Onspect: Ontology Based Aspects for Adaptive Distributed Systems

Parisa Rashidi
School of Electrical Engineering and
Computer Science
Washington State University
Pullman, Washington
Email: prashidi@eecs.wsu.edu

***Abstract*— Interoperability as an important issue of software engineering, currently is supported at data type level and specification level while it's usually ignored at semantic level. To achieve semantic level interoperability, ontology modeling as a powerful means of expressing and sharing knowledge can be used to add meaningful standard semantics to syntactic annotations, leading to more abstract expressions. In current paper we focus on using ontology for modeling semantic pointcuts in aspect oriented programming (AOP) paradigm. Current AOP models, like many other programming models, primarily rely on a syntactic representation and ignore pointcut expression at semantic level. We present a new approach of pointcut modeling based on semantics instead of underlying program's syntax, using ontology modeling which will provide us with the facility to conceptually modularize crosscutting concerns. We also show how so called Ontology based Aspects (Onspects) dynamically modularize a distributed crosscutting concern in heterogeneous modules without knowledge of detailed implementation. Finally, as an example, we demonstrate utilization of Onspects in a distributed system to apply dynamic on-fly updates to a set of heterogeneous clients.**

I. Introduction

An important issue in software engineering is interoperability between heterogeneous components which is defined as the ability of software components to interact with each other regardless of implementation language or the underlying hardware [1]. Currently, interoperability is supported at *data type* level and *specification* level, mostly being concerned with independent data representation and utilization of abstract data types. Interoperability has also been defined at semantic level to deal with design intent and semantics as opposed to syntactic form.

Compared to the first two types of interoperability which have been frequently studied and investigated [2], semantic level interoperability has received less attention due to some problems such as ambiguity at conceptual levels and lack of consensus on concepts' representation. Nevertheless, achieving semantic interoperability is an important issue and can have many advantages such as providing a common conceptual understanding of application in a heterogeneous system.

In order to achieve semantic level interoperability, different approaches can be taken; the approach we choose is based on using ontology to model semantic pointcuts. An ontology is an explicit specification of a conceptualization [3] which can be used as a powerful tool for adding semantics to syntactic forms. Ontology and ontology modeling started in AI and philosophy and became popular with the idea of semantic web, resulting in ontology modeling languages such as RDF [4], OIL+DAML [5], OWL [6], besides ontologies such as Wordnet [11], GUM [12], and SENSUS [13]. Also a different class of ontology modeling in AI led to complex ontology modeling systems such as GFO [7] and OpenCyc/ResearchCyc [8] which are able to capture foundation ontologies.

The knowledge that is represented by ontology can be shared and reused unambiguously among different implementations and organizations [3]. Huge effort has been put into constructing ontology of different domains such as medical informatics or classification of music, but programming domain ontology has received little attention. To our knowledge, no ontology about programming domain knowledge has been developed, except for recent work done by Su et al. [10] which suggest as a basic approach to represent program ontology in OWL [6].

In current paper, we try to adapt an ontological based approach to Aspect-Oriented programming (AOP) paradigm. AOP improves separation of concerns, using basic elements of `pointcut` and `advice` to modularize crosscutting concerns and so makes it easier to evolve and maintain a software system [14]. Current mainstream AOP techniques separate crosscutting concerns based on syntax despite the fact that a concern is more a semantic matter. Lack of semantics in AOP has led to problems such as fragile pointcuts which is a result of aspect's tight dependence with syntax.

In current paper, we present a new approach of aspect modeling(Onspect), based on semantics instead of relying on underlying program's syntax using ontology modeling. This provides us with the facility to conceptually modularize crosscutting concerns which in turn can resolve many problems such as problem of fragile aspects. In our model, Onspects dynamically modularize a distributed concern in crosscutting heterogeneous modules without knowledge of detailed implementation.

