Knowledge sharing in a collaborative business environment

Seema Degwekar* and Stanley Y.W. Su

Database Systems R&D Center,
Department of Computer and Information Science and Engineering,
University of Florida,
Gainesville, Florida, USA
Fax: +1-352-392-1220 E-mail: spd@cise.ufl.edu
E-mail: su@cise.ufl.edu
*Corresponding author

Abstract: In recent years, business organisations have perceived a growing need to collaborate with one another to solve common problems and to stay competitive. An important form of collaboration is sharing of human/ organisational knowledge. In this work, we use different types of business rules and structures of these rules to capture multi-faceted business policies, strategies, regulations, constraints, processes and operating procedures. The occurrence of any event of interest to these organisations can initiate the processing of multiple rules and rule structures. We present a rule specification language and an event-triggered knowledge sharing system for the specification and processing of distributed events, triggers, heterogeneous business rules and rule structures in an enhanced web service infrastructure.

Keywords: business knowledge representation and sharing; rule language; web services; event and rule processing; decision support; electronic business.

Reference to this paper should be made as follows: Degwekar, S. and Su, S.Y.W. (2008) 'Knowledge sharing in a collaborative business environment', *Int. J. Electronic Business*, Vol. 6, No. 1, pp.67–92.

Biographical notes: Seema Degwekar received her PhD in Computer Engineering from the University of Florida in 2007. She obtained her Bachelor's Degree in Computer Engineering from the University of Mumbai in 2000, and her Masters Degree in Computer Science from the University of Florida in 2002. Her research areas include event and rule based systems, knowledge management and sharing, distributed computing, indexing large databases, web services, peer-to-peer systems, XML and web databases.

Stanley Y.W. Su is a Distinguished Professor Emeritus and Adjunct Professor of the Department of Computer and Information Science and Engineering at the University of Florida. He is an IEEE Fellow. He was the Director of the Database Systems Research and Development Center, during 1977–2005 and Distinguished Professor with the Computer and Information Sciences and Engineering Department and the Electrical and Computer Engineering Department since 2001. He received his PhD Degree in Computer Science from the Computer Science Department, University of Wisconsin-Madison in 1968. His research areas are database and knowledge base management, distributed and parallel computing, collaborative systems, web services, semantic modelling, e-government, e-business and e-learning.

1 Introduction

To enable business organisations to compete in the global economy, they must tackle complex problems in areas such as supply chain management, product design, planning and manufacturing, business negotiation, etc. Along with sharing data resources, it would be very useful to also share the *multi-faceted* business knowledge expressed in their policies, constraints, regulations, processes and procedures. Such knowledge sharing will enable collaborating businesses to learn from each other and will better equip them to solve, not only the specific problem they currently work together on, but also similar problems in the future.

Knowledge can be broadly classified as tacit or explicit (Nonaka and Takeuchi, 1995). The explicit multi-faceted knowledge we mentioned above can be specified using any of the following three popular types of *business rules* (Loucopoulos and Katsouli, 1992; Sowa, 2000) used in existing rule-based systems (Business Rules Group, 2000; Rouvellou et al., 2000): integrity constraints, logic-based derivation rules, and action-oriented rules. Integrity constraint rules, used in database systems, ensure the consistency of a database (Ullman, 1982). Logic rules are commonly used in expert systems for decision-support (Ullman, 1988). Action-oriented rules (Widom and Ceri, 1996) are used in event–based systems (Buchmann et al., 2004; Carzaniga et al., 2000) and production rule systems (Riley, 2006; Brownston et al., 1985). These different types of rules can be used to capture business constraints, policies, and regulations.

Business processes and operating procedures encode a significant portion of human/organisational knowledge. A process or procedure can consist of a set of *activities* linked by conditional transitions in a structure that specifies their order and conditions of execution (Hollingsworth, 1995). An activity performs some operations on the input data to produce some output data. Such operations can check the validity of the input data, use the input data to derive some other data, or modify/update the input data. Thus, each type of activity can be represented using the above three types of rules. The relationship between these activities can be captured using a *rule structure* to model an organisational or inter-organisational process or procedure. Expressing organisational policies, regulations, processes etc., as high-level declarative rule and rule structure specifications is more desirable than implementing them in agent or application code because members of the collaborating organisations can easily understand and/or modify them.

We define a *collaboration federation* as a number of autonomous business organisations collaborating with one another to solve some common problems. They are interested in sharing data and knowledge that are pertinent to solving the problems. Organisations would notify each other whenever an *event* of importance occurs and share the data associated with the occurrence of this event (i.e., *event data*). An event is anything of interest and importance like a slip in the production schedule, withdrawal of a product order, or a machine malfunction on the manufacturing floor. Along with these event data, organisations can also share the data generated by the execution of relevant knowledge rules triggered by the event occurrence. In this paper, we shall focus on this *event-triggered* data and knowledge sharing among collaborating business organisations.

On receiving event data, an organisation processes applicable rules and rule structures. A rule or rule structure, whose input data form a subset of the event data, is considered applicable for processing. The data generated by this processing becomes part of the event data. Thus, event data is a dynamic data set that continues to be added or modified by the processing of applicable rules and rule structures. Each new version of

event data is sent to all organisations that have applicable rules and rule structures for another round of processing. This process would continue until no rules and rule structures are applicable to the last version of event data. At this time, all collaborating organisations would have processed all the applicable rules and rule structures and received all the data that are relevant to the event occurrence. In order to achieve the above distributed, event-triggered data sharing and rule processing, we need to have, among other facilities like advanced user interfaces,

- a rule specification language for specifying different types of rules and rule structures
- a mechanism to achieve the interoperation of heterogeneous, distributed rules and rule structures
- an infrastructure for managing and processing distributed events, transmitting and merging event data, and triggering the processing of rules and rule structures in a uniform manner.

They form the focus of this paper.

To achieve the interoperability of heterogeneous business rules, one could have used three types of rule engines to process the three different types of rules and find a way to make the engines themselves interoperable. We believe this will result in a very complex and inefficient system. Another approach taken in Bassiliades et al. (2000) and Rosenberg and Dustdar (2005a) is to choose one knowledge representation (e.g., Event-Condition-Action (ECA) rule) and convert the other two types of rules into the chosen one so that they can all be processed by a single rule engine. One problem with this approach is that the semantics of these rule types are quite different. It is not always possible to convert one type of rule into another without some loss of meaning. Also, most existing rule engines *interpret* rules at runtime resulting in an inefficient, unscalable and centralised rule processing system. In our work, we use a *compilation* approach by translating different types of rules and rule structures at the rule definition time into code and wrapping the code as web services for their uniform registration, discovery and invocation in a web service infrastructure.

The intended contributions of this paper are:

- To introduce an XML-based rule specification language for the specification and exchange of multi-faceted business knowledge. The language has more expressive power than existing rule markup languages because it can be used to specify different types of rules and rule structures.
- To present an approach of processing heterogeneous rules and rule structures in a distributed fashion without the use of different types of rule engines.
- To present an enhanced web service infrastructure and a distributed event-trigger-rule-based system used for processing dynamic event data and multi-faceted knowledge in a collaborative business environment. The rule set introduced by the Business Rules Group (2000) is used to demonstrate the utility of the rule language, processing strategy, infrastructure and system.

We acknowledge some existing systems and approaches that are related to our work in three areas: rule markup languages, event- and rule-based systems, and rule interoperability. Several recent efforts like SRML (Cover, 2001), BRML (Cover, 2002)

and RuleML (The Rule Markup Initiative, 2000) are concerned with developing a rule markup language for business applications. Of these, SRML and BRML address only condition-action and derivation rules, respectively. RuleML is an ongoing effort that aims to include all three types of rules. It has made significant progress in capturing derivation rules. However, support for integrity constraints, ECA rules, and rule structures is still lacking. Besides the ones mentioned above, there are many other efforts such as the Semantic Web Rule Language (SWRL) (2003), which aims to extend the set of axioms of the Web Ontology Language (2004) to include Horn-like (deductive) rules, the Rule Language in OWL (2004), which allows the specification of deductive rules in OWL and provides the facility for translating these rules into Java Expert System Shell (JESS) rules, and the Agent-Object-Relationship Markup Language (AORML) (Wagner, 2003), which represents the behaviour of business entities (business processes, events, agents, claims, etc.) by means of reaction (or ECA) rules. All of the above languages and systems support only one or two of the rule types we are interested in, and offer no support for the notion of rule structure as defined in our language and system. In Wagner et al. (2005), the authors define a rule language by means of MOF/UML models. They present the abstract syntax of the language, but do not provide concrete mappings to a corresponding DTD or XML Schema.

