ARML: an Active Rule Markup Language for Sharing Rules among Active Information Management Systems

Eunsuk Cho, Insuk Park, Soon J. Hyun, and Myungchul Kim

School of Engineering, Information and Communications University (ICU)
P.O.Box 77, Yusong, Daejon, 305-600, Korea
{escho,ispark,shyun,mckim}@icu.ac.kr

Abstract. As the information management paradigm shifts from data management to knowledge management, active rule support for various information services has become a practical approach to add intelligence to the existing information management systems. Various active rule systems have been developed using different rule description languages and, yet, their business logics are not sharable with each other. In this paper, we propose an XML-based ECA rule definition language, called Active Rule Markup Language (ARML). ARML provides an XML-based uniform rule description for the sharing of business rules among heterogeneous active rule processing systems. Business rules are made interoperable and therefore reusable through ARML when mapped to ARML rule description facility. We discuss the language construct, the proposed rule interoperability, and a prototype implementation.

1 Introduction

Active rule processing technology has been widely accepted to achieve one of the ultimate database system's goal of auto-everything in the course of decision-making. It minimizes (or eliminate, after all) users' intervention in processing a series of complex tasks and managing data automatically with the verification of integrity constraint, transaction, security, etc. Typically, active information management has rule-processing features based on an Event-Condition-Action (ECA) construct by which complicated business strategies are modeled and processed without looking into application programs and the underlying database. For example, an Internet bookstore that stores data about books, publishers, orders, etc., is accessed by a number of distributed users and when inventory of books run short and some threshold amount is reached, it should be desirable for the database management system to automatically place orders to the publishers of the books rather than waiting until the inventory manager detects it and reports the situation. There has been a great deal of effort paid in research and development of active data services and systems [1, 2, 5, 6, 8, 11, 12, 13, 16, 18].

Most of them have rule-processing features based on an Event-Condition-Action (ECA) construct and are centered on the notion of rule and event definition. This way, complex business logics of an enterprise can be represented in the form of ECA rule chains, and changes can be made without looking into application programs and the underlying database.

Different systems define different rule definition language facilities. Ariel uses Ariel Rule Language (ARL) and REACH uses REAL [1, 2]. Similarly, Starburst, AIMS, HiPAC, SAMOS use their own rule definition languages [6, 11, 12, 13]. Besides these languages, triggers viewed as another type of ECA rules in SQL99 comply with their rule description grammar [19].

They are system-dependent and have different syntactic features such as using different keywords for event, condition, and action descriptions. These syntactic diversities do not allow rule reusability and knowledge sharing, and thus demand rule developers of an enterprise to build from the scratch the entire business logics that are similar with existing business rules described in different rule language environments. Reading and understanding the rules described in the existing rule languages would be very difficult, unless designers have professional knowledge about the specific syntactic structures of them.

In this paper, we propose an XML-based rule definition language, called Active Rule Markup Language (ARML) to enable business logics defined in various rule languages to be shared and reused among different systems. With its easy-to-define rule description facility using XML, ARML provides a uniform ECA platform for heterogeneous active information systems. Using ARML, rules of the similar services with different systems can be easily understood and reused so as to save the cost for creating and modeling business rules from the scratch. For example, an Internet bookstore may send its business rules about contract and conditions related to pricing, amount of order, discount rate and cancellation represented in ARML to publishers. Then, the publisher will be able to understand the business rules of the customer to form marketing plans and make sales decisions although they have different active information systems.

ARML, by taking advantages of XML standard with its ease of understanding, excellent expressiveness and web representation power, is simple to use and inexpensive to implement by using well-developed XML APIs and XML utilities. It allows developers not to be dependent on the specific development tools and languages. Also, it is human-readable and its vocabularies are easily shared through Document Type Definition (DTD) [3]. Active rules can be presented in various ways by using eXtensible Stylesheet Language (XSL) [4, 9].