To represent programming domain ontology, we introduce a formal model of programming domain ontology based on concepts, attributes, relationship and constraints to model basic semantic units. In order to map formal model into a

programming paradigm, we utilize annotation facilities provided in Java 1.5 to introduce simple annotation templates which represent program semantics. Annotation templates are converted into OWL for further processing; we avoid using OWL like annotations' style in order to keep annotations comprehensive and simple. Reasoning and querying is achieved via OWL standard set of reasoning operators. We define semantic pointcut using semantic quantifiers which consist of expressions that refer to elements of semantic annotation in order to refer to concerns semantically.

In current model, in order to provide adaptive behavior Onspects can be dynamically added to remote heterogeneous systems which requires dynamic weaving capability besides a distributed pointcut mechanism. For this purpose, we use JAsCo framework [16], [17] to utilize dynamic weaving mechanism. Distribution also can be achieved using serialization and communicating with an Onspect daemon.

The remainder of the paper consists of a more thorough review of current AOP models' limitations in section II and ontology definition in section III. Section IV describes proposed solution in detail and in section V, as an example, utilization of Onspects in a multi-agent system is demonstrated. This is followed by conclusions and future work in section VI.

II. Current AOP models Limitations

Aspect-oriented programming (AOP) was introduced to capture and untangle the crosscutting concerns with the main objective of improving the separation of concerns both in design and implementation. Using AOP programmer can implement crosscutting concern as a separate modularity unit, called aspect, instead of scattering these concerns over all program. In order to define how corresponding modularization will take place for aspect oriented programming languages, the join point model (JPM) was introduced which mainly consists of three basic elements [14]:

- A set of points, called `join points` that represent all locations in code where a concern is scattered over.
- The `pointcut` definition language, that identifies corresponding join points.
- Corresponding semantics at join points called `advice` that are woven into base code in location identified by `join points`.

Despite AOP's increasing success, it has still many limitation and shortcomings, one of these limitations is that almost all of AOP languages rely on a mechanism too tailored on the syntax of the program to manipulate. For example, AspectJ provides a set of primitives such as `call`, `execution`, `set` and `get` which can be used in combination with wild cards (+, $*$) and boolean operators (||, &&, !) to specify a pointcut. Using this approach, one has to explicitly refer to all methods which are involved in a pointcut one by one, making pointcut definition complicated, error prone and also fragile to changes in the base code. This problem is usually referred to as fragile pointcut. A trivial solution would be to use naming conventions to convey semantics, for example to restrict every method which is involved in logging concern, to end with "log" and then to use wild cards to express corresponding pointcut. But it's not a scalable solution to convey semantics with large number of concerns, besides there is always danger of having name conflicts and unwanted triggered joinpoints. This problem will even be more evident in distributed system with distributed pointcuts.

So, relying only on underlying syntax can be problematic, leading to limited expressive power, but using a semantic based approach developer does not need to explicitly specify how to find joinpoints, rather developer specifies the goals of query and it's up to program to find joinpoint location. This can be compared to approaches of relational databases which work based on declarative queries to extract data, while approaches like hierarchical database models and the network database models have an explicit form of how to traverse data in order to find desired data.

Our approach in this paper to model semantic aspects is based on ontology. Using a semantic description of joinpoints, pointcut description would not be effected by changes in base code and also would not suffer from triggering unwanted joinpoints. Besides, as concerns are more a semantic concept than a syntactic one, it seems appropriate to query at semantic level, being also easier for developers to identify concerns.

Relying on ontology, also provides the ability to identify joinpoint in remote heterogeneous modules without knowing about implementation details. This provides us with semantic level interoperability at pointcuts.

III. Ontology

Ontologies are used in artificial intelligence, semantic web, software engineering and information architecture as a form of knowledge representation about the world or some part of it. The term ontology has its origin in philosophy, where it is the name of a fundamental branch of metaphysics concerned with existence. According to Gruber [3], the meaning of ontology in the context of computer science, is a description of the concepts and relationships that can exist for an agent or a community of agents. Another definition identifies an ontology as an explicit specification of a conceptualization that describes concepts and relationship between concepts in semantic and knowledge level in a universal and machine-understandable way [3].

