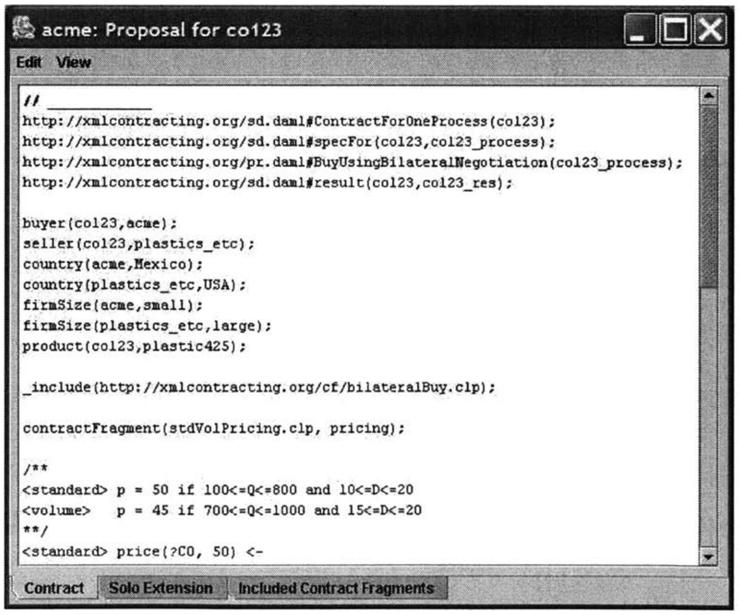
Figure 7-21: Proposal Complete



```
<?xml version="1.0" encoding="UTF-8"?>
<negotiation-message>
<header>
<type>proposal</type>
<sender>plastics_etc</sender>
<recipient>acme</recipient>
</header>
<terms format="RuleML">
<rulebase xmlns:fo="http://www.w3.org/1999/XSL/Format">
<imp>
<_rlab>
<cterm>
<_opc>
<ctor>emptyLabel</ctor>
</_opc>
</cterm>
</_rlab>
<_head>
<clit CNEG="POSITIVE">
<_opr>
<rel>URI__http://xmlcontracting.org/sd.daml_8ContractForOneProcess</rel>
</_opr>
```

Figure 7-22: Negotiation Message

Click Send to send this negotiation message to the other party. Acme receives the message and parses it to obtain the contract terms. Acme's *Proposal* window appears (Figure 7-23). Notice that the *Contract* tab contains all the contract rules from Plastics Etc, but the *Solo Extension* tab is empty (Figure 7-24), since those rules were not sent to Acme.



```
// ___________
http://xmlcontracting.org/sd.daml#ContractForOneProcess(co123);
http://xmlcontracting.org/sd.daml#specFor(co123,co123_process);
http://xmlcontracting.org/pr.daml#BuyUsingBilateralNegotiation(co123_process);
http://xmlcontracting.org/sd.daml#result(co123,co123_res);

buyer(co123,acme);
seller(co123,plastics_etc);
country(acme,Mexico);
country(plastics_etc,USA);
firmSize(acme,small);
firmSize(plastics_etc,large);
product(co123,plastic425);

_include(http://xmlcontracting.org/cf/bilateralBuy.clp);

contractFragment(stdVolPricing.clp, pricing);

/**
<standard> p = 50 if 100<=Q<=800 and 10<=D<=20
<volume>   p = 45 if 700<=Q<=1000 and 15<=D<=20
**/
<standard> price(?CO, 50) <-
```