ARML defines a new tag set specifying ECA rule description, coupling mode, rule execution precedence, and meta-information. It represents the business rules and constraints with a series of method calls. It hides the detailed implementation in the form of methods so that business experts build the business processes easily by enumerating the required methods and filling out the constraint values. By simply modifying the sequence of method calls, adding the required method call, and changing the constraint values, the business rules and strategies of an enterprise can be comfortably maintained and updated at the change of business constraints without touching upon a complex mass of application programs.

There have been some efforts to employ XML facility in describing business rules [8, 14, 15, 16]. ActiveWeb introduced an idea of using XML-based active rules for deriving web views and for defining access control by user access behaviors. Reactive E-Service proposed the concept of active XML rules for pushing reactive services to

XML-enabled repositories. Although they used XML-based active rules for their applications, the main idea and goal are different from our work and not suitable to apply active rules to the general active information services because their XML-based active rules focus on the specific reactive applications, such as web personalization and pushing reactive services. RuleML, CommonRules, and BRML support XML-based business rules for Web communication between applications on the various rule systems. Although they deal with reaction rules based on ECA construct, it is also hard to apply them to the active information management since they employ many general features of multiple forms of rule systems, such as SQL, Prolog, logic programming, production rules, and ECA rules. With these they intend to integrate rule-markup approaches and to package the rule aspects of various domains such as engineering, commerce, law, and Internet.

The rest of this paper is organized as follows. In Section 2, we describe ARML construct and its semantics according to DTD. Section 3 introduces ARML layer for rule interoperability, and gives an example of ARML and its interpretation. Section 4 describes the implementation of ARML. We conclude this paper and remark future work in Section 5.

2 ARML Approach and DTD

ARML structure follows the rule definition language of AIMS and further includes some features of other rule languages, such as a table for rule specification and a rule set name for rule grouping [1, 6]. Table 1 shows rule definition language structures used in different active database systems implementations. The *DEFINE RULE*, *create rule*, and *define rule* clauses represent the beginning of the rule description and define a rule name for identifying the rule. The *EVENT*, *when*, and *on* keywords are used for specifying the event. The *CONDITION* and *if* specify the condition expression evaluated when rules are triggered. The *ACTION* and *then* imply the action part of the rule. The *AFTER*, *BEFORE*, *precedes*, *follows*, and *priority* specify the rule priority among rule sets triggered by the same event.

Beside these general rule features, each rule language has its own syntactic and semantic features, such as coupling modes, rule-set name, etc. ARML integrates ECA features of rule definition languages and expresses active rules in XML by encapsulating the detailed implementation of applications into a method call.

AIMS	Starburst	Ariel
DEFINE RULE	**create rule** name **on** table	**define rule** rule-name
rule-name-specification	**when** triggering-operations	[**in** ruleset-name]
EVENT event-specification	[**if** condition]	[**priority** priority-val]
CONDITION	**then** action-list	[**on** event]
[DEC declaration]	[**precedes** rule-list]	[**if** condition]
[EXP expression]	[**follows** rule-list]	**then** action
ACTION action-specification		
MODE ec-coupling, ca-coupling		
AFTER precedence-rule-list		
BEFORE follow-rule-list		

Table 1 Active Rule Definition Language Structures

Table 2 shows ARML DTD. It consists of the rule description part and the method call part representing vocabularies and structural information of an XML-based active rule.