Ontologies can play an important role; knowledge captured in ontologies can be used to annotate data, generalize or specialize concepts, drive intelligent user interfaces and even infer entirely new information. There are different kinds of languages for representing ontology, basically categorized as:

- *Graphical notations*: Semantic networks, Topic Maps
- *Logic Based*: Description Logics, Rules, First Order Logic, Conceptual graphs
- *Probability and Fuzzy*

Recently, huge effort has been put into defining and constructing ontology of different domains, such as medical informatics or classification of music. Despite the fact that much effort has been put into constructing ontology of different domains, little effort has been put into defining and

exploiting programming domain ontology. To our knowledge, no ontology about programming domain knowledge has been developed, except for recent work done by Su et al. [10]. They suggest as a basic approach to represent program ontology in OWL [6].

We use OWL (DL version) [6] to model ontology, it is currently one of the most popular languages for ontology modeling as part of World Wide Web Consortium (W3C) formal standards and is a revision of the DAML+OIL [5]. Because OWL is written in XML, OWL information can be easily exchanged between different types of computers using different operating systems and application languages. It provides a standard set of operation such as $\cup, \cap, \neg, \exists, \forall, \leq, \geq$ to query and reason about ontology. It also provides a set of axioms [6] as shown in Table I, we use a smaller subset of these axioms in our framework as Ω in Table II.

OWL Axioms	
SubClassOf	$C_1 \sqsubseteq C_2$
EquivalentClass	$C_1 \equiv C_2$
DisjointWith	$C_1 \sqsubseteq \neg C_2$
SameIndividualAs	$\{x_1\}\, equiv\, \{x_2\}$
DifferentForm	$\{x_1\} \sqsubseteq \neg \{x_2\}$
SubPropertyOf	$P_1 \sqsubseteq P_2$
EquivalentProperty	$P_1 \equiv P_2$
InverseOf	$P_1 \equiv \bar{P}_2$
TransitiveProperty	$P^+ \sqsubseteq P$
FunctionalProperty	$T \sqsubseteq \leq 1P$
InverseFunctionalProperty	$T \sqsubseteq \leq 1P^-$

TABLE I
OWL AXIOMS

SubClassOf	$C_1 \sqsubseteq C_2$
EquivalentClass	$C_1 \equiv C_2$
DisjointWith	$C_1 \sqsubseteq \neg C_2$
SameIndividualAs	$\{x_1\}\, equiv\, \{x_2\}$
InverseOf	$P_1 \equiv \bar{P}_2$
EquivalentProperty	$P_1 \equiv P_2$

TABLE II
SUBSET OF AXIOMS USED - Ω

IV. PROPOSED SOLUTION

In following, we describe our proposed solution in detail. First we provide a formal definition of our framework and in subsequent section, we provide formal framework mappings to Java annotation and also to OWL format.

A. Formal Ontology Definition

In general, ontology construction comprises of two primary steps; first, defining an appropriate formal template for representing knowledge and second, defining contents of template. We define formal template of programming domain ontology as following:

$$O = \langle C, A^C, R, H, X \rangle \tag{1}$$

The first element C, is a set of concepts where each individual concept is denoted as c_i and its attribute are denoted by $A^C(c_i)$. Set C is defined as:

$$\begin{aligned} C &= C_1 \cup C_2 \cup C_3 \cup C_4 \\ C_1 &= \{a \mid a \in Agents\} \\ C_2 &= \{b \mid b \in Behaviors\} \\ C_3 &= \{s \mid s \in Subjects\} \\ C_4 &= \{r \mid r \in Roles\} \end{aligned} \tag{2}$$

As it can be seen, four basic concept sets represent conceptual space of a program, the first one is $Agents$ set which represents the program and its functionality as a whole, $Behaviors$ describes set of methods and functionalities of agent, $Targets$ refers to any object that is used by a $Behavior$ and $Roles$ refers to role of object used as $Target$ in a $Behavior$.