Event- and rule-based systems presented in Buchmann et al. (2004) and Krishnamurthy and Rosenblum (1995) couple event notification with condition-action rules alone. There are also many so-called active database systems, which use only condition-action rules as surveyed in Widom and Ceri (1996). E-DEVICE (Bassiliades et al., 2000) proposes an active knowledge based system to support the processing of all three rule types in an active OODB system by mapping derivation rules and integrity constraints into condition-action rules. However, the system offers no support for integrity constraints. The system presented in Rosenberg and Dustdar (2005a) provides a web service interface to heterogeneous rule engines, thereby providing a uniform API to access each engine. This same system has also been used to demonstrate rule integration in BPEL (Rosenberg and Dustdar, 2005b). Here, the authors focus on the integration of business rules with web service composition. A process workflow is augmented with pre- and post- activity rules that encapsulate business logic. Different from this work, rule execution in our system is governed not only by explicit triggers that link events to rules and rule structures, but also by implicit triggers determined by examining whether the input data to a rule or rule structure are a subset of the event data. Implicit triggers give our system the forward-chaining like behaviour which is not present in the referenced system. Furthermore, rule execution in the referenced system is carried out by individual engines interpretively, whereas we use a compilation approach. Support for rule structures is lacking and it is also not clear how one rule engine can make use of the results generated by another. W3C has established a Rule Interchange Format working group (The Rule Interchange Format, 2005) to produce a framework or language to translate rules between different systems. The purpose of this group is to enable rule interoperability by allowing rules specified in one format to be processed by a different rule engine. At the time of this writing, the RIF Core has been developed, which focuses only on derivation rules. It is the working group's aim to include integrity constraints as well as ECA rules in the language.

The rest of this paper is organised as follows. Section 2 explains the three types of rules and rule structure. Section 3 presents the algorithms for converting rules and rule structures into web services. Section 4 presents the architecture of our implemented event-trigger-rule-based system and the strategy used to decompose and process a structure of different types of rules in a web service infrastructure. Section 5 summarises the paper.

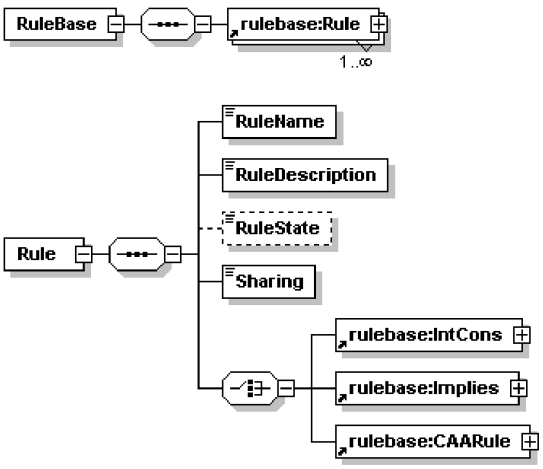
2 Business rule definition language, rule structure and trigger

In this section, we present an XML-based rule language for knowledge specification using the three types of rules and rule structures discussed above. To the best of our knowledge, no single rule language published in existing literature specifies all three types of rules and rule structures. We shall present the rule language first before we present the algorithms for converting rules and rule structures into web services. Rules and rule structures captured in XML are used for two purposes:

- translation to the corresponding web services for rule invocation
- knowledge exchange among collaborating organisations.

Each rule has the following characteristics: a name, a description, an optional state, and a *sharing* attribute that indicates if the rule is local or can be shared with other organisations. The rule state can be either *active* or *suspended*. Figure 1 shows the *Rule* constructs that define a *RuleBase* in the XML format. The syntaxes of the three types of rules are described in the following subsections.

Figure 1 XML syntax for a rule base



2.1 Integrity constraints

An integrity constraint (Ullman, 1982) ensures that changes made to the database do not result in a loss of data consistency. For constraint specification, we adopt some syntactic constructs of our earlier work on a Constraint Specification Language (Su et al., 2001),

which was patterned after the Object Constraint Language (Warmer, 1998). Since we use an object-oriented data model for modelling event data, constraints can be specified on an object or on one or more of its attributes. Constraints can be classified into two types: *attribute constraint* and *inter-attribute constraint*.

An attribute constraint is of the form

$$x\ \theta\ n, \text{ or } x \text{ in } \{n_1, n_2, \ldots, n_a\}$$

where x is an object or object attribute, n is a value from x's domain, θ is one of the six arithmetic comparison operators ($>, \geq, <, \leq, =, \neq$), and $\{n_1, n_2, \ldots, n_a\}$ represents a set of enumerated values from x's domain. Thus an attribute constraint specifies the allowed or valid constant values of an object or object attribute. Examples of attribute constraints are: $a > 10$, b in $\{2, 5, 9\}$, where a and b are object attributes.

An inter-attribute constraint is of the form

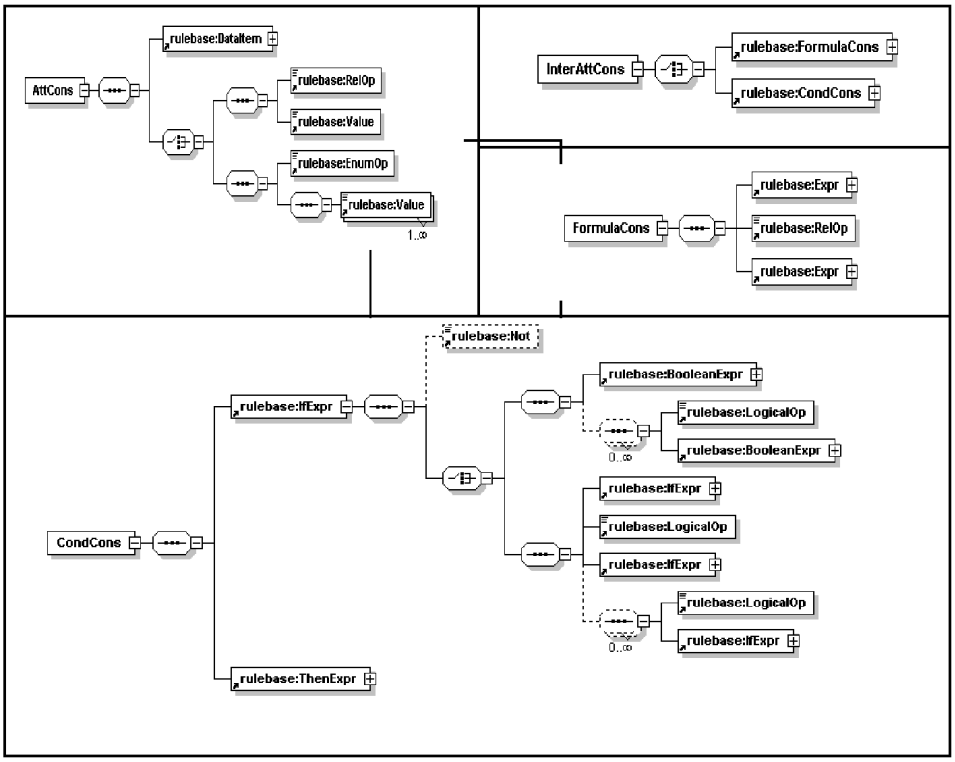
$$f_1(x_1, x_2, \ldots, x_b)\ \theta\ f_2(y_1, y_2, \ldots, y_c), \text{ or } P_1\alpha, P_2\alpha \ldots \alpha P_d: -Q_1\alpha\ Q_2\alpha \ldots \alpha Q_e$$

where $f_1(x_1, x_2, \ldots, x_b)$ and $f_2(y_1, y_2, \ldots, y_c)$ are mathematical formulas relating the objects or object attributes $x_1, x_2, \ldots, x_b$, and $y_1, y_2, \ldots y_c$, respectively. $P_1, P_2, \ldots, P_d$ and $Q_1, Q_2, \ldots, Q_e$ are expressions of the form $f_1(x_1, x_2, \ldots, x_b)\ \theta\ f_2(y_1, y_2, \ldots, y_c)$ connected by the logical operator α in $\{\Lambda, V\}$. The sets $\{P_1, P_2, \ldots, P_d\}$ and $\{Q_1, Q_2, \ldots, Q_e\}$ are linked by an if-then relationship. If $P_1\alpha\ P_2\alpha \ldots \alpha P_d$ holds true, then $Q_1\alpha\ Q_2\alpha \ldots Q_e$ must also hold true. Based on the above definition, we split the inter-attribute constraints into two sub-types: *formula constraints* and *condition constraints*.

The XML syntaxes (in diagram form) for attribute and inter-attribute constraints are shown in Figure 2. The object or object attribute is represented by the element *DataItem*, which is further described with name and type information. Since we generate web services from such constraint specifications programmatically, we require the precise name and type information as needed by the programming language used to implement the web service. Element *RelOp* represents the arithmetic comparison operators, element *EnumOp* represents the enumeration operators *in* or *not in*, and *Value* describes constant values.

The formula constraints are represented by the element *FormulaCons*, whereas the condition constraints are represented by the element *CondCons*. The functions f_1 and f_2 in the formula constraints use the element *Expr*, which is explained below. The expressions in condition constraints are described by the elements *IfExpr*, which represents an *if* construct, and *ThenExpr*, which represents a *then* construct. An *Expr* element is composed of one or more *Term* elements linked by mathematical operators. A *Term* can be a single data item, a constant value, or an operation. An operation can take in zero or more data items as input and produce zero or more data items as output. An *IfExpr* element is composed of one or more Boolean expressions, modeled by *BooleanExpr*, which in turn can be either a predicate expression (an expression that links two *Expr* elements by a comparison operator), or a single term. Due to space limitations, we cannot include the specification of *Expr*, *BooleanExpr*, *Term*, and other elements that complete the rule language specification. Interested readers are referred to Rule Specification Language Schemas (2005) for the complete XML schemas.