Its design principles are (1) to make ARML rules easily bound with any rule definition language, (2) to achieve readability, and (3) to give uniform rule language features such as rule priorities, rule structure, and coupling modes. ARML's vocabulary and structural information can easily be shared by other languages through DTD. It gives developers an efficient way to implement ARML interpreter with a human-readable structure and semantics of ARML rules. It consists of two major part, rule description and method call by way modifying the DTD of XML-RPC, a simple protocol to represent the request and response of a remote procedure call in the form of XML. Modifying XML-RPC into ARML, we added a new element, *variable* to express variable type parameter. ARML makes use of only the requesting part of XML-RPC to describe condition predicates and actions but not the response part of it. The element *rule* as the root element of an ARML rule should have seven sub-elements; *ruleDef, event, condition, action, coupling, precedence,* and *info*. The element *ruleDef* defines the rule with a rule name, a rule set, and a rule table. The element *info* represents metadata of the ARML rule. Other sub-elements represent the notions of traditional ECA construct as follows.

```
<!ELEMENT rule (ruleDef, event ,
     condition ,action , coupling, precedence,
     info)>
<!ELEMENT ruleDef
     (ruleName, table?, ruleSet?)>
<!ELEMENT event (eventName | algebra)?>
<!ELEMENT condition
    (methodCall | boolean | algebra)?>
<!ELEMENT action (methodCall+)>
<!ELEMENT coupling (ec? , ca?)>
<!ELEMENT precedence (after?, before?)>
<!ELEMENT info
     (designer*, description?, category?)>
<!ELEMENT algebra (and | or | seq)>
<!ELEMENT and
     ((eventName | methodCall)+ , algebra?)>
<!ELEMENT or
     ((eventName | methodCall)+ , algebra?)>
<!ELEMENT seq
     ((eventName | methodCall)+ , algebra?)>
<!ELEMENT before (ruleList?)>
<!ELEMENT after (ruleList?)>
<!ELEMENT ruleList (ruleName)*>
<!ELEMENT ruleName (#PCDATA)>
<!ELEMENT table (#PCDATA)>
<!ELEMENT ruleSet (#PCDATA)>
<!ELEMENT eventName (#PCDATA)>
<!ELEMENT designer (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT category (#PCDATA)>
<!ELEMENT date  (#PCDATA)>
<!ELEMENT ec (#PCDATA)>
<!ELEMENT ca (#PCDATA)>
<!ELEMENT methodCall
     (methodName, params)>
<!ELEMENT methodName (#PCDATA)>
<!ELEMENT params (param*)>
<!ELEMENT param (value)>
<!ELEMENT value (i4 | int | boolean | string |
     dateTime.iso8601 | double | base64 |
     variable | struct | array)>
<!ELEMENT i4 (#PCDATA)>
<!ELEMENT int (#PCDATA)>
<!ELEMENT boolean (#PCDATA)>
<!ELEMENT string (#PCDATA)>
<!ELEMENT double (#PCDATA)>
<!ELEMENT variable (#PCDATA)
<!ELEMENT dateTime.iso8601
     (#PCDATA)>
<!ELEMENT data (value*)>
<!ELEMENT base64 (#PCDATA)
<!ELEMENT array (data)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT member (name, value)>
<!ELEMENT struct (member*)>
```

Table 2 DTD of ARML

The element *event* may have a sub-element *eventName* or *algebra.* The element *eventName* represents a primitive event and *algebra* depicts a composite event by combining primitive events with the sub-element of *algebra*. Some algebraic operators, such as disjunction, conjunction, sequence, not, closure, and history are used for combining multiple events [5]. In the current implementation, we employ three operators: conjunction, disjunction, and sequence that are defined as the sub-elements *and*, *or,* and *seq,* respectively. The element *condition* may consist of a sub-element *methodCall*, *boolean,* or *algebra*. The sub-element *methodCall* represents application-specific conditions. When a rule is triggered, the boolean value of the condition method returns to determine whether an action be triggered or not. The sub-element *boolean* is used for specifying absolute condition, i.e., true or false. The sub-element *algebra* is required for specifying multiple conditions.

The element *action* has a series of sub-element *methodCall*s to describe simple or complex business logic. The element *methodCall* of ARML follows the syntax and the structure of XML-RPC. XML-RPC provides ARML with a well-defined representation of a method call. The element *coupling* has two sub-elements: *ec* and *ca*. They specify the transactional relationship between event and condition, and condition and action, respectively. We consider three possible coupling modes: immediate, deferred, and decoupled. The element *precedence* can be composed of sub-element *before*, *after*, or both. Each has a sub-element *ruleList* to define a partial order of triggered rules which schedules the rule execution. When rule R1 specifies rule R2 in its BEFORE list and both rules are triggered, R1 will be executed before R2. Likewise, if R1 specifies R2 in its AFTER list, this indicates that when both rules are triggered, R2 will be executed before R1. The element *info* used as a metadata has multiple elements of *designer*, *description*, and *category*. The sub-element *designer* represents who develops the rule; the *description* specifies the usage of the rule; and *category* element specifies the category of the rule. Meta information of a rule can be used to make rule indexes in searching and categorization.