A^C is a collection of attribute for concepts such as if c_i is a concept in C , then its attribute can be denoted by $A^C(c_i)$. As a basic framework, we define a minimum set of attributes as:

$$\begin{aligned} A^C &= A^C_1 \cup A^C_2 \cup A^C_3 \cup A^C_4 \\ A^C_1 &= \{n \mid n \in Names\} \\ A^C_2 &= \{c \mid c \in Complexity\} \\ A^C_3 &= \{p \mid p \in PreCond\} \\ A^C_4 &= \{e \mid e \in Effect\} \end{aligned} \tag{3}$$

Here, $Names$ refers to names of individual concept, $Complexity$ refers to computational complexity of $Behaviors$ and $Precond$ and $Effect$ refer to preconditions and effects of $Behaviors$. Sets of individual values that each element of A^C can take on, is defined in Table III.

$name_i \in$	$\Sigma^*, \Sigma = \{a, A, .., z, Z, 1, 2..9\}$
$complexity_i \in$	$\Psi^*, \Psi = \{\alpha \in \mathrm{N} \mid \alpha, n, log(n), n^\alpha, \alpha^n\}$
$precond_i \in$	$\{a\ \Re\ b \mid a, b \in C_3\ ,\ \Re \equiv \{\leq, \geq, \equiv, \neg, \|\|, \&\&\}\}$
$effect_i \in$	$\{a\ \Re\ b \mid a, b \in C_3\ ,\ \Re \equiv \{\leq, \geq, \equiv, \neg, \|\|, \&\&\}\}$

TABLE III
PREDEFINED VALUES FOR A^C

R represents set of relationships; Each relationship $r\ \langle c_p, c_q \rangle$ in R represents a binary association between concept c_p. Set of Relations for ontology is defined as following:

$$\begin{aligned} R &= R_1 \cup R_2 \cup R_3 \cup R_4 \cup R_5 \\ R_1 &= \{is\_a, is\_part\_of, is\_similar\_to\} \\ R_2 &= \{creates, accesses, manipulates\} \\ R_3 &= \{has\_input, has\_output\} \\ R_4 &= \{has\_role\} \\ R_5 &= \{\omega \mid \omega \in \Omega\} \end{aligned} \tag{4}$$

R_1 consists of set of relations required to define a hierarchal structure between $Targets$ or between $Behaviors$ and also to find similar $Targets$ or $Behaviors$. R_2 provides information about relation between $Behavior$ and its $Targets$. R_3 defines a relation over input and output of a $Behavior$. R_4 defines role of a particular $Target$ in a $Behavior$ and finally R_5 provides a subset of OWL standard operators.

Set X is a collection of axioms where each axiom in X is a constraint on the attribute of concepts or relations or is a constraint on the relationships between individuals. Set X is defined as:

$$\begin{aligned} X \quad = \quad & \{name_i \in A^C(x), \forall x \in C\} \cup \\ & \{complexity_i \in A^C(x), \forall x \in C_2\} \cup \\ & \{precond_i \in A^C(x), effect_i \in A^C(x), \forall x \in C_2\} \cup \\ & \{a\ R_1\ b \mid a, b \in\ (C_1 \cup C_2 \cup C_3)\} \cup \\ & \{a\ R_2\ b \mid a \in\ (C_2 \cup C_1), b \in\ C_3\} \cup \\ & \{a\ R_3\ b \mid a \in\ (C_2 \cup C_1), b \in\ C_3\} \\ & \{a\ R_4\ b \mid a \in\ C_3, b \in\ C_4\} \end{aligned} \tag{5}$$

Finally, set H represents a concept hierarchy derived from C as a set of superclass-subclass relations where $\langle c_p, c_q \rangle \in H$ if c_p is a superclass of c_q.