Figure 2 XML syntax for attribute and inter-attribute constraints



2.2 Logic-based derivation rule

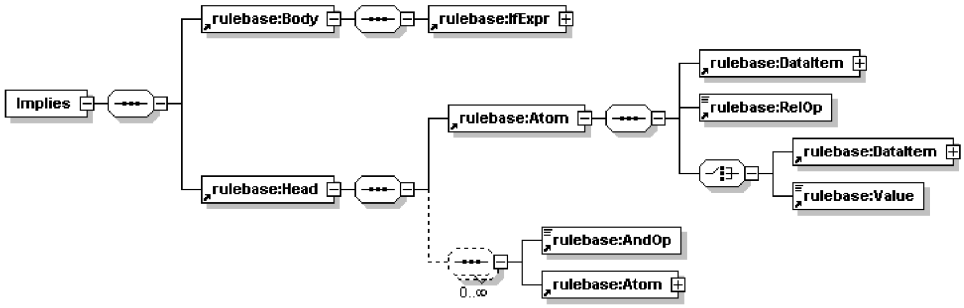
Logic-based derivation rules (Ullman, 1988), also known as inference rules or deductive rules, assess a given premise to come to some conclusion. They can be expressed as

$$P \rightarrow Q, \text{ or } P \Rightarrow Q$$

which means that given the premise(s) P evaluate to *true*, the specified conclusion Q is also determined to be true. Both P and Q can be complex Boolean expressions linked with logical operators Λ and V, with the restriction that the expressions in Q be connected only using the operator Λ. If V semantics are desired, we first obtain the set of expressions $\{e_1, e_2, ..., e_m)$ in Q, where m is the number of predicates in Q connected by V. The original rule $P \rightarrow Q$ is then represented as m rules, $r_1, r_2, \ldots, r_m$, where each r_i has P as the premise and e_i as the conclusion.

The derivation rule syntax is shown in Figure 3. It is derived in part from RuleML (The Rule Markup Initiative, 2000). The implication is represented by the element *Implies*, which consists of the *Head* and *Body* elements. The *Head* element contains one or more *Atom* elements linked by the Λ operator. Each of these atoms specifies a new or derived value for the indicated object or object attribute. The *Body* element consists of one or more *IfExpr* elements linked by the logical operators Λ and V.

Figure 3 XML syntax for a derivation rule



2.3 Action-oriented rule

Action-oriented rules are typically found in active database systems (Widom and Ceri, 1996). They are known as ECA rules or just event-action rules (Krishnamurthy and Rosenblum, 1995). When an event specified by the event clause occurs, the ECA rule checks for the truth value of the condition clause, and executes the action clause if the condition clause is true.

The general format of the rule is thus

On E If C then execute A.

If we separate the event (E) from the Condition-Action (CA) part, we see that the CA part is in fact an action-oriented rule. The event is used to indicate when to perform the action-oriented rule. Also, it may be useful to specify the actions to be taken when the condition expression evaluates to false. We can model this concept as two complementary if-then rules

C_1A, and C_2B

where $C_1 = C_2$, and A and B are the respective action clauses. Another way would be to represent both the if-then rules using a single if-then-else rule. In our earlier work, we have used this format to define what we call a *condition-action-alternative_action* (CAA) (Lee et al., 2001) rule. Thus, the format of the CAA rule is

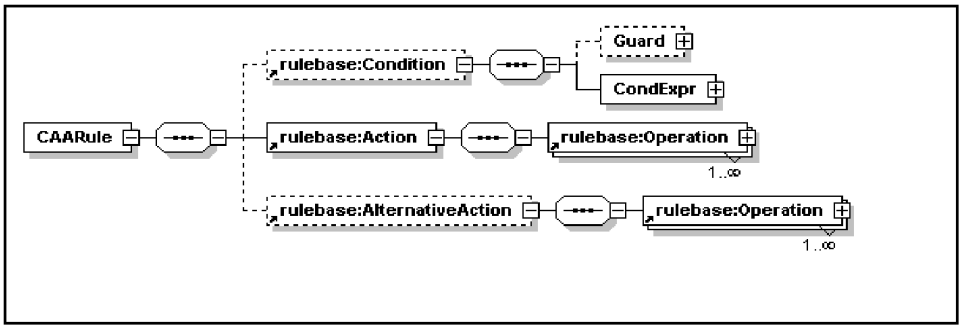
If C then A else B

where C is the condition expression, and A, B are the action and alternative action clauses, respectively. We use the above form of the action-oriented rule in this paper.

We present the syntax of the CAA rule in Figure 4. The condition clause is represented using the *Condition* element, the action clause by the *Action* element and the alternative-action clause by the *AlternativeAction* element, respectively.

The condition expression is composed of two parts – a *guard* clause and a *condition* clause. The guard clause is optional. It can have an ordered list of expressions to be evaluated in turn. If any expression is false, the entire rule is skipped, thereby serving as a means of conditionally deactivating the rule. If both the guard clause and the condition clause are true, then the action is taken. If the guard clause is true and the condition clause is false, the alternative action is taken. The *Action* and *AlternativeAction* elements consist of a set of operations.

Figure 4 XML syntax for a CAA rule



Since we allow events and rules to be defined and published by different organisations, we separate event (E) specifications from condition-action-and-alternative_action (CAA) specifications. We allow organisations to use a trigger (to be discussed in Section 2.5) to link an event to, not only a CAA rule, but also an integrity constraint or a derivation rule. A trigger can also link an event to a structure of rules of different types. Thus, an event occurrence can invoke an integrity constraint check, a derivation of new data, an execution of a CAA rule, or a structure of these rules.

2.4 Rule structure

When a specific event occurs, an organisation may have different types of rules that need to be executed in a specific order to carry out a procedure or process. It is very natural to model such a procedure or process by specifying the structural relationships between individual rules. We capture these relationships in a rule structure introduced in our earlier work (Lee et al., 2001).

In an organisational process, a rule r may be required to execute before another rule s, thus establishing a direct *link* between r and s. Similarly, a rule r may be required to execute before multiple rules $s_1, s_2, \ldots, s_m$, $m > 1$, which can then be processed in parallel. In this case, r is connected to $s_1, s_2, \ldots, s_m$ in a *split* construct. A rule s may be required to wait for all of a given set of rules $r_1, r_2, \ldots, r_n$, $n > 1$ to finish before it can start its own execution.

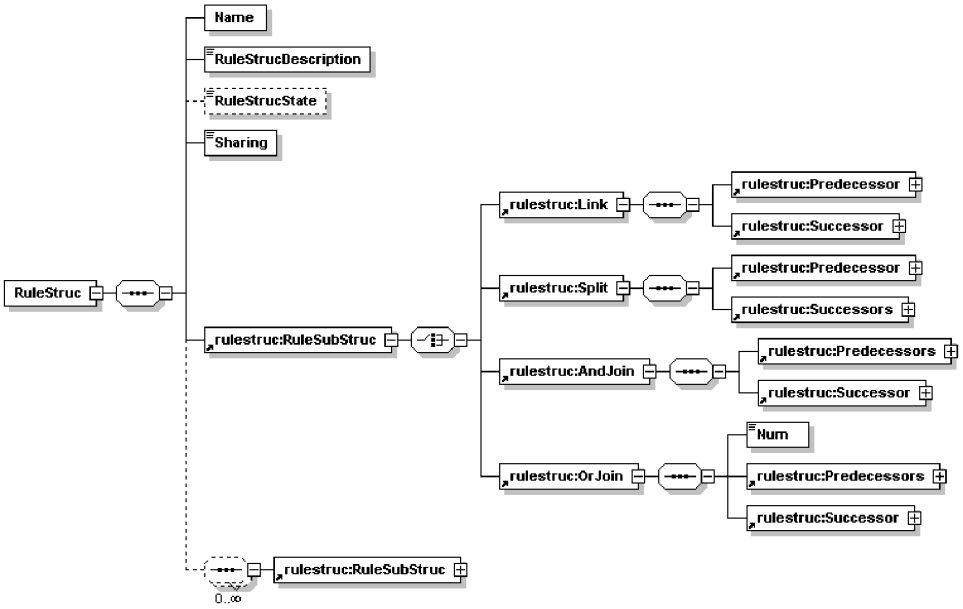
In this case, $r_1, r_2, \ldots, r_n$ are connected to s in an *and-join* construct. Also, s may be required to wait for not all but a subset of the rules $r_1, r_2, \ldots, r_n$, $n > 1$ to finish execution. This establishes an *or-join* relationship between $r_1, r_2, \ldots, r_n$ and s. In each type of relationship, the rule(s) that governs(govern) the execution of other rules is(are) termed predecessor(s), and the rule(s) that executes(execute) after the predecessor(s) is(are) termed successor(s).