3 Interoperability of ARML Rules

3.1 Interoperability by ARML Rule Mediation

The rules written in a rule language cannot be used by another active rule system and are hardly understood each other because of different syntactic and semantic characteristics. ARML plays the role of mediation between active rule systems with different language environments. In that, business logics represented and implemented in rules in a system can be made operable by other systems through ARML, thus saving the rule designers the trouble of developing and implementing complex rules from scratch.

For this, ARML rules and the underlying system rules need a language translation interface. It translates an ARML active rule into system-executable rule code and binds the method call of ARML with the method of business objects. It finally generates an active rule written in a system-dependent rule definition language.

ARML integrates various ECA features into a uniform rule description in XML. It makes a business logic modeled in active rules sharable through a uniform rule description. Once a business rule running on a specific active database is converted into an ARML active rule, it can be executed on other active systems through ARML facility. Fig. 1 shows how ARML supports interoperability through ARML. The rule mediation architecture through ARML translates an ARML active rule into system-executable rule code and binds the method call of ARML with the method of business objects.

In Fig. 1, each system should have a set of classes which implement business tasks bound with ARML method calls. The *Rule #1*, an ARML active rule enters into each ARML rule interpreter and they generate *AIMS_Rule #1, Star_Rule #1,* and *Ariel_Rule #1* written in their own rule definition languages. At the same time, the rule interpreter binds the method calla of an ARML rule to the methods of a class. After each interpreter translates and binds an ARML rule (i.e., *Rule #1*), the systems are able to execute the generated rules (i.e., *AIMS_Rule #1, Star_Rule #1, and Ariel_Rule #1*).

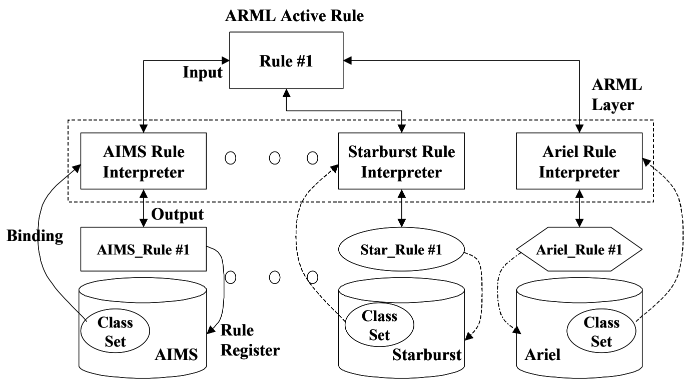

Fig. 1 Rule Mediation Architecture

3.2 Rule Interpretation: Example

In this section, we show an example of an ARML rule and its interpretation into a specific rule language to show how business logics can be shared by two different systems through ARML. We consider two rule definition languages, AIMS and Starburst. Their active rules can be described and implemented in ARML and vice versa through the mediation interface.

As an example, we use the rule *sal-control* that controls employees' salaries given in [6]. This rule implies "whenever a new employee is inserted into employee table, or his/her salary field is changed, check whether the average salary exceeds 50 or not. If exceeds, the rule deletes the employees whose salary exceeds 80."

```
<rule>
  <ruleDef>
    <ruleName>sal-control</ruleName>
    <table>employee</table>
  </ruleDef>
  <event>
    <algebra>
      <or>
        <eventName>
          inserted
        </eventName>
        <eventName>
          updated salary
        </eventName>
      </or>
    </algebra>
  </event>
  <condition>
    <methodCall>
      <methodName>
        employ.checkSalary
      </methodName>
      <params>
        <param>
          <value><int>50</int></value>
        </param>
      </params>
    </methodCall>
  </condition>
  <action>
    <methodCall>
      <methodName>
        employ.deleteHighSalary
      </methodName>
      <params>
        <param>
          <value><int>80</int></value>
        </param>
      </params>
    </methodCall>
  </action>
  <coupling>
    <ec>immediate</ec>
    <ca>immediate</ca>
  </coupling>
  <precedence>
    <before>
      <ruleList>
        <ruleName>cascade</ruleName>
      </ruleList>
    </before>
  </precedence>
  <info>
    <designer>escho </designer>
    <description>
      salary control
    </description>
    <category>business</category>
  </info>
</rule>
```