$$H = \{(a, b) |\ a\ \Re\ b, \Re \in R_1\} \tag{6}$$

Above, we defined a formal template which can be considered as a basic template with relatively minimum redundancy, we believe that this template can be extended to provide further semantic information about program. Anyway redundancy should be avoided for ease of programming and also for more efficient representation in terms of space and processing time. We also provided some constraints on values that each set can take, in order to assist in content definition. However, defining standard content for programming domain ontology is not our goal in current paper and we believe that programming domain ontology can be standardized at least for traditional application, like many other domains. A collection of current standard ontologies can be found at DAML Ontology Library [18], SchemaWeb [19], Swoogle [20] and OntoSelect [21]. Even if a standard global ontology is not used for an application, developing ontology by groups of developers who work in the same organization, can be helpful in communications between peers.

B. Mapping to Java Annotations

The formal template that we provided in previous section, is implemented as a set of three simple annotation templates in Java, called `BehaviorDescriptor`, `SubjectDescriptor` and `AgentDescriptor`. The first template plays the main role and is used to annotate methods, the second one is used to annotate fields of class and the last one is used to describe the whole application as an agent. These descriptors are shown in shown in fig. 1, fig. 2 and fig. 3.

```
@Retention(RUNTIME)
@Target ({ElementType.METHOD})
public @interface BehaviorDescriptor {
   public String   name();
   public String   complexity() default "[unassigned]";
   public String[] preCond() default "[unassigned]";
   public String[] effect() default "[unassigned]";
   public String[] Targets();
   public String[] Inputs() default "[unassigned]";
   public String[] Outputs() default "[unassigned]";
   public String   is_a() default "[unassigned]";
   public String[] is_part_of() default "[unassigned]";
   public String[] is_similar_to() default "[unassigned]";
   public String[] creates() default "[unassigned]";
   public String[] accesses() default "[unassigned]";
   public String[] manipulates()default "[unassigned]";
}
```

Fig. 1. Method Annotation Template

```
@Retention(RUNTIME)
@Target ({ElementType.FIELD})
public @interface SubjectDescriptor {
   public String   name();
   public String   is_a() default "[unassigned]";
   public String[] is_similar_to() default "[unassigned]";
   public String   has_role() default "[unassigned]";
   public String[] same_as()default "[unassigned]";
}
```

Fig. 2. Field Annotation Template

```
@Retention(RUNTIME)
@Target ({ElementType.PACKAGE})
public @interface AgentDescriptor {
   public String   name();
   public String   is_a() default "[unassigned]";
   public String   is_part_of() default "[unassigned]";
   public String[] is_similar_to() default "[unassigned]";
   public String[] same_as()default "[unassigned]";
}
```

Fig. 3. Package Annotation Template

Fig. 4 shows how these templates can be used to annotate a binary search method which is part of an accounting application and searches through salary file based on a key and returns corresponding record. Fig. 5 demonstrates annotation for `bfr` acting as buffer and shows how corresponding fields can be annotated. From fig. 4 we can see that method's syntax does not provide enough semantic information to infer its behavior. It's possible that one programmer calls binary search method `bsearch2()`, while another programmer might call it `myBinarySearch()`, those differences besides different parameter names make it impossible in syntax level to recognize that both method provide the same semantics.

As can be seen from fig. 4, these annotation can also serve as a standard format of commenting. And as already default values have been assigned to most annotation elements, it's not necessary to fill out every element, rather developers can use annotation elements according to application needs and only

```
@BehaviorDescriptor(
        name          = "Binary Search",
        complexity    = "log(n)",
        Targets       = {"Salary File","Record Key","Record"},
        Inputs        = {"Key"},
        Outputs       = {"Record"},
        is_a          = "Search",
        is_similar_to = {"Uniform Binary Search"},
        accesses      = {"Salary File","Hash Index"},
        manipulates   = "Buffer")
public Rec bsearch2(int k)
 {
     Rec r1 = new Rec();
```

Fig. 4. Behavior Annotation Sample

```
@SubjectDescriptor(
        name          = "Buffer",
        is_a          = "Record Cache")
public Rec[] bfr = new Rec[25];
```

Fig. 5. Field Annotation Sample

for required methods or fields to simplify use of annotations in practice.