> *A rule structure can now be defined as a directed graph with different types of rules as nodes, which are connected by link, split, and-join, and or-join constructs.*

The syntax of a rule structure is given in Figure 5. A rule structure is represented using the *RuleStruc* element at the root. A rule structure is composed of one or more substructures (*RuleSubStruc*), each of which represents one of the relationships discussed above. The predecessors and successors in each rule path are expressed using the elements *Predecessor* (rule r in a link or split), *Predecessors* (rules $r_1, r_2, \ldots, r_n$ in a join),

Successor (rule *s* in a link or join), or *Successors* (rules s_1, s_2, …, s_m in a split). *Num* represents the number of rules *s* may be required to wait for in an *or-join* relationship.

Figure 5 XML syntax for a rule structure



2.5 Trigger

A trigger specifies a number of alternative events that will trigger the processing of a single rule or a structure of rules. Just like events and rules, triggers can be explicitly defined by collaborating organisations that are different from those that defined events, rules and rule structures. In this case, the organisation that contains the rule or rule structure becomes an implicit subscriber of the event. Triggers can also be automatically generated by the system to link a distributed event to a distributed rule or rule structure, if the event data (or part of it) can provide the input data needed for processing the rule or rule structure. In this case, the rule or rule structure is said to be applicable to the event.

We stress the importance of allowing collaborating organisations to independently define events, rules, rule structures and triggers in a distributed, collaborative environment because it gives the flexibility and power to express and enforce policies, regulations, constraints, procedures and processes that are important to achieve inter-organisational sharing and collaboration.

3 Heterogeneous rules as web services

3.1 Creating a web service

Figure 6 outlines the general algorithm for web service synthesis. It takes as input a specification of rules in XML that conforms to the rule definition language introduced in Section 2. Each rule is translated to code and wrapped as a web service.

The algorithm invokes appropriate handler methods to create the source code for the rule (steps 3–5) depending on its type. Depending on the application framework, we need to create specific configuration files required for successful compilation and deployment of a web service (steps 6–8). To facilitate discovery of a web service using UDDI (2001), we also publish the web service to a private UDDI registry (step 9).

Figure 6 Algorithm for web service synthesis

Algorithm 1 createWebService
1. *rules* = getRules(*ruleBase*);
2. **for** each rule *r* **do**
3. **if** (*r* is an integrity constraint) icHandler(*r*); **end if**
4. **else if** (*r* is a derivation rule) drHandler(*r*); **end if**
5. **else if** (*r* is a CAA rule) caaHandler(*r*); **end if**
6. createConfigFiles(*r*);
7. compile(*r*);
8. deploy(*r*);
9. publish(*r*);
10. **end for**

3.2 Mechanisms for converting different types of rules

3.2.1 Integrity constraints

Each integrity constraint rule is represented as a web service with the operation/method *check(...)*. The algorithm determines the type of the constraint and generates the appropriate program statements for *check(...)*, details of which are given in Figure 7. The method *check(...)* examines the supplied input data and returns a Boolean value which is *true* if the specified constraint is satisfied and *false* otherwise. Specific cases to handle comparison operators, enumeration operators, formula constraints and condition constraints are shown in the algorithm. All data items referenced in the constraint rule constitute the input to *check(...)*.

If the constraint type is an attribute constraint, the rule code checks if the relationship specified by the comparison or the enumeration operator holds. If the constraint is an inter-attribute constraint of the formula constraint subtype, the algorithm first generates code for the expressions on the left hand side and the right hand side as *lExprCode* and *rExprCode*, respectively. The method *check(...)* then determines if the expression represented by *lExprCode* is related to the one represented by *rExprCode* as specified by *op*, where *op* is the comparison operator used. For a condition constraint, the algorithm generates code for the if-part and the then-part as *ifPartCode* and *thenPartCode*, respectively. The method *check(...)* then determines if *ifPartCode* is true. If so, it determines if *thenPartCode* also holds true. Each of these code statements for the integrity constraint web service is captured in the list *stmts*.

After all the constraints have been examined, a service interface file is created. The name and type information from the parameter name and type lists is used to create the operation *check(...)*. Next, a service implementation file is created. Each code statement from the list *stmts* is written to the file.

Figure 7 Algorithm for an integrity constraint

Algorithm 2 icHandler
(1) *paramNames* = null, *paramTypes* = null; *stmts* = null;
(2) **if** constraint *c* is an attribute constraint **do**
(3) *d* = *c* → DataItem; add *d* → Name and *d* → Type to
(4) *paramNames* and *paramTypes* respectively
(5) *op* = *c* → getNextChild();
(6) **if** *op* is a comparison operator
(7) add "if d → Name op c → Value return true;" to *stmts*
(8) **end if**
(9) **if** *op* is the enumeration operator *in*
(10) *values* = getValues(c);
(11) add "if d → Name in values return true;" to *stmts*
(12) **end if**
(13) **if** *op* is the enumeration operator *not in*
(14) values = getValues(c);
(15) add "if ⌐(d → Name) in values return true;" to *stmts*
(16) **end if**
(17) **else if** constraint *c* is an inter-attribute constraint **do**
(18) *dataItemList* = getInputData(*c*);
(19) **for** each DataItem *d* in *dataItemList* **do**
(20) add *d* → Name and *d* → Type to *paramNames* and
(21) *paramTypes* respectively
(21) **end for**
(22) **if** *c* is a formula constraint **do**
(23) *lExprCode* = getExprCode(*c* → Expr);
(24) *op* = *c* → RelOp;
(25) *rExprCode* = getExprCode(*c* → Expr);
(26) add "if (lExprCode op rExprCode) return true;" to
(27) *stmts*
(28) **end if**
(28) **else if** *c* is a condition constraint **do**
(29) *ifPartCode* = getIfExprCode(c → IfPart);
(30) *thenPartCode* = getThenExprCode(*c* → ThenPart);
(31) add "if (ifPartCode = true and thenPartCode = true)
(32) return true;" to *stmts*
(32) **end else**
(33) **end else**
(34) createServiceInterfaceFile(*paramNames*, *paramTypes*);
(35) createServiceImplementationFile(*paramNames*,
(36) *paramTypes*, *stmts*);

Function: createServiceImplementationFile(paramNames, paramTypes, stmts)
(1) **for** each stmt *s* in stmts **do**
(2) write *s* to file
(3) **end for**
(4) write "return true;" to file

3.2.2 Derivation rules

Each derivation rule is translated to a web service with the operation/method *implies(...)*. This method examines the input data to determine if the body of the implication is true. If so, it returns the new data specified in the head of the implication. Figure 8 shows the algorithm for creating the method.

Figure 8 Algorithm for derivation rules

Algorithm 3 drHandler
(1) *paramNames* = null, paramTypes = null;
(2) *body* = *implies* → Body;
(3) *head* = *implies* → Head;
(4) *dataItemList* = getInputData(*body*, *head*);
(5) **for** each DataItem *d* in *dataItemList* **do**
(6) add *d* → Name and *d* → Type to *paramNames* and
(7) *paramTypes* respectively.
(8) **end for**
(9) *bodyVal* = getBodyCode(*body*);
(10)*headVal* = getHeadCode(*head*);
(11)createServiceInterfaceFile(*paramNames*, *paramTypes*);
(12)createServiceImplementationFile(*paramNames*, *paramTypes*, *bodyVal*, *headVal*);

Function: getHeadCode(head)
(1) *headVal* = null;
(2) *atomList* = getAtoms(*head*);
(3) **for** each Atom *a* in *atomList* **do**
(4) *dataItem* = *a* → DataItem;
(5) *value* = *a* → getNextChild();
(6) add the pair (*dataItem*, *value*) to *headVal*
(7) **end for**
(8) return *headVal*;

Function: createServiceImplementationFile(paramNames, paramTypes, bodyVal, headVal)
(1) write "map = null;" to file
(2) write "if (bodyVal) then"
(3) **for** each pair (*dataItem*, *value*) in *headVal* **do**
(4) write "map.put(dataItem, value);" to file
(5) **end for**
(6) write "end-if" to file
(7) write "return map;" to file

The input parameters to the method *implies(...)* are the data items required by the body of the implication and the data items referenced in the 'value' part of the head. The body contains one or more Boolean expressions linked by logical operators *and* and *or*. The *getBodyVal(...)* function translates the implication body into program statements. Each Boolean expression in the implication body makes use of the function *getExprCode(...)* from Figure 7 to generate the code for the expression. The head of the implication contains one or more *Atom* elements linked by the *and* operator. Each of these elements contains a derived data item and the corresponding value. This value can be a constant value or another data item. The function *getHeadMap(...)* constructs a hashmap named *headVal*, which contains the new data as a *(name, value)* pair indexed by *name*. If the body evaluates to true, the output contains each pair in *headVal*, otherwise it is null.

A service interface file is created using information from *paramNames* and *paramTypes*. A service implementation file is basically an *if-then* block. The expression in the *if* statement is the implication body and the expression in the *then* statement is the implication head. The statements in the *then* block transfer each *(data item, value)* pair

from *headVal* into *map*. At runtime, if the body is true, *map* will contain the new data values, otherwise it will be empty.