Table 3 Example of an ARML Rule, *sal-control*

Table 3 shows the description of the rule. An ARML active rule begins by declaring the <*rule*> tag, the document element. It should have the seven sub-elements that define specific operations of ECA construct and meta-information of a rule (refer back to Section 2 for details).

The <*ruleDef*> defines a rule name as *sal-control* and a rule table on which the rule is defined as *employee*. A rule name is unique in the system so that the application can identify a rule by its name. The <*event*> defines a composite event. It consists of two primitive events, *inserted* and *updated salary* which are connected by <*or*> tag. Each event name will later be translated into a proper event description according to the system-dependent rule definition language, such as *update_emp_salary* and *insert_emp* of AIMS event description (shown in Table 4).

The <*condition*> specifies constraints of a rule to determine whether the action of a rule be executed or not. The example uses <*methodCall*> for the condition, *employ.checkSalary* to determine whether the average salary exceeds 50 which is encapsulated by <*params*> tag.

The <*action*> may have several <*methodCall*> tags to define the rule's action which implements the business logic. The *employ.deleteHighSalary* is bound with the method deleteHighSalary of a business object (i.e., employ) for executing the actual business operation. It deletes employees whose salary exceeds the threshold value of 80 that has been passed by <*params*> tag. The <*coupling*> element has <*ec*> and <*ca*> tags represents the transactional relationship among event, condition, and action. If the system does not support coupling modes, they are merely ignored during the interpretation. The couple modes in the example shows are *immediate*.

The <*precedence*> tag defines the rule priority between rules triggered by the same event. Although a way to define a rule priority is different among active systems, they can be expressed with <*before*> and <*after*> tags and encoded into the proper rule code by ARML interpreter. Since the rule *cascade* is defined in the <before> tag of *sal-control*, the rule *sal-control* has a lower priority than *cascade*, thus *sal-control* is triggered after executing *cascade* when both of them are triggered by the same event. The <*info*> is used for specifying meta-information of a rule. It does not affect rule execution and it is ignored during the interpretation. It has <*designer*>, <*description*>, and <*category*> tags. It is useful for searching and categorization of rules.

Table 4 shows the rule *sal-control* written in AIMS and Starburst rule definition language. They are interpreted from ARML active rule in Table 3. The AIMS rule has a composite event, *update_emp_salary OR insert_emp*. It calls the methods *checkSalary* and *deleteHighSalary* of *Employ* class for condition and action, respectively. The detailed implementation of two methods is in class *Employ*. Since AIMS supports a method call in active rules, an AIMS rule in Table 4 can be registered and triggered without modification. During the interpretation, AIMS interpreter ignores the <*table*> tag of ARML rule because it is not applicable in AIMS.

4 An ARML-based Active Information Management System

4.1 Architecture

We have implemented an ARML-defined rule management system called AIMS/ARML as an extension to our prototype active information management system, AIMS. The original AIMS adopted a layered architecture and Java facility to support active capability on the top of the conventional relational database systems. It provides composite event description and detection, three typical coupling modes (i.e., immediate, deferred, and decoupled), rule priorities, Java-based rule definition language, termination analysis, etc.

Fig. 2 shows the architecture of our prototype system in which ARML and AIMS are integrated in a layered implementation consisting of two major parts: ARML Interpreter as an ARML layer and AIMS as an underlying active information system. The ARML Interpreter parses ARML rules and associates ARML method call with the method of business objects. AIMS gives a high degree of automation to the existing database through ECA rule processing.