Figure 7-23: Acme's Proposal - Contract

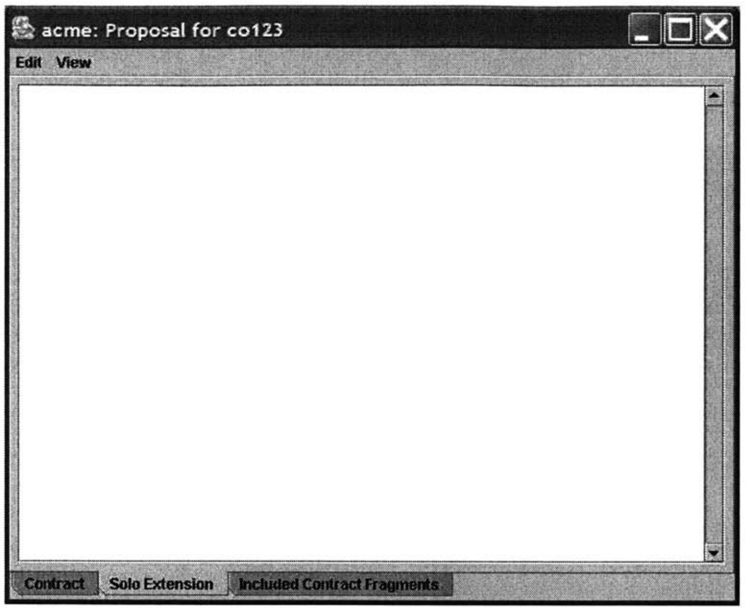
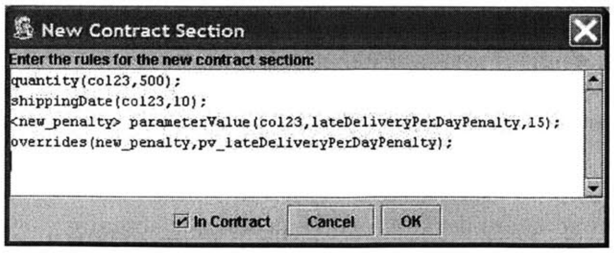


Figure 7-24: Acme's Proposal - Solo Extension

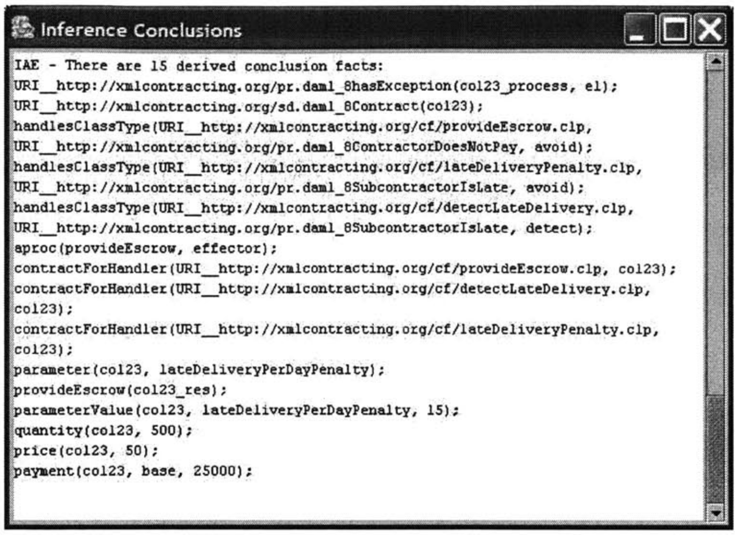
Now Acme can add some rules to create a counterproposal. Suppose that Acme wants to order 500 units with a shipping date of 10 days after the order. In addition, it feels that the late delivery penalty of $12 per day is too little; it would prefer $15 instead. Acme can add the appropriate rules by selecting *Add a contract section* from the Edit menu of its *Proposal* window (Figure 7-25). Notice that the `parameterValue` fact sets the penalty to $15, and an overrides statement declares that this new `parameterValue` takes precedence over the original. (A built-in MUTEX rule ensures that each parameter can only have one value. See the Appendix.)



```
quantity(col23,500);
shippingDate(col23,10);
<new_penalty> parameterValue(col23,lateDeliveryPerDayPenalty,15);
overrides(new_penalty,pv_lateDeliveryPerDayPenalty);
```