C. Mapping to OWL

In a distributed system, nodes can exchange the mentioned templates in order to share knowledge about each other's functionalities. To provide a standard exchangeable format, templates should be converted into OWL format; using OWL format has three main advantages: first it's standard language for developing ontologies accessed on web, second as it's encoded in XML, it can be easily exchanged between heterogeneous machines and third because reasoning engines are already available for OWL.

To convert annotations into OWL format, annotation processing facility of Java 1.5 (apt) along with reflection is used. Using these facilities all methods with annotations type of `BehaviorDescriptor`, fields with annotation type of `SubjectDescriptor` and packages with `AgentDescriptor` annotation can be found in a program.

To generate OWL content, we use IODT which is an Eclipse toolkit for ontology-driven development providing OWL generation, visualization and also reasoning and inference through a simple API [22]. Part of generated OWL file is shown in Fig. 6 which refers to relations that hold for annotation of example behavior in fig. 4. Fig. 7 shows part of corresponding ontology diagram; in order to avoid complexity, all diagram is not shown.

Translating into OWL is almost straightforward; $Behavior$, $Subject$, $Role$ and $Agent$ are considered as top level ontology classes; $Target$, $Input$ and $Output$ are considered as subclasses of $Subject$. Every annotated element is translated into a corresponding ontology class with appropriate subclass definition using `SubClassOf` operator. Sets R_2, R_3 and R_4 defined in Eq. 4 are implemented using `ObjectProperty` relationships and sets R_1 and R_5 take davantage of built-in operations of IODT primitive OWL operators such as `SameAs` or `SubClassOf`. Axioms of set X are implemented through `Restriction` operator and are applied to

```
<owl:Class rdf:about="&Program;BinarySearch">
    <rdfs:subClassOf rdf:resource="&Program;Search"/>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="&Program;accesses"/>
            <owl:hasValue rdf:resource="&Program;salary%20File"/>
        </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="&Program;accesses"/>
            <owl:hasValue rdf:resource="&Program;hash%20Index"/>
        </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="&Program;manipulates"/>
            <owl:hasValue rdf:resource="&Program;buffer"/>
        </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="&Program;hasInput"/>
            <owl:hasValue rdf:resource="&Program;record%20Key"/>
        </owl:Restriction>
    </rdfs:subClassOf>
```

Fig. 6. Part of generated OWL File

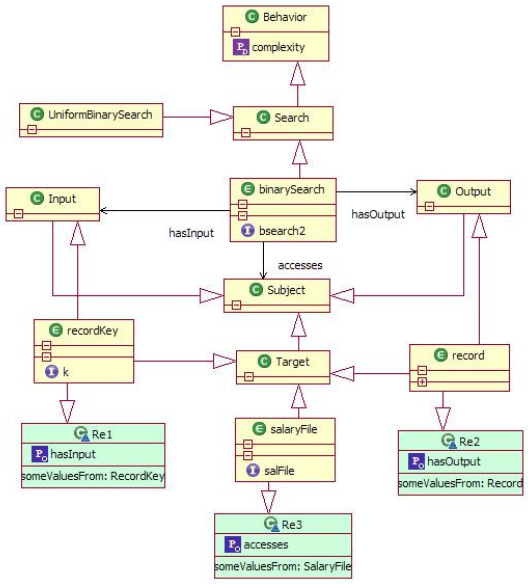

Fig. 7. Part of Ontology Diagram

appropriate classes with cardinalities of `SomeValues` or `AllValues`. In order to speedup retrievals and inferences, each annotated element's name along its corresponding OWL class name are saved as a mapping file. Mapping file also contains $Subject$ types to make it easy to detect possible errors in advice definition. To prevent ambiguity, $Input$ should be identified in the same order as in argument list to prevent type mismatch.