AIMS	Starburst
//rule definition	//intermediate Rule Definition
DEFINE RULE sal-control	**create rule** sal-control **on** employee
EVENT	**when** inserted, updated(salary)
update_emp_salary OR insert_emp	**if**(select avg(salary) from emp) > 50
CONDITION	**then**
DEC	employ.deleteHighSalary();
Employ employ = new Employ(aof);	**precedes** cascade
EXP	
employ.checkSalary(50);	//class definition
ACTION	class Employ {
employ.deleteHighSalary(80);	void checkSalary(int avg){
MODE	//checking procedure
IMMDIATE, IMMEDIATE	}
BEFORE cascade	void deleteHighSalary(int threshold){
	//deleting procedure
//class definition	}
class Employ {	}
void checkSalary(int avg) {	
//checking procedure	// complete rule definition
}	// after interpretation
	create rule sal-control **on** employee
void deleteHighSalary(int threshold){	**when** inserted, updated(salary)
//deleting procedure	**if**(select avg(salary) from emp) > 50
}	**then**
}	delete from emp
	where em-no in (select emp-no from
	inserted union select emp-no from
	new-updated)
	and salary > threshold
	precedes cascade

Table 4 Active rules, *sal-control* interpreted from the ARML rule in Table 3

ARML Interpreter consists of *Parser, Method Binder,* and *Comparator*. *Parser* checks the validation and well-formedness of an ARML rule and it generates the document tree of an ARML rule reflecting the hierarchical structure and semantics of an active rule. *Method Binder* receives a document tree as an input and associates XML-based method call with the method of business objects to generate a *Rule Code*. The connection between system interface and ARML interface is made through *Method Binder*. It requests *Comparator* to check whether the association be made by inspecting the business objects. That is, it checks whether the methods specified in ARML exist in *Class Storage*. If there is no method to be bound, *Comparator* issues an error. If there is no error found on validation, well-formedness, and association, a system-executable *Rule Code* is generated. *Rule Register* loads *Rule Code* to the rule processor. *Class Storage* retains the business objects associated with the method call in ARML. A loaded *Rule Code* is executed just like the original active rule specified in AIMS rule definition language. That is, the rule is triggered when the event detector of the system detects the proper event and the condition is satisfied.

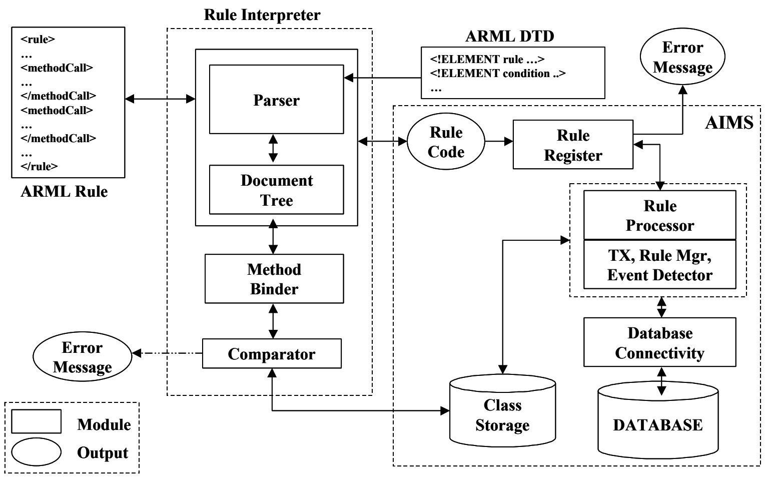

Fig. 2 Architecture of AIMS/ARML

4.2 Rule Interpretation Process

In order for ARML rules to be executed on AIMS, rules are compiled and registered to the *RuleBase*. ARML interpreter receives the DTD of ARML and ARML rules as inputs. It communicates with *Class Storage* for checking the validity of method binding. ARML interpreter generates intermediate rule definition written in AIMS rule definition language. The generated AIMS rules are sent to *rule compiler* which then reads rule definition and generates the source file of a rule class, which is again converted to Java class file. Then, it invokes Java compiler and redirects Java compiler's output to itself. It creates a rule object after it compiles a rule description. Finally, it registers the rule in *RuleBase* to be triggered by AIMS. These ARML rule interpretation and rule object generation steps are integrated seamlessly so that the users can easily define and execute their rules. Fig. 3 shows the rule compilation process in AIMS/ARML.