Figure 7-25: Adding rules to change a parameter value

Selecting *Inference conclusions* from the View menu of the *Proposal* window shows all the conclusions that the inferencing engine deduces from the rules in the contract, the solo extension, and the included contract fragments (Figure 7-26). Notice that it infers that the value of `lateDeliveryPerDayPenalty` is 15. In addition, from the pricing scheme, the quantity, and the shipping date, the system infers a price of \$50 and a base payment of \$50 x 500 = \$25,000.



```
IAE - There are 15 derived conclusion facts:
URI__http://xmlcontracting.org/pr.daml_8hasException(col23_process, el);
URI__http://xmlcontracting.org/sd.daml_8Contract(col23);
handlesClassType(URI__http://xmlcontracting.org/cf/provideEscrow.clp,
URI__http://xmlcontracting.org/pr.daml_8ContractorDoesNotPay, avoid);
handlesClassType(URI__http://xmlcontracting.org/cf/lateDeliveryPenalty.clp,
URI__http://xmlcontracting.org/pr.daml_8SubcontractorIsLate, avoid);
handlesClassType(URI__http://xmlcontracting.org/cf/detectLateDelivery.clp,
URI__http://xmlcontracting.org/pr.daml_8SubcontractorIsLate, detect);
aproc(provideEscrow, effector);
contractForHandler(URI__http://xmlcontracting.org/cf/provideEscrow.clp, col23);
contractForHandler(URI__http://xmlcontracting.org/cf/detectLateDelivery.clp,
col23);
contractForHandler(URI__http://xmlcontracting.org/cf/lateDeliveryPenalty.clp,
col23);
parameter(col23, lateDeliveryPerDayPenalty);
provideEscrow(col23_res);
parameterValue(col23, lateDeliveryPerDayPenalty, 15);
quantity(col23, 500);
price(col23, 50);
payment(col23, base, 25000);
```

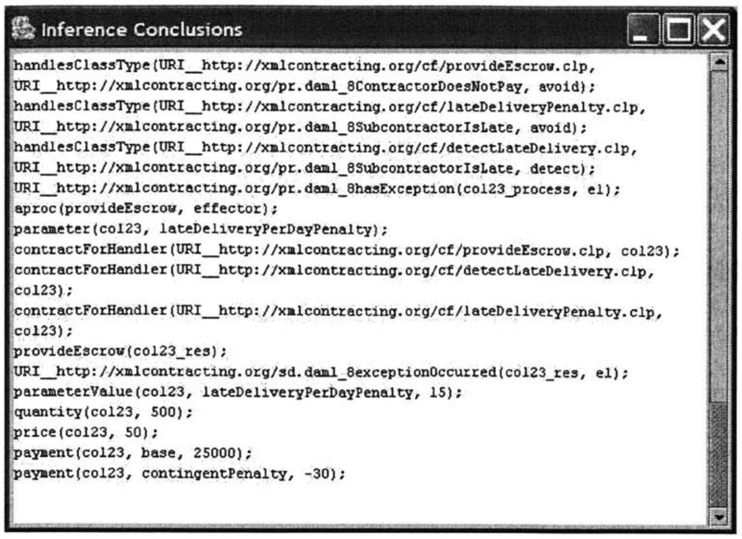
Figure 7-26: Inference conclusions[1]

In this way, Acme is enabled to modify the contract terms and send a counterproposal back to Plastics Etc. This exchange of counterproposals continues until the provisions are acceptable to both parties and a final contract is established. Alternatively, a negotiation deadline might force the parties to terminate their exchange of counterproposals before an agreement is reached, but this is not implemented in the current prototype.

After negotiation, the execution phase begins. Although this prototype does not include an *Execution* component, we can simulate it by adding the appropriate *ContractResult* facts. Select *Add a contract section* again and add the following fact to declare that the product is actually shipped 12 days after the order (i.e. 2 days later than the 10 days specified in the contract):

```
shippingDate(col23_res,12);
```

[1] The URIs here are encoded so that they are accepted as predicates by the IBM CommonRules rule engine. See Section 8.3.



```
handlesClassType(URI__http://xmlcontracting.org/cf/provideEscrow.clp,
URI__http://xmlcontracting.org/pr.daml_8ContractorDoesNotPay, avoid);
handlesClassType(URI__http://xmlcontracting.org/cf/lateDeliveryPenalty.clp,
URI__http://xmlcontracting.org/pr.daml_8SubcontractorIsLate, avoid);
handlesClassType(URI__http://xmlcontracting.org/cf/detectLateDelivery.clp,
URI__http://xmlcontracting.org/pr.daml_8SubcontractorIsLate, detect);
URI__http://xmlcontracting.org/pr.daml_8hasException(col23_process, e1);
aproc(provideEscrow, effector);
parameter(col23, lateDeliveryPerDayPenalty);
contractForHandler(URI__http://xmlcontracting.org/cf/provideEscrow.clp, col23);
contractForHandler(URI__http://xmlcontracting.org/cf/detectLateDelivery.clp,
col23);
contractForHandler(URI__http://xmlcontracting.org/cf/lateDeliveryPenalty.clp,
col23);
provideEscrow(col23_res);
URI__http://xmlcontracting.org/sd.daml_8exceptionOccurred(col23_res, e1);
parameterValue(col23, lateDeliveryPerDayPenalty, 15);
quantity(col23, 500);
price(col23, 50);
payment(col23, base, 25000);
payment(col23, contingentPenalty, -30);
```