D. Defining Semantic Pointcuts

Using our ontology model, ontology based pointcuts can be defined relying on semantic constructs of a program instead of syntax based constructs. In order to support semantic constructs beside ordinary AOP constructs, an extension to current aspect oriented languages is necessary. As we want to

provide the capability to execute on-fly policies on remote hosts, we use JAsCo's `hook` and `connector` style for declaring Onspects. In JAsCo each aspect contains one or more hooks that define crosscutting behavior in abstract and then connectors deploy aspect onto concrete context. We try to add minimum constructs to JAsCo language with the aim of simplfying mapping from proposed extension to JAsCo language. Onspects are constructed using the following syntax described in EBNF form:

$$
\begin{array}{rcl}
pc & ::= & sig\ [\ not^{*} op\ sig\ ]^{*}\ [\ \textbf{on}\ group\ ]^{*} \\
sig & ::= & \textbf{behavior}.descpB\ rel\ value\ | \\
 & & \textbf{agent}.descpA\ rel\ value\ | \\
 & & \textbf{subject}.descpS\ rel\ value\ | \\
descpB & ::= & \textbf{complexity}\ |\ \textbf{name}\ | \\
 & & \textbf{precond}\ |\ \textbf{effect}\ | \\
 & & \textbf{hasInputs}\ |\ \textbf{hasOutputs}\ |\ \textbf{hasTargets}\ | \\
 & & \textbf{isA}\ |\ \textbf{is\_part\_of}\ | \\
 & & \textbf{isSimilarTo}\ | \\
 & & \textbf{creates}\ |\ \textbf{accesses}\ |\ \textbf{manipulates} \\
descpS & ::= & \textbf{name}\ | \\
 & & \textbf{isA}\ | \\
 & & \textbf{sameAs}\ |\ \textbf{isSimilarTo}\ | \\
 & & \textbf{hasRole} \\
descpA & ::= & \textbf{name}\ | \\
 & & \textbf{isA}\ |\ \textbf{is\_part\_of}\ | \\
 & & \textbf{sameAs}\ |\ \textbf{isSimilarTo} \\
rel & ::= & \textbf{==}\ |\ \textbf{!=} \\
op & ::= & \textbf{\&\&}\ |\ \| \\
not & ::= & \textbf{!} \\
group & ::= & node\ [\ ,\ node\ ]^{*} \\
node & ::= & \textbf{localhost}\ |\ namStr \\
asp & ::= & \texttt{class}\ id\ \{\ aspBody^{*}\ \} \\
aspBody & ::= & ar\ jStmt^{*}\ hk \\
ar & ::= & id.\ addRm\ (\ group\ ) \\
addRm & ::= & \textbf{add}\ |\ \textbf{remove} \\
hk & ::= & \texttt{hook}\ id\ \{\ hkBody\ \} \\
hkBody & ::= & jStmt^{*}\ hkCtor\ adv \\
hkCtor & ::= & id\ (\ id\ (\ \texttt{..args}\ )\ )\ \{\ ctorBody\ \} \\
ctorBody & ::= & jStmt^{*}\ trig\ (\ id\ ) \\
trig & ::= & \texttt{execution}\ |\ \texttt{call} \\
adv & ::= & loc\ (\ )\{\ adBody\ \} \\
loc & ::= & \texttt{before}\ |\ \texttt{after}\ |\ \texttt{around}\ | \\
 & & \texttt{after returning} \\
adBody & ::= & adStm\ \texttt{proceed}(arg^{*})\ adStm \\
adStm & ::= & jStmnt^{*}\ setS^{*}jStmnt^{*} \\
setS & ::= & \textbf{behavior}.descpB = value\ | \\
 & & \textbf{agent}.descpA = value\ | \\
 & & \textbf{subject}.descpS = value \\
connBody & ::= & Type\ id = \texttt{new}\ id\ (\ pc\ )\ jStmnt^{*} \\
conn & ::= & \texttt{connector}\ id\ \{\ connBody\ \}
\end{array}
$$