3.2.3 CAA rules

Each CAA rule is translated to a web service with the operation/method *perform(...)*. If the guard clause evaluates to true, this method examines the input data to see if the condition is satisfied. If so, the operations specified in the action clause are executed, and the output of those operations is returned, otherwise the operations specified in the alternative_action clause are executed and the output of those operations is returned. Figure 9 gives the algorithm for creating the method *perform(...)*.

Figure 9 Algorithm for CAA rules

Algorithm 4 caaHandler
(1) *paramNames* = null, *paramTypes* = null, *actionMap* = null, *altActionMap* = null;
(2) *actionData* = null, *altActionData* = null; *condition* = caarule → Condition;
(3) *condExpr* = condition → CondExpr; *action* = caarule → Action;
(4) *altAction* = caarule → AlternativeAction;
(5) *dataItemList* = getInputData(*condition*, *action*, *altAction*);
(6) **for** each DataItem *d* in *dataItemList* **do**
(7) add *d* → Name and *d* → Type to *paramNames* and *paramTypes* respectively
(8) **end for**
(9) *dataItemList* = getInputData(*action*, *altAction*)
(10)**for** each DataItem *d* in *dataItemList* **do**
(11) add *d* → Name and *d* → Type to *paramNames* and *paramTypes* respectively.
(12)**end for**
(13)*dataItemList* = getOutputData(*action*);
(14)**for** each DataItem *d* in *dataItemList* **do**
(15) add *d* → Name and *d* → Type to *actionData*
(16)**end for**
(17)*dataItemList* = getOutputData(*altAction*);
(18)**for** each DataItem *d* in *dataItemList* **do**
(19) add *d* → Name and *d* → Type to altActionData
(20)**end for**
(21)*conditionVal* = getCondCode(*condition*); *actionVal* = getActionCode(*action*);
(23)*altActionVal* = getAltActionCode(*altAction*);
(24)createServiceInterfaceFile(*paramNames*, *paramTypes*);
(25)createServiceImplementationFile(*paramNames*, *paramTypes*, *conditionVal*,
(26)*actionVal*, *altActionVal*, *actionData*, *altActionData*);

Function: getActionCode(action)
(1) *actionVal* = null;
(2) *operationList* = getOperations(*action*);
(3) **for** each operation *o* in *operationList* **do**
(4) *name* = *o* → OperationName;
(5) *dataItemList* = getInputData(*o*);
(6) *return* = *o* → Return;
(7) create the parameter list from *dataItemList*
(8) **if** *return* == null
(9) add "name(parameter list);" to *actionVal*
(10) **end if**
(11) **else**
(12) *returnData* = all data items in return → DataItem
(13) **if** *returnData* contains only one data item
(14) add "returnData = name(parameter list);" to *actionVal*
(15) **end if**
(16) **else**
(17) *outputClass* = createOutputClass(name+ "Output");
(18) **for** each DataItem *d* in *returnData* **do**
(19) include d as a member of the output class;
(20) **end for**
(21) add "outputDataInstance = instanceOf(outputClass);" to *actionVal*

Figure 9 Algorithm for CAA rules (continued)

```
(22)    add "outputDataInstance = name(parameter list);" to actionVal
(23)   end else
(24)  end else
(25)end for
(26)return actionVal;

Function: createServiceImplementationFile paramNames, paramTypes, conditionVal,
actionVal, altActionVal, actionData, altActionData);

(1)  write "map = null;" to file
(2)  write "if ("+conditionVal+") then"
(3)  write actionVal to file
(4)  for each pair (dataItem, value) in actionData do
(5)   write "map.put(dataItem, value);" to file
(6)  end for
(7)  write "else" to file
(8)  write altActionVal to file
(9)  for each pair (dataItem, value) in altActionData do
(10)  write "map.put(dataItem, value);" to file
(11)end for
(12)write "return map" to file
```

The input parameters to the rule are the data items referenced in the condition clause, and the data items specified as input to the operations in the action and alternative_action clauses. The output data from the operations in the action clause are added to the hashmap *actionData*, and the output data from the operations in the alternative_action clause are added to the hashmap *altActionData*. Both these maps contain *(name, value)* pairs, indexed by *name*. The function *getConditionCode(...)* translates the condition clause to program statements. Similarly, the functions *getActionCode(...)* and *getAltActionCode(...)* translate the action and alternative_action clauses to code, respectively. The output of the method *perform(...)* is a hashmap named *map*, which contains the result of the operations performed by the rule. At runtime, either the action or the alternative_action clause is executed, and *map* contains the corresponding output.

The function *getActionCode(...)* is shown in some detail in Figure 9. *getAltActionCode(...)* is very similar and is not shown separately. An action (or alternative_action) clause consists of a list of operations to be executed. The function translates each operation to program statements of the form *output = operation name (input)*, or *operation name (input)* depending upon whether the operation contains return parameters or not. These calls are of course, web service calls. In case the operation returns multiple data items, we create an output class that captures all of these data items as the class' data members. All such program statements are stored in the array *actionVal* (or *altActionVal*), which is returned. Service interface and implementation files are constructed in a manner similar to that for derivation rules.

3.3 Mechanism for a rule structure

A rule structure links rules together in a directed acyclic graph as explained in Section 2. It is defined and published as a web service just like any other rule. We require the following condition to be met to generate the web service for a rule structure.

> *Each of the rules in a rule structure must have already been defined and published as a web service before the rule structure is defined.*

Each rule structure is represented as a web service with the operation/method *execute(...)*. Figure 10 shows the algorithm for creating *execute(...)*. This method takes in all the input data items required by all the rules in the structure, and returns the output data items produced by all these rules. The objective of the method is to invoke the rules in the order specified by the structure. To facilitate this, it creates *invoker threads* for each rule in the structure. The algorithm to create such an invoker thread (*createInvokerThread(...)*) is also shown in the figure.

Figure 10 Algorithm for rule structure

Algorithm 5 rsHandler

(1) *ruleStrucList* = getRuleSubStruc(*ruleStruc*);
(2) *ssVal* = null;
(3) *distinctRules* = get distinct rules referred in *ruleStruc*
(4) **for** each rule *r* in *distinctRules* **do**
(5) createCallerRoutine(*r*)
(6) **end for**
(7) **for** each RuleSubStruc *rss* in *ruleStrucList* **do**
(8) **if** *rss* is a Link *ssVal* += linkHandler(*rss* → Link); **end if**
(9) **if** *rss* is a Split *ssVal* += splitHandler(*rss* → Split); **end if**
(10) **if** *rss* is an ANDJoin *ssVal* += andJoinHandler(*rss*→ ANDJoin); **end if**
(11) **if** *rss* is an ORJoin *ssVal* += orJoinHandler(*rss* → ORJoin); **end if**
(12)**end for**
(13)createServiceInterfaceFile();
(14)createServiceImplementationFile(*ssVal*);

Function: createInvokerThread (Rule r)
(1) write "call"+*r.name*+" extends Thread", thus creating a caller class for the rule
(2) *dataItemList* = getInputAndOutput(*r*);
(3) **for** each DataItem *d* in *dataItemList* **do**
(4) declare a corresponding public data member with same name and type
(5) **end for**
(6) write "public void run()" to file
(7) prepare for rule web service call and write it out to the file
(8) write code to call the rule web service to the file
(9) **for** each DataItem *d* generated by the web service **do**
(10) copy *d*'s the value to the corresponding data member of the class
(11)**end for**

Function: splitHandler(Split s)

(1) *pred* = *s* → Predecessor;
(2) *succList* = *s* → Successors;
(3) *splitVal* += "*callPred* = invoker thread to call *pred*;"
(4) *splitVal* += "**for** each Rule *succ* in *succList* **do**";
(5) *splitVal* += "*callSucc* = invoker thread to call *succ*;";
(6) *splitVal* += "**end for**";
(7) *splitVal* = "**if** *callPred.state* == NEW and *callPred* not in *toBeExecuted*"
(8) *splitVal* += "add *callPred* to *toBeExecuted*;";
(9) *splitVal* += "Start *callPred*;";
(10)*splitVal* += "**end if**";
(11)*splitVal* += "**if** *callPred.state* == TERMINATED and *callPred* in *toBeExecuted*"
(12)*splitVal* += "remove *callPred* from *toBeExecuted*;";
(13)*splitVal* += "copy output of rule to member variables;"
(14)*splitVal* += "**end if**";
(15)*splitVal* += "**for** each Rule *succ* in *succList* **do**";
(16)*splitVal* += "**if** *callSucc.state* == NEW and *callSucc* not in *toBeExecuted*";
(17)*splitVal* += " add *callSucc* to *toBeExecuted*;"
(18)*splitVal* += " **end if**";
(19)*splitVal* += "**if** *callPred.state* == TERMINATED and *callSucc.state* == NEW";
(20)*splitVal* += " Start *succ*;";