Figure 7-27: Inference conclusions from simulated contract execution

Open the *Inference Conclusions* window again (Figure 7-27). Notice that the system inferred from the rules in `http://xmlcontracting.org/cf/detectLateDelivery.clp` that the `e1` (i.e. *SubcontractorIsLate*) exception has occurred. In addition, the system used the rules in `http://xmlcontracting.org/cf/lateDeliveryPenalty.clp` to conclude that there is a contingent penalty of (2 days late x $15 per day late) = $30. With rule inferencing, exception handler instances are automatically executed when the appropriate situations arise in the *ContractResult*.

8 Implementation

SweetDeal is implemented in Java and compatible with JDK 1.3 or later. The system consists of several Java packages, shown in Table 8-1. The prototype code and documentation will be available for download at `http://ebusiness.mit.edu/bgrosof/#sweet`.

Package	Description
sweet.deal	Main SweetDeal classes, used by multiple system components
sweet.deal.processrep	Process repository classes
sweet.deal.contractrep	Contract repository classes
sweet.deal.agent	Market agent classes
sweet.deal.agent.wizard	Proposal Creation Wizard classes
sweet.deal.aprocs	Dummy implementations of attached procedures used in the examples (ex. `provideEscrow`)
sweet.deal.ui	Utility classes for miscellaneous user interface elements

Table 8-1: Java packages in SweetDeal

`sweet.deal.Demo` provides a demonstration of the system, allowing you to run the Proposal Creation Wizard and browse the process and contract repositories.

8.1 Process Repository

The implementation of the process repository uses Jena [23], a software package from Hewlett-Packard Labs, to parse and process the DAML+OIL process ontology. At startup, the process repository calls Jena to load the ontology into memory. Whenever it receives a process knowledge query, the process repository uses the Jena API to programmatically traverse the ontology to find the answer, as described below.

8.1.1 Interface

```
public String[] getExceptions(String process, boolean withSubclasses)
public String[] getHandlers(String exception, boolean withSubclasses)
public boolean hasSubclass(String class1, String class2, boolean closed)
```

8.1.2 Query Handling

Currently, Jena does not provide a reasoning engine for DAML+OIL, although this is planned for future versions. (Other researchers [42] are also working on DAML+OIL reasoning and inferencing.) However, its existing attribute access and hierarchy traversal capabilities are sufficient to implement the simple queries we require.

First consider `hasSubclass`. This is trivial to implement, since we directly use Jena's `hasSubclass` method to determine whether one DAML class is a subclass of another class.

The implementations of `getExceptions` and `getHandlers` are more involved. First consider `getExceptions`. Recall from Section 4.1.2 that each *hasException* link from a process P to an exception E in the MIT Process Handbook is represented as a `daml:Restriction` that is a superclass of P in the DAML+OIL ontology. This restriction declares that the *hasException* property could have values of class E. Therefore, to find all the exceptions for a process P, we examine the superclasses of P that are restrictions for the *hasException* property and add all of their `hasClass` values to the result set. If `withSubclasses` is on, we add the subclasses of these `hasClass` values as well.

`getHandlers` is implemented analogously, by looking for superclasses of the exception that are restrictions on the *isHandledBy* property.

8.2 Contract Repository

The contract repository has a simple interface, allowing agents to retrieve a contract fragment given its URI and to query for contract fragments based on certain atttributes:

```
public String getContractFragment(String uri)
public String[] getTemplates(String process)
public String[] getRiskAssessments(String exception)

public String[] getPricingSchemes()

public String[] getHandlerInsts(String handler)

public String[] getHandlerInsts(String handler, String exception)

public String[] getAProcBindings(String procedure)
```

As described in Section 6.3, the contract repository indexes contract fragments by their attributes. Queries are answered by simply retrieving the appropriate entries from the index.

8.3 Market Agent

We use IBM CommonRules [13] as the inferencing engine for contract rules. Note that there are certain characters, such as '#', '-', and '~', that may be used in a URI but cannot be part of predicate names in CommonRules. SweetDeal includes a Java class, `sweet.deal.URIEncode`, that encodes and decodes URIs in a format that can be used as CommonRules predicates. For example, `http://xmlcontracting.org/pr.daml#hasException` is encoded as `_uri_http://xmlcontracting.org/pr.daml_8hasException`. We avoid name clashes by encoding any '_' characters in the URI as well.