The ARML interpretation procedure executed within ARML Interpreter is further detailed in Fig. 4. First, ARML parser checks well-formedness and validation of an ARML document. Since ARML is one of XML applications, this verification step is required for the remaining rule processes. Well-formed ARML means that an ARML rule comply with all XML syntax rule and all elements are correctly positioned. ARML's strict adherence to ordering and nesting rules allows data to be parsed and handled much more quickly than when using markup languages without these constraints. And, validation can ensure that an ARML active rule received from other application or newly written for the business application is correctly formatted. This helps avoid errors in referring ARML active rules from erroneous data input. If there is any mismatch between DTD and active rules, ARML parser reports this situation and the rule designer can fix it.

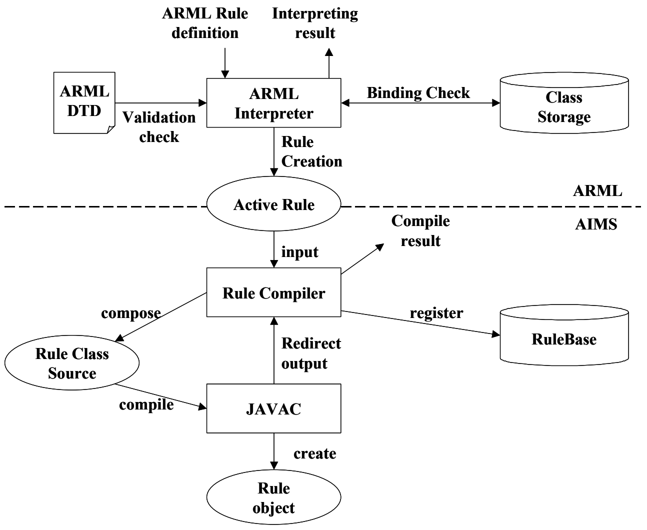

Fig. 3 ARML-AIMS Rule Compilation Structure

After checking validation and well-formedness, ARML parser generates DOM tree containing all information of an ARML rule. To generate DOM tree, we use JDOM package, which gives many advantages such that it provides a Java-centric, so it is very well with out system because AIMS and ARML are implemented in Java; it allows a user to deal with an XML document in tree form without the idiosyncrasies of DOM; it allows very quick parsing because it is very light; it supports validation through DTDs at building DOM tree; and it is concrete classes not abstract.

In *extracting the first level element* step, ARML parser analyzes DOM tree and it extracts root's sub-elements i.e., root's sub-trees, for interpreting each element separately. Each sub-element is processed for generating the rule code. We use a divide-and-conquer strategy so as to reduce implementation complexity. After extracting, *Interpretation and Binding Method* is executed on each sub-element. *Method Binder* examines each node and extracts rule information. It composes the method call and checks the possibility of method binding through the communication with *Comparator*. *Comparator* talks with *Class Storage* to reply the request from *Method Binder* and returns binding information. *Class Storage* is constructed by parsing the business object source file and extracting the corresponding method.

During *Interpretation and Binding Method* step, partial rule codes related with each root's sub-element are generated. These partial rule codes are integrated into the rule source to complete the rule code and the rule source is stored in a file during *Integrating Code Set* and *Creating Rule Code* processes. The generated rule source is passed to rule compiler and registered for rule triggering.

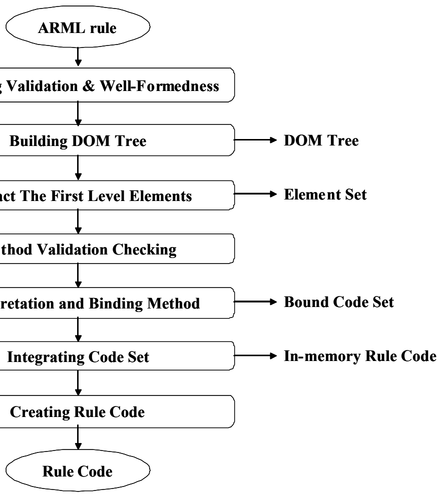
ARML rule
Checking Validation & Well-Formedness
Building DOM Tree
DOM Tree
Extract The First Level Elements
Element Set
Method Validation Checking
Interpretation and Binding Method
Bound Code Set
Integrating Code Set
In-memory Rule Code
Creating Rule Code
Rule Code