Figure 10 Algorithm for rule structure (continued)

```
(19)splitVal += "if callPred.state == TERMINATED and callSucc.state == NEW";
(20)splitVal += " Start succ;";
(21)splitVal += " end if";
(22)splitVal += "if callSucc.state == TERMINATED and callSucc in toBeExecuted"
(23)splitVal += "remove callSucc from toBeExecuted;";
(24)splitVal += "copy output of rule to member variables;"
(25)splitVal += "end if";
(26)splitVal += " end for";
(27)return splitVal;

Function: orJoinHandler(ORJoin o)

(1) predList = o → Predecessors;
(2) succ = o → Successor;
(3) orJoinVal = "for each Rule pred in predList do"
(4) orJoinVal += "callPred = invoker thread to call pred;"
(5) orJoinVal = "if callPred.state == NEW and callPred not in toBeExecuted"
(6) orJoinVal += "add callPred to toBeExecuted;";
(7) orJoinVal += "Start callPred;";
(8) orJoinVal += "end if";
(9) orJoinVal += "if callPred.state == TERMINATED and callPred in toBeExecuted"
(10)orJoinVal += "remove callPred from toBeExecuted;";
(11)orJoinVal += "copy output of rule to member variables;"
(12)orJoinVal += "end if";
(13)orJoinVal += "end for";
(14)orJoinVal += "num = o → Num; count = 0;"
(15)orJoinVal = "for each Rule pred in predList do"
(16)orJoinVal += "if callPred.state == TERMINATED"
(17)orJoinVal += "count++;";
(18)orJoinVal += "end if"
(19)orJoinVal += "end for"
(20)orJoinVal += "callSucc = invoker thread for succ;"
(21)orJoinVal += "if callSucc.state == NEW and callSucc not in toBeExecuted"
(22)orJoinVal += "add callSucc to toBeExecuted;"
(23)orJoinVal += "end if"
(24)orJoinVal += "if callSucc.state == NEW and count >= num"
(25)orJoinVal += "Start callSucc;"
(26)orJoinVal += "end if"
(27)orJoinVal += "if callSucc.state == TERMINATED and callSucc in toBeExecuted"
(28)orJoinVal += "remove callSucc from toBeExecuted;";
(29)orJoinVal += "copy output of rule to member variables;"
(30)orJoinVal += "end if";
(31)return orJoinVal;

Function: createServiceImplementation(ssVal)

(1) write "toBeExecuted == null" to file;
(2) write "do" to file
(3) for each path execution p in ssVal
(4) write p to file
(5) end for
(6) write "while toBeExecuted is not empty" to file
```

The algorithm *rsHandler(...)* breaks down the given rule structure into substructures. Each substructure contains a *link*, a *split*, an *and-join*, or an *or-join* relationship between two or more rules. The algorithm generates invoker threads for every distinct rule in the substructure. Depending on the substructure type, one of the four *handler* functions is

called. Each of these handler functions is responsible for generating code that will cause the rule execution in the manner specified by the corresponding substructure type. The method *execute(...)* is then constructed by putting together code generated by each of the handler routines. The method *execute(...)* maintains an array *toBeExecuted* to keep track of those rules that have yet to begin execution.

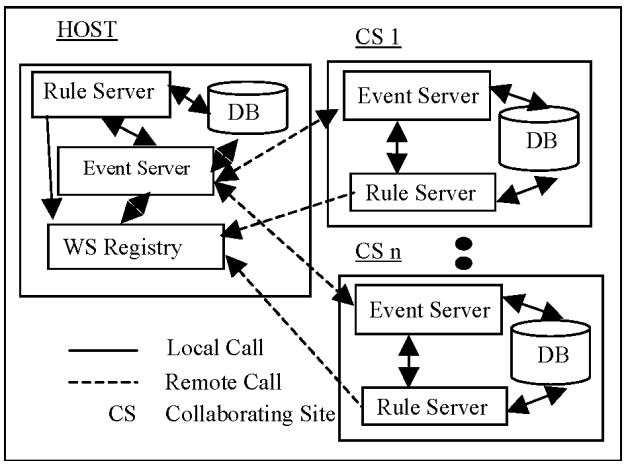
linkHandler(...) generates program statements to ensure that a successor rule r_2 is executed only after the predecessor rule r_1 has finished execution as follows. First, the code creates an invoker thread object for r_1. If this is a newly created thread, r_1 is included in *toBeExecuted*, if not already put in, and the thread for r_1 is started. Once this rule has finished execution, it copies the output items from r_1 to the rule structure's member variables, and removes r_1 from *toBeExecuted*. It creates an invoker thread for r_2, and puts it in *toBeExecuted* if this is a newly created thread. Once r_1 finishes execution, r_2 is allowed to proceed. Once r_2 has finished execution, it is removed from *toBeExecuted*, and its output copied to member variables. Similarly, *splitHandler(...)* generates code to invoke the successor rules only after the predecessor rule has been executed, *andJoinHandler(...)* generates code to execute the successor rule only after all the predecessor rules have been executed, and *orJoinHandler(...)* generates code to execute the successor rule only after the specified number of predecessor rules have been executed. The algorithms for *splitHandler* and *orJoinHandler* are included in Figure 10.

4 System architecture and rule processing

4.1 System architecture

The distributed event- and rule-based system has a peer-to-peer server architecture shown in Figure 11. All organisations have identical subsystems installed at their sites. Each collaborating site creates and manages its own events, rules, rules structures and triggers, but their specifications are registered/published at the host site of a collaboration federation. The host maintains a repository of these specifications.

Figure 11 System architecture



The *rule server* component at each site stores and manages the web services generated for the rules and rule structures defined at that site. These web services are registered at the web service registry (*WSRegistry*) of the host site. The *event server* component is responsible for storing information about events defined at that particular site and the information about event subscribers. A collaborating site can specify a trigger linking a distributed event to a rule or rule structure, thus becoming an *explicit subscriber* of that event. This information is stored by the event server in a local database. Triggers can be automatically and dynamically generated by the system if the event data schema associated with an event occurrence is a superset of the input data schema of a rule or rule structure. In this case, the site that has the rule or rule structure becomes an *implicit subscriber* of the event. Both explicit and implicit subscribers will be notified upon the occurrence of an event. Distributed rules and rule structures are invoked and processed by replicas of the rule server. An event server at any site can serve as the coordinator for a particular knowledge sharing session initiated by an event occurrence at that site. It handles the aggregation of the dynamic event data sets associated with the event occurrence.

We have implemented the algorithms for converting knowledge rules to web services in Java, with the Sun Java System Application Server Platform Edition 9 as our application server. The event and rule servers have been implemented using the Enterprise JavaBeans 2.1 framework. To facilitate easy and efficient lookup, we publish the deployed web service to our private UDDI registry using the UDDI4J API. This registry makes use of the Apache jUDDI project to communicate with a MySQL database that stores the web service information. The MySQL database is also used by the event server to store the event and trigger information. We use a private registry instead of a publicly available UDDI-based registry for two reasons. By eliminating clutter typically found in a business registry, we speed-up registry look up. Also, a private registry provides some level of security, as it is available only to the organisations participating in a collaboration federation.

4.2 Event-triggered processing of rules

We now use an example scenario to describe the event-triggered processing of distributed rules and rule structures, and explain how a rule structure is decomposed and processed.

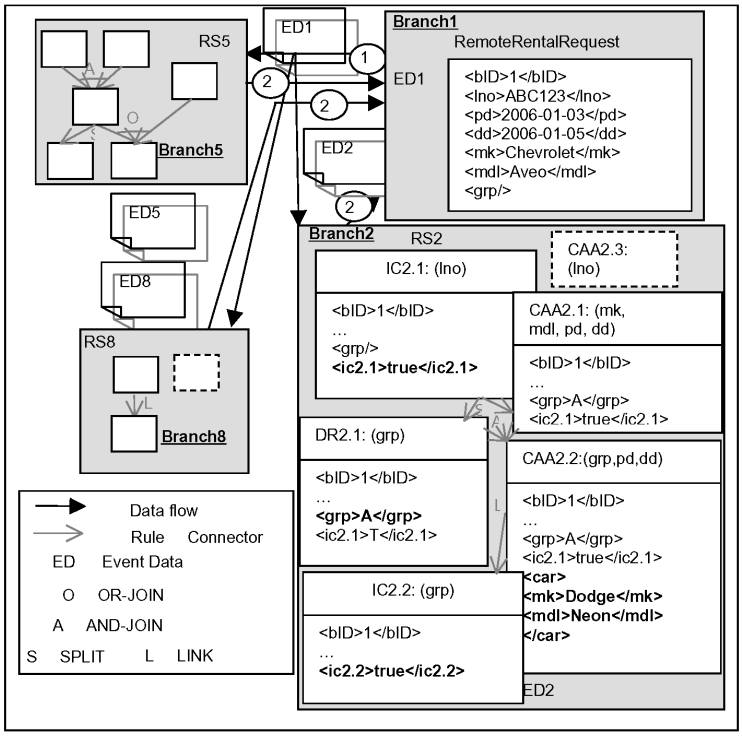
4.2.1 Distributed rule and rule structure processing

In their first white paper (Business Rules Group, 2000), the Business Rules Group has provided a set of business rules for the operation of a fictitious car rental company named *EU-Rent*, which has 1000 branches in towns in different countries. Each branch may wish to share some data and knowledge specifications with the other branches to achieve better management of the company as a whole. Thus, each branch is a collaborating site in the collaboration federation of the EU-Rent company. Although this is not a real-world inter-organisational setting, it suffices to serve our purpose due to the following reasons. First, the rule set has been independently constructed and published for academic use by a well-known group. Second, each of the rules in the rule set belongs to one of the three different rule types processed by our system. Third, as can be seen from the rule set, managing the activities of multiple branches requires the interoperation of different rule types captured in rule structures.