8.4 Limitations

To facilitate development and testing, the system components currently communicate with each other via local Java calls (recall Section 3.3). However, since we defined all interface methods to use either primitive Java types (e.g. `String`, `boolean`, `int`) or the XML `org.w3c.dom.Element` type, it should be relatively straightforward to extend the system to use SOAP-RPC calls. For example, the Web Services Developer Pack [35] from Sun Microsystems can take a Java interface and automatically generate corresponding SOAP clients and servers.

9 Conclusion

This thesis presents SweetDeal, a novel approach for automated contracting that combines a rule-based contract representation with knowledge about business processes and exceptions. This enables contracts to include provisions that manage exception conditions which may arise during contract execution, a vital capability for electronic marketplaces. We use emerging Semantic Web formats (DAML+OIL and RuleML, respectively) to represent process knowledge and contract provisions in a way that allows the rule-based provisions to reference process knowledge. This work helped us identify substantive limitations of DAML+OIL (inheritance overrides) and RuleML (incorporation by reference) that could be basis for future work. In addition, this thesis defines mechanisms for storing and querying these two types of business knowledge. A process repository stores and retrieves process knowledge, while a contract repository stores and retrieves modular contract provisions. Finally, we specify and implement the mechanisms of a software agent that uses the knowledge from these repositories to create contracts with substantial automation. We suggest that the representations and mechanisms developed in this thesis could serve as the foundation for automated evaluation, negotiation, and execution of contracts as well.

We believe that the widespread adoption of rule-based contracts with exception handling provisions would result in substantial benefits for electronic marketplaces. Besides protecting against misbehavior, such contracts incentivize market participants to reduce their likelihood of committing exceptions, for example by improving their internal business processes. Eventually, this would cause exception conditions to occur less frequently overall, increasing the efficiency of the marketplace.

For a detailed enumeration of the contributions of this thesis, see the Introduction (Chapter 1).

While performing the research for this thesis, we uncovered many related areas that could be explored in the future. Next we sketch these future research directions.

9.1 Future Directions

9.1.1 Automated Contract Evaluation and Negotiation

As described in Chapter 7, this thesis focuses on automated contract creation. Future research could involve automated contract evaluation and negotiation. An agent could evaluate contracts automatically using a utility function based on its preferences. In turn, this would allow it to compare alternate contract provisions for counterproposals, which forms the basis for automated negotiation [9][32][33].

9.1.2 Other Types of Contracting Parties

This thesis considers the simplest case of a buyer contracting to buy some product from a seller. A natural extension is to consider contracts involving intermediaries like auctioneers. The auctioneer agrees to conduct an auction specified by certain terms and conditions, and the seller and bidders agree to abide by those terms during the auction process. The MIT Process Handbook is rich in content on exceptions and exception handlers for various types of auctions.

9.1.3 Richer Negotiation Protocol

Future work could provide a richer negotiation protocol and allow for more types of negotiation messages than the simple proposal/counterproposal exchange described in this thesis. This protocol could be based on the FIPA ACL (Foundation for Intelligent Physical Agents, Agent Communication Language) specifications [28].

9.1.4 SOAP-Enabling the System Components

As described in Section 8.4, the system components (e.g. market agents, contract repository, process repository) currently communicate with each other via local Java calls. Extending the system to use SOAP messages over HTTP would allow system components to be located anywhere on the Internet and make it possible for other software systems (perhaps implemented in other programming languages) to access these system components remotely.

9.1.5 Extending RuleML With Incorporation by Reference

As explained in Section 6.1, RuleML does not currently have a mechanism that allows rules to include other rulesets by reference. Such a mechanism may involve XInclude [26].

9.1.6 Property Inheritance Overrides in a DAML+OIL Ontology

As described in Section 4.1.3, there is no direct way of expressing overrides of property inheritance in DAML+OIL. One possible solution is to augment the DAML+OIL ontology with RuleML rules that use the priority overrides mechanism to explicitly implement property overrides.