Fig. 4 ARML-AIMS Interpretation Procedure

5 Conclusion

In this paper, we proposed an XML-based ECA rule definition language, called Active Rule Markup Language (ARML). ARML provides rule definition facility with the uniform description of active rules for heterogeneous systems and therefore makes complex business rules exchangeable among different systems. ARML allows the rule developers to reduce the time and effort in rule development and maintenance by reusing and sharing business rules with other systems. XML facility makes active rule description simple to use and easy to implement. With rule interpreters between ARML and underlying rule systems and APIs representing the business tasks, active rules can be shared, reused, and exchanged between business partners saving them the cost of developing a large number of complex business rules.

In the present implementation, ARML active rules are defined manually, that is, application rule designer writes ARML tags and fills out the context. A graphical rule management tool would make users define active rules more conveniently and diminish errors through automatic code generation. The use of meta-information is another task to be accomplished as a future work.

References

1. Hanson, E.: The design and implementation of the Ariel active database rule system. IEEE Transactions on Knowledge and Data Engineering. (1996) Vol. 8, Issue: 1, 157-172
2. Buchmann, A. P., et al.: The REACH active OODBMS. Proc. of the ACM SIGMOD International Conference on Management of Data. (1995) 476
3. Bray, T., et al.: Extensible Markup Language (XML) 1.0 (Second Edition) W3C Recommendation 6 October 2000
4. Kaplan, A., and Lunn, J.: FlexXML: engineering a more flexible and adaptable web. IEEE Information Technology: Coding and Computing. (2001) 405-410
5. Paton, N. W., Díaz, O.: Active database system. ACM Computing Serveys. (1999) 31(1), 63-103
6. Widom, J.: The Starburst active database rule system. IEEE Transactions on Knowledge and Data Engineering. (1996) Vol. 8, Issue: 4, 583-595
7. UserLand Software, Inc.: XML-RPC. information available at http://www.xmlrpc.com
8. Kiyomitsu, H., Takeuchi, A., Tanaka, K.: ActiveWeb: XML-based Rules for Web View Derivations and Access Control. ITVE 2001, IEEE. (2001) Vol. 23, No. 6, 31-39
9. Adler, S., et al.: Extensible Stylesheet Language (XSL) Version 1.0 W3C Candidate Recommendation 15 October 2001
10. Boley, H., et al.: Rule Markup Language. information available at http://www.dfki.uni-kl.de/ruleml/
11. Min, H.J.: Design and Implementation of an Object-oriented Rule Management System for Active Database Services. Master Dissertation, ICU, Korea. (2000)
12. McCarthy, D., Dayal, U.: The Architecture Of An Active Data Base Management System. Proc. of the ACM SIGMOD. (1989) 215-223
13. Gatziu, S., et al.: SAMOS: An active object-oriented database system. IEEE Data Engineering, Special issue on active databases. (1992) 15(1-4): 23-26
14. Robin Cover and OASIS: Business Rules Markup Language (BRML). information available at http://www.oasis-open.org/cover/brml.html
15. Grosof, B., Chan, H., et al.: IBM CommonRules home pages. information available at http://www.research.ibm.com/rules/ and http://alphaworks.ibm.com
16. Bonifati, A., et al.: Pushing Reactive Services to XML Repositories using Active Rules. 10th International World Wide Web Conference. (2001) 633-641
17. jdom.org: JDOM. information available at http://www.jdom.org
18. Gatziu, S., Dittrich, K. R.: Events in an Active Object-Oriented Database System. In Proceedings of the 1^{st} International Workshop on Rules in Database Systems. (1993) 23-39
19. Türker, C., Gertz, M.: Semantic integrity support in SQL:1999 and commercial (object-) relational database management systems. VLDB Journal (2001) 10(4): 241-269