We have used our rule specification language to define EU-Rent's rules and rule structures, and convert and register them as web services. There are a total of 46 rules, of which 29 are CAA rules, 13 are integrity constraint rules and 4 are derivation rules. There are 9 rule structures discernible from the rule set. Each of these rules and rule structures took about 6–7 s to be converted, compiled, deployed and published as a web service on a Windows XP machine with an Intel Pentium 4 processor and 1 GB RAM. The major component of the total time required for web service creation is in the deployment, and publication activities. On average, the compilation takes a few ms for a rule or rule structure, whereas the deployment takes about 5–6 s, and the publication an additional 0.5–1 s.

To demonstrate the event-triggered processing of distributed rules and rule structures, we use the scenario depicted in Figure 12. A customer approaches a local branch *Branch1* with the request for a car rental. *Branch1* is unable to satisfy his/her request. It posts an event *RemoteRentalRequest* and the event data in XML format (*ED1*) is sent to all subscribing branches (denoted as step 1 in the figure). The event data contains the branch identifier (*bID*), the customer license number (*lno*), the pickup date (*pd*) and dropoff date (*dd*), the make (*mk*), the model (*mdl*) and the group or class (*grp*) of the car requested. Assume that *Branch2*, *Branch5* and *Branch8* are branches that have applicable rules. Each branch has a local set of rules that need to be processed when the *RemoteRentalRequest* event occurs. Let us consider in some detail the rules to be executed at *Branch2*, which has defined a trigger linking the event to the rule structure *RS2*, but no trigger that links the event to the applicable rule *CAA2.3*.

Figure 12 Event-triggered processing of distributed rules and rule structures in a collaboration federation



CAA2.3 is a rule that determines if a customer is eligible for a loyalty incentive scheme at the branch. If the customer has made four or more rental reservations at that branch, he/she is eligible and can receive free rentals or upgrades. This rule is invoked (shown by a dotted line in the figure) by the rule server because the rule input data, the customer's license number (*lno*), is a subset of the event data and a trigger is automatically generated by the system to link the event to the rule. In this scenario, we assume the customer is not eligible, and hence *CAA2.3* generates no output. The capability to invoke applicable rules even though no triggers are explicitly specified by users is a very important feature of the system because some collaborating organisation may publish only shareable rules and rule structures without specifying any trigger.

Now let us consider the rule structure *RS2*. A customer may be blacklisted due to earlier problems with payments and returns. The branch must not serve such a customer, so the first rule in *RS2* checks that the customer is not blacklisted by means of the integrity constraint rule *IC2.1*. This constraint rule checks if the customer license number is not in the list of blacklisted customers. The input to this rule is the customer's license number (*lno*), and the output is a data item of type Boolean, with the same name as the constraint's name having a true/false value to indicate whether the constraint was satisfied or not. The result of this check is written to the event data file as the new data item *ic2.1* (shown in bold face). Next, the derivation rule *DR2.1* and condition-action-alternative_action rule *CAA2.1* are executed. If no group is specified for the car requested, *DR2.1* assigns the default car group *A* to the request. The input to this rule is the requested car group (*grp*), and the output is the default car group A, if the input car group is null. In this scenario, the event data file did not mention any car group, so DR2.1 supplies the default car group A. If the make and the model have been specified for the car request, *CAA2.1* checks if *Branch2* has an available car with the same make and model. The input to this rule is the make (*mk*), model (*mdl*), pickup date (*pd*), and drop-off date (*dd*). The output contains the details of an available car. In this scenario, we assume that *CAA2.1* cannot find an available car. Once both these rules have been executed, rule *CAA2.2* is invoked. If *CAA2.1* cannot find an available car, *CAA2.2* checks if there is an available car in the specified group, which could be the group in the event data or the default group assigned by *DR2.1*. The input is thus the car group (*grp*), the pick-up date (*pd*), and the drop-off date (*dd*). The output items of *CAA2.2* are the make and model details of an available car. After *CAA2.2* has finished execution, the integrity constraint rule *IC2.2* is invoked to check to make sure that the quota for the group has not been exceeded. At each stage, new data added to the event data file are shown in bold face in the figure.

From the above scenario, we see that an event occurrence can lead to the processing of individual rules as well as rule structures. The data generated by rule *CAA2.3* and the rule structure *RS2* are added to the event data file. Thus, the event data file now contains new data resulting from the application of the local rules at *Branch2*. This data is returned to *Branch1*. *Branch1* receives the updated event data from *Branch2* (*ED2*), *Branch5* (*ED5*), and *Branch8* (*ED8*) (denoted as step 2 in the figure). It then merges this data and sends out the new version to all branches (not shown in the figure) that have rules and rules structures applicable to the new version. Thus a second round of event data distribution is initiated. The process will continue until no rules and rule structures are applicable to the last version of event data. At this time, all involved branches would have received the final version of the event data, which can be used for their local decision-support and problem solving. The implemented prototype system runs on

multiple computers. The above scenario takes a total of 2 s for the requesting branch to send out its event data file to remote branches and for the remote branches to invoke the applicable rules and send the updated event data file back to the requesting branch.

4.2.2 Decomposition of rule structure

We use Figure 12 to present the technique used to decompose a rule structure into substructures for processing. A rule structure is a directed *graph*, in which rules (nodes) can be interconnected by *link*, *split*, *and-join* and *or-join* constructs (edges). An XML document however, organises its constituent elements in a *tree* structure. Each of the four structural constructs, when considered independently, can be represented using a tree (by means of the predecessor-successor relationships), but the combination of these constructs that forms a particular rule structure may not always be a tree, as can be seen from rule structures *RS2* and *RS5* in Figure 12. It is worth noting that if we break a rule structure into substructures, where each represents a single structural relationship between two or more rules, each substructure is a tree, and can now be represented using XML. This does result in the same rule being referred at a maximum of two times, first when describing the relationship where this rule is a successor, and the next when describing the relationship where this rule is a predecessor. Rules that have no predecessors or successors still appear only once in the representation.

For a given rule structure, substructures are generated as we move from the top to the bottom of the graph. At each level, we follow a left-to-right order to generate substructures. Taking *RS2* of Figure 12 as an example, the first substructure generated would be the *split* construct with *IC2.1* as the predecessor rule and *DR2.1*, *CAA2.1* as the successor rules. The second substructure would be the *and-join* with *DR2.1* and *CAA2.1* as the predecessor rules and *CAA2.2* as the successor rule. The third substructure would be the *link* with *CAA2.2* as the predecessor rule and *IC2.2* as the successor rule.

Decomposing a complicated rule structure into simpler substructures also facilitates its maintenance. Inserting a new relationship into an existing rule structure document requires only creating the appropriate substructure and inserting it into the correct position in the document. Similarly, deleting or modifying a structural relationship needs to address only the substructure that represents that relationship, without affecting other parts of the rule structure document.

4.3 Discussion

In this section, we consider some issues related to our system and highlight the approaches we have taken or are taking to resolve them. Some future tasks are also identified.

Distributed rules may form a cycle that causes a non-termination problem. Also, they may be inconsistent or conflicting. One possible approach is to analyse the published rules for contradictions, inconsistencies, and cyclic conditions (Baralis et al., 1998). However, due to the dynamic and independent nature of the rules in a collaboration federation, this strategy does not seem to be suitable. We have investigated both cycle avoidance and cycle detection and resolution strategies. So far, the most promising approach is the following. The host rule server determines possible rule cycles based on rule input and output characteristics. During runtime, the coordinating site monitors the executed rules to determine if a possible cyclic path is being followed. If a rule that has

already been executed once, is deemed to be applicable for the next round, it is deactivated from executing again, thus preventing a possibly non-terminating execution.

Inconsistencies and conflicts during rule execution in a collaboration federation may reflect differing expert opinions. However, there may be some rules to resolve these inconsistencies and conflicts. When an event coordinator determines that a particular data item has two or more values given by different sites, it checks with the host site to determine if there is a global resolution policy in place for that data item. If so, this policy is applied and the resolved value of the data item is used (e.g., by taking the average or minimum or maximum value). If not, all of the values are tagged with their site identifiers and are sent to all sites that contain applicable rules. Each site may have a local resolution policy in place to determine the resolved value. This resolved value is then used by any applicable global or local rules. In the absence of either a global or a local resolution rule, all relevant organisations will be informed of the detected inconsistency or conflict.