9.1.7 Event Delivery for Situated Procedures

Currently, SCLP defines sensors as a *pull* mechanism for the rule system to acquire data from other software systems. If a sensor's value changes over time, the rule engine has to call the sensor periodically to obtain any updated data. This polling may be impractical and expensive performance-wise, especially for sensors like exception detectors that rarely change their values but require the system to recognize these changes immediately. Future versions of SCLP could define a *push* mechanism (e.g. event delivery) [44] for data acquisition, where an event-deliverer attached procedure notifies the rule system whenever its data value changes.

9.1.8 Inferencing Performance

For large contracts with many rules, the rule engine's inferencing performance may become an issue. One optimization is *partial evaluation*, where the rule engine evaluates a few select rules to determine a subset of rules that will never fire in this session, based on current instance facts. Then the rule engine can simply ignore that subset in subsequent reasoning. For example, if a contract has some tariff rules (perhaps from an included contract fragment) but the buyer and seller are both in the same country, then the rule engine could ignore all the tariff rules in its reasoning.

9.1.9 Contract Fragment Integrity and Access Control

As discussed in Section 6.4, it may be important to ensure that a referenced contract fragment does not change and to allow creators to limit the parties that can access their contract fragments.

9.1.10 Advanced Query for Contract Fragments

The current contract repository supports simple queries for contract fragments, based on one or two attributes specific to each contract fragment type. For example, queries for exception handler instances are based on either a handler class or a (handler class, exception class) pair. More advanced queries based on the content of contract fragments may be useful for repositories with large numbers of contract fragments.

10 References

1. B. N. Grosof, Y. Labrou, and H. Y. Chan, "A Declarative Approach to Business Rules in Contracts: Courteous Logic Programs in XML." In *Proc. First ACM Conference on Electronic Commerce (EC99)*, 1999.

2. P. Maes, R. Guttman, and A. Moukas, "Agents that Buy and Sell: Transforming Commerce as We Know It." *Communications of the ACM*, March 1999.

3. C. Baral and M. Gelfond, "Logic Programming and Knowledge Representation." *Journal of Logic Programming* 19,20:73-148, 1994.

4. EECOMS Project. http://www.research.ibm.com/rules/eecoms.html

5. D. M. Reeves, M. P. Wellman, and B. N. Grosof, "Automated Negotiation From Declarative Contract Descriptions." To appear in *Computational Intelligence*, special issue on Agent Technology for Electronic Commerce, 2002. (Revised and extended from 2001 Autonomous Agents conference paper.)

6. Malone, T.W., et. al., "Tools for Inventing Organizations: Toward a Handbook of Organizational Processes." *Management Science*, 1999, **45**(3): p. 425-443.

7. M. Klein, C. Dellarocas, and J. A. Rodríguez-Aguilar, "A Knowledge-Based Methodology for Designing Robust Multi-Agent Systems." In *Proc. Autonomous Agents and Multi-Agent Systems*, 2002.

8. G. Wyner and J. Lee, "Applying Specialization to Process Models." In *Proc. Conference on Organizational Computing Systems*, 1995.

9. M. Klein, P. Faratin, H. Sayama, and Y. Bar-Yam, "Negotiating Complex Contracts." In *Proc. Autonomous Agents and Multi-Agent Systems*, 2002.

10. Rule Markup Language. http://www.dfki.uni-kl.de/ruleml/

11. B. N. Grosof, "Standardizing XML Rules: Preliminary Outline of Invited Talk." In *Proc. IJCAI-01 Workshop on E-business and the Intelligent Web*, 2001.

12. B. N. Grosof, "Building Commercial Agents: An IBM Research Perspective (Invited Talk)". In *Proc. 2nd Intl. Conference and Exhibition on Practical Applications of Intelligent Agents and Multi-Agent Technology (PAAM97)*, 1997.

13. IBM CommonRules – Business Rules for Electronic Commerce. http://www.research.ibm.com/rules/

14. Requirements for a Web Ontology Language (W3C Working Draft 07 March 2002). http://www.w3.org/TR/2002/WD-webont-req-20020307/

15. B. N. Grosof, "Automating Law in the Small: Contracts, Regulations, and Prioritized Argumentation", 1-hour Invited Address. Presented at the 2001 International Conference on Artificial Intelligence and Law (ICAIL-2001), held Washington University Law School, St. Louis, MO, USA, May 21-25, 2001.

16. DARPA Agent Markup Language. http://www.daml.org

17. Ontology Inferencing Language. http://www.ontoknowledge.org/oil

18. DAML+OIL (March 2001) Reference Description. http://www.w3.org/TR/daml+oil-reference

19. W3C Web Ontology (WebOnt) Working Group. http://www.w3.org/2001/sw/WebOnt/

20. Resource Description Framework (RDF). http://www.w3.org/RDF/

21. Annotated DAML+OIL (March 2001) Ontology Markup. http://www.daml.org/2001/03/daml+oil-walkthru.html

22. DAML Tools. http://www.daml.org/tools/

23. Hewlett-Packard Labs Semantic Web Activity – Jena Toolkit. http://www.hpl.hp.com/semweb/jena-top.html

24. OILEd Ontology Editor. http://www.ontoknowledge.org/oil/tool.shtml

25. A. Bernstein, M. Klein, and T.W. Malone., "The Process Recombinator: A Tool for Generating New Business Process Ideas". In *Proc. International Conference on Information Systems (ICIS-99)*, 1999.

26. XML Inclusions (XInclude) Version 1.0. http://www.w3.org/TR/xinclude/

27. CommonAccord. http://www.commonaccord.org/

28. Foundation for Intelligent Physical Agents, Agent Communication Language Specifications. http://www.fipa.org/repository/aclspecs.html

29. Hypertext Style: Cool URIs don't change. http://www.w3.org/Provider/Style/URI.html

30. World Wide Web Consortium, Semantic Web Activity. http://www.w3.org/2001/sw/

31. Web Naming and Addressing Overview (URIs, URLs, ...). http://www.w3.org/Addressing/

32. P. Faratin and M. Klein, "Automated Contract Negotiation and Execution as a System of Constraints." In *Proc. Workshop on Distributed Constraint Reasoning (IJCAI-01)*, Seattle, USA, 33—45, 2001.

33. N. R. Jennings, P. Faratin, A.R. Lomuscio, S. Parsons, C. Sierra, and M. Wooldridge, "Automated Negotiation: Prospects, Methods, and Challenges." *International Journal of Group Decision and Negotiation*, 10 (2) 199-215, 2001.

34. Michael P. Wellman, Amy Greenwald, Peter Stone, and Peter R. Wurman, "The 2001 Trading Agent Competition." In *Proc. 14th Conference on Innovative Applications of Artificial Intelligence*, Edmonton, Canada, 2002.

35. Java Web Services Developer Pack. http://java.sun.com/webservices/webservicespack.html

36. Simple Object Access Protocol (SOAP). http://www.w3.org/TR/SOAP/

37. United States Government, Internet Fraud Complaint Center, *IFCC 2001 Annual Internet Fraud Report*. April 2002. www.ifccfbi.gov

38. B. N. Grosof and Y. Labrou, "An Approach to using XML and a Rule-based Content Language with an Agent Communication Language." In *Proc. IJCAI-99 Workshop on Agent Communication Languages (ACL-99)*, Stockholm, Sweden, 1999. Revised version appears in F. Dignum, M. Greaves (Eds.), *Issues in Agent Communication*, Lecture Notes in Computer Science Vol. 1916, Springer-Verlag, Berlin, German, 2000.

39. D. M. Reeves, B. N. Grosof, M. P. Wellman, and H. Y. Chan, "Towards a Declarative Language for Negotiating Executable Contracts." In *Proc. AAAI-99 Workshop on Artificial Intelligence in Electronic Commerce (AIEC-99)*, Orlando, FL, USA, 1999.

40. D. Koller and A. Pfeiffer, "Object-Oriented Bayesian Networks." *In Proc. 13th Annual Conference on Uncertainty in AI (UAI)*, Providence, Rhode Island, USA, 1997.

41. R. Fikes, "Strawman DAML+OIL Query Language Proposal." August 22, 2001. http://www.daml.org/listarchive/joint-committee/0572.html

42. J. Z. Pan and I. Horrocks, "Semantic Web Ontology Reasoning in the SHOQ(D_n) Description Logic." In *Proc. 2002 Int. Workshop on Description Logics (DL-2002)*, 2002.

43. B.N. Grosof, "Representing E-Business Rules for the Semantic Web: Situated Courteous Logic Programs in RuleML." In *Proc. Workshop on Information Technologies and Systems (WITS '01)*, New Orleans, Louisiana, USA, 2001.

44. B.N. Grosof, D.W. Levine, and H.Y. Chan, "Flexible Procedural Attachment to Situate Reasoning Systems." U.S. Patent 5,778,150. Granted July 7, 1998; filed July 1, 1996.