In a collaboration federation, since events, rules, and triggers can be defined by different collaborating organisations, it is very likely that there will be discrepancies in the terminologies used. People searching for registered events and web services need some form of common ontology to resolve these discrepancies. We are investigating the use of the OWL language to define an ontology for a particular application domain in a collaboration federation. Thus, the host site in the system architecture will also incorporate an ontology database to map the terms used in event, rule and trigger specifications to concepts and concept associations defined in the domain ontology. The site will also use an ontology manager to resolve discrepancies and identify similarities between the specified terms, and to facilitate search.

The notion of transaction is an important research issue. Each time an event occurs, multiple rounds of event data transmission and distributed rule processing can take place. They should be treated as a single transaction. In distributed databases, the focus is on maintaining data integrity, and hence a transaction commits only if all of its sub-transactions commit. In a collaborative e-business environment, the focus is on sharing as much knowledge associated with an event as possible. It may not be desirable to stop the rule processing if one site aborts because other sites may still produce useful data. The traditional ACID properties of transaction need to be re-examined and defined for an event-triggered, knowledge-sharing system like ours. Another important issue to be addressed is the specification and enforcement of trust and security in a collaborative e-business environment. This issue is out of the scope of this paper. In our previous work (Yang et al., 2002, 2005), we have introduced a trust agreement specification language and a trust-based security model for establishing inter-organisational security policies that govern the interaction, coordination, collaboration, and resource sharing of collaborating organisations. Interested readers are encouraged to look up these references.

The system architecture allows organisations to join a collaboration federation by installing the developed software and tools at their network sites. It is highly expandable and scalable because more computational power can be added as more shareable knowledge and more collaborating organisations are added to the federation. Also, since the system components are implemented as servers, multiple collaboration federations can be accommodated: i.e., an organisation can be a member of multiple federations and its servers can process different event data sets concurrently in multiple threads. However, as the number of organisations in a federation increases, the number of federations grows, and the number of event occurrences of different event types

increases, the performance of the entire network can deteriorate with a centralised host. For scalability, event types may have to be categorised and managed by multiple hosts.

Our approach assumes that organisations can specify data conditions for each rule in a precise manner and data provided by collaborating organisations are 100% correct. However, in some real-world situations, this may be an optimistic assumption. For example, an organisation may only be able to say with a certain degree of certainty that if some data conditions are met, a certain action should be performed. Also, the data provided by an organisation may be incomplete, inexact and conflicting. These data anomalies will affect what rule conditions are satisfied, what new data are generated and what operations are processed. This is a potential area of future work and we are interested in investigating the approach of defining and evaluating rules probabilistically.

5 Summary

In this paper, we presented our idea of managing dynamic event data and sharing multi-faceted knowledge among organisations that form a collaboration federation. Different aspects of knowledge are specified in different types of rules, rule structures, and triggers, which link events of interest to distributed rules and rule structures. The interoperation of distributed events, rules, rule structures and triggers for supporting decision-making is the main concern of this research. The approach taken for achieving their interoperation is to translate rules and rule structures into code and wrap them as web services for their discovery, invocation and interoperation in a web service infrastructure in a uniform manner. Event data are transmitted to collaborating sites that contain applicable rules and rule structures. The architecture of an event-trigger-rule-based system implementing the above ideas is described. Several R&D issues and techniques for transmitting event data and processing distributed rules are also discussed.

References

Baralis, E., Ceri, S. and Paraboschi, S. (1998) 'Compile-time and runtime analysis of active behaviors', *IEEE Transactions on Knowledge and Data Engineering*, Vol. 10, No. 3, pp.353–370.

Bassiliades, N., Vlahavas, I. and Elmagarmid, A. (2000) 'E-DEVICE: an extensible active knowledge base system with multiple rule type support', *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 5, pp.824–844.

Brownston, L., Farrell, R., Kant, E. and Martin, N. (1985) *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*, Addison-Wesley, Reading, MA.

Buchmann, A., Bornhövd, C., Cilia, M., Fiege, L., Gartner, F., Liebig, C., Meixner, M. and Mühl, G. (2004) 'DREAM: distributed reliable event-based application management', in Levene, M. and Poulovassilis, A. (Eds.): *Web Dynamics Adapting to Change in Content, Size, Topology and Us*e, Springer-Verlag, Germany, pp.319–352.

Business Rules Group (2000) *Defining Business Rules – What are They Really?*, Final Report, http://www.businessrulesgroup.org/first_paper/BRG-whatisBR_3ed.pdf

Carzaniga, A., Rosenblum, D. and Wolf, A. (2000) 'Achieving scalability and expressiveness in an internet-scale event notification service', *Proceedings of the Nineteenth ACM Symposium on Principles of Distributed Computing (PODC '00)*, Oregon, USA, pp.219–227.

Cover, R. (Ed.) (2000) *Simple Rule Markup Language*, http://xml.coverpages.org/srml.html

Cover, R (Ed.) (2002) *Business Rules Markup Language*, http://xml.coverpages.org/brml.html

Hollingsworth, D. (1995) *The Workflow Reference Model*, Document Number TC00-1003, Workflow Management Coalition, http://www.wfmc.org/standards/docs/tc003v11.pdf

Krishnamurthy, B. and Rosenblum, D.S. (1995) 'Yeast: a general purpose event-action system', *IEEE Transactions on Software Engineering*, Vol. 21, No. 10, pp.845–857.

Lee, M., Su, S.Y.W. and Lam, H. (2001) 'A web-based knowledge network for supporting emerging internet applications', *WWW Journal*, Vol. 4, Nos. 1–2, pp.121–140.

Loucopoulos, P. and Katsouli, E. (1992) 'Modelling business rules in an office environment', *ACM SIGOIS Bulletin*, Vol. 13, No. 2, pp.28–37.

Nonaka, I. and Takeuchi, H. (1995) *The Knowledge Creating Company*, Oxford University Press, New York, NY.

Riley, G. (2006) *C Language Integrated Production System*, http://www.ghg.net/clips/CLIPS.html

Rosenberg, F. and Dustdar, S. (2005a) 'Business rules integration in BPEL – a service-oriented approach', *Proceedings of the 7th International IEEE Conference on E-Commerce Technology*, Germany, July, pp.476–479.

Rosenberg, F. and Dustdar, S. (2005b) 'Towards a distributed service-oriented business rules system', *Proceedings of the IEEE Third European Conference on Web Services*, Sweden, November, pp.14–24.

Rouvellou, I., Degenaro, L., Chan, H., Rasmus, K., Grosof, B.N., Ehnebuske, D. and McKee, B. (2000) 'Combining different business rules technologies: a rationalization', *Proceedings of the OOPSLA 2000 Workshop on Best-practices in Business Rule Design and Implementation*, Minnesota, USA.

Rule Language in OWL (ROWL) (2004) http://www.cs.cmu.edu/~sadeh/MyCampusMirror/ROWL/ROWL.html

Rule Specification Language Schemas (2005) http://www.cise.ufl.edu/~spd/RuleBase.xsd, http://www.cise.ufl.edu/~spd/RuleStruc.xsd

Semantic Web Rule Language (2003) http://www.daml.org/2003/11/swrl/

Sowa, J. (2000) *Knowledge Representation: Logical, Philosophical and Computational Foundations*, Brooks Cole, Pacific Grove, CA.

Su, S.Y.W., Huang, C., Hammer, J., Huang, Y., Li, H., Wang, L., Liu, Y., Pluempitiwiriyawej, C., Lee, M. and Lam, H. (2001) 'An internet-based negotiation server for e-commerce', *VLDB Journal*, Vol. 10, No. 1, pp.72–90.

The Rule Interchange Format (2005) W3C Working Group, http://www.w3.org/2005/rules/

The Rule Markup Initiative (2000) http://www.ruleml.org

UDDI (2001) http://www.uddi.org

Ullman, J. (1982) *Principles of Database Systems*, 2nd ed., Computer Science Press, Rockville, MD.

Ullman, J. (1988) *Principles of Database and Knowledge-Base Systems*, Vols. I–II, Computer Science Press, Rockville, MD.

Wagner, G. (2003) 'The agent-object-relationship metamodel: towards a unified view of state and behavior', *Inf. Syst*., Vol. 28, No. 5, pp.475–504.

Wagner, G., Damasio, C. and Lukichev, S. (2005) *First-Version Rule Markup Languages*, REWERSE IST 506779, Deliverable I1-D3-1.

Warmer, J. and Kleppe, A. (1998) *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley Longman, Boston, MA.

Web Ontology Language (OWL) (2004) http://www.w3.org/TR/owl-features/

Widom, J. and Ceri, S. (1996) *Active Database Systems, Triggers and Rules for Advanced Database Processing*, Morgan Kaufmann, San Mateo, CA.

Yang, S., Lam, H. and Su, S.Y.W. (2002) 'Trust-based security model and enforcement mechanism for web service technology', *Proceedings of the Third VLDB Workshop on Technologies for E-Services*, Hong Kong, China, pp.151–160.

Yang, S., Su, S.Y.W. and Lam, H. (2005) 'A non-repudiation message protocol for e-commerce', *International Journal of Business Process Integration and Management*, Vol. 1, No. 1, pp.34–42.