11 Appendix

11.1 Reference: SweetDeal Predicates

Special Predicate

`_includes(?CF)`: This incorporates ?CF by reference into the contract.

SweetDeal Mechanism Predicates

`sensable(?Predicate)`: ?Predicate is intended to be bound to a sensor attached procedure.
`effectable(?Predicate)`: ?Predicate is intended to be bound to an effector attached procedure.
`parameter(?CO,?P)`: The contract has a parameter named ?P.
`parameterValue(?CO,?P,?Value)`: The value of parameter ?P in this contract is ?Value.
`riskFor(?RA,?E)`: ?RA is a risk assessment that estimates the risk for the ?E exception class.
`risk(?CO,?E,?Risk)`: The risk of exception ?E on this contract is estimated to be ?Risk.
`templateFor(?T,?Process)`: The template ?T is a partial specification for the process.

Domain Specific Predicates

`buyer(?CO,?Buyer)`: ?Buyer is the buyer for this contract.
`seller(?CO,?Seller)`: ?Seller is the seller for this contract.
`product(?CO,?Product)`: ?Product is the product for this contract.
`price(?CO,?P)`: The price for the product in this contract is ?P dollars per unit.
`quantity(?CO,?Q)`: The contract is for ?Q units of the product.
`shippingDate(?CO,?Date)`: The shipping date specified in the contract ?CO is ?Date.
`payment(?R,?Type,?Amount)`: In the contract result ?R, there is a payment of the specified type and amount from the buyer to the seller.
`country(?Agent,?Country)`: The market agent ?Agent is located in the specified country.
`firmSize(?Agent,?Size)`: The market agent ?Agent has the specified size.
`creditRating(?Agent,?Rating)`: The credit rating of the market agent is ?Rating.

Process Ontology Predicates

`http://xmlcontracting.org/pr.daml#hasException(?P,?E)`:
Exception ?E could occur when process ?P is carried out.
`http://xmlcontracting.org/pr.daml#isHandledBy(?E,?H)`:
Exception ?E is intended to be handled by handler ?H.

Contract Ontology Predicates

`http://xmlcontracting.org/sd.daml#specFor(?CO,?P)`:
Contract ?CO is a specification for process ?P.
`http://xmlcontracting.org/sd.daml#result(?CO,?R)`:
The contract's result is represented by ?R.
`http://xmlcontracting.org/sd.daml#exceptionOccurred(?R,?E)`:
Exception ?E happened during the execution of this contract.
`http://xmlcontracting.org/sd.daml#exceptionLikely(?R,?E)`:
Exception ?E is likely to occur in the current situation (but has not actually occurred).
`http://xmlcontracting.org/sd.daml#detectsException(?H,?EC)`:
Handler ?H detects exception class ?EC.
`http://xmlcontracting.org/sd.daml#anticipatesException(?H,?EC)`:
Handler ?H anticipates exception class ?EC.
`http://xmlcontracting.org/sd.daml#avoidsException(?H,?EC)`:
Handler ?H avoids exception class ?EC.
`http://xmlcontracting.org/sd.daml#resolvesException(?H,?EC)`:
Handler ?H resolves exception class ?EC.

11.2 Built-in Rules

The following are the built-in rules in the SweetDeal system. These basic rules are always included when performing inference on contract provisions.

```
// ***** Calculating payment
// base payment = price * quantity
payment(?CO,base,?Payment) <-
  quantity(?CO,?Q) AND price(?CO,?P) AND
  multiply(?P,?Q,?Payment) ;

// ***** MUTEX properties
// Each contract can only have one price and one quantity.
// For every contract and exception, there can only be one risk estimate.
// For every contract and parameter, there can only be one parameter value.

MUTEX price(?CO, ?X) AND price(?CO, ?Y)
  GIVEN notEquals(?X, ?Y) ;

MUTEX quantity(?CO, ?X) AND quantity(?CO, ?Y)
  GIVEN notEquals(?X, ?Y) ;

MUTEX risk(?CO,?Exception,?X) AND risk(?CO,?Exception,?Y)
  GIVEN notEquals(?X,?Y) ;

MUTEX parameterValue(?CO,?Param,?X) AND parameterValue(?CO,?Param,?Y)
  GIVEN notEquals(?X,?Y);
```