In suggested syntax, new keywords are identified in bold and ordinary keywords in typewriter. Basic keywords for defining pointcuts are **agent**, **subject**, **behavior** and **on**(group) which make it possible to quantify over sets of agents, behaviors, objects or nodes in network, they also can be combined by **&&**, || or **!** as logical operators. Each of these quantifiers can take on values based on their corresponding descriptors such as **complexity** for **behavior**. Using **on**(group) quantifier provides the facility to restrict pointcuts to a predefined set of agents, these groups can be defined appropriately through **add** and **remove** keywords. The rest of syntax is JAsCo's syntax for defining aspects, connectors and hooks and shows how Onspects can be integrated with them. The only extension to advice part is to have statements that assign values to **behavior**, **subject** and **agent** descriptors. An example is shown in fig. 8 and fig. 9 which triggers methods with complexity $O\left(log\left(n\right)\right)$ that access a buffer.

```
class LoggingAspect {

    loggingGroup.add("Accounting Agent","Human Resource Agent");

    hook LoggingHook {

       LoggingHook(method(..args)) {
           execution(method);
       }

       before() {
           System.out.println( thisJoinPoint.getName() +
             " accessing a buffer with complexity log(n):");
       }
   }
}
```

Fig. 8. An Example of Logging by Onspect - hook

```
connector LoggingConnector {

   LoggingHook h =
       new LoggingHook(
               (on loggingGroup &&
                behavior.complexity == "log(n)") &&
                behavior.accesses("Buffer"));
}
```

Fig. 9. An Example of Logging by Onspect - connector

E. Inference and Deployment

Using suggested syntax, a node m is able to define a pointcut p_k on any other node n_i based on its OWL descriptor o_i. To identify all the nodes that a specific pointcut applies to, OWL

inference engines like IODT reasoning facility can be used. We call each individual quantification on **behavior**, **subject** and **agent** a basic quantification operator. First, an Onspect-to-IODT converter, converts each basic quantification Q_j into its corresponding IODT query q_j, then it verifies q_j for each o_i. In order to improve performance, first **on**(group) quantification is evaluated so that only members of group will be considered for further evaluation. If current basic quantification q_j can be applied to n_i, q_j is added to set of valid quantification $\sum_v Q_{iv}$. After all quantifications have been evaluated, nonempty set of $\sum_v Q_{iv}$ are passed to Onspect converter which basically converts onspect definition into normal aspect definition based on o_i and $\sum_v Q_v i$ and mapping file. Converted pointcuts are defined as JAsCo connectors which can be simply added to a running application by placing them in application's execution path, JAsCo runtime environment finds new connectors and adds them to application. Advices are converted into a refinable `getBeforeAdvice()`, `getAroundAdvice()` or `getAfterAdvice()` method in connector. A refinable method is implemented in the connector and does not need to be implemented in initial aspect definition. These dynamic capabilities provide the facility to have dynamic policies in a system as we will point out in subsequent section. After Onspects are converted, a communicator component sends serialized connector to corresponding node.

At receiving side, an Onspect daemon listens for incoming messages, whenever it receives a new connector, it simply adds connector to application's execution path. Client only needs to do define a generic hook (fig. 10) which is a representative of future connector that would be added to system. If more than one policy at a time are predicated to be applied to system, then a predefined number of additional `getAdviceN()` methods are needed. Onspect daemon keeps track of number of connectors N and can report it to any other node who wants to define a pointcut over current node in order to append correct number N to `getAdviceN()` method.

V. On-fly policy in a multi-agent system

In this section, we provide a simple example based on an instant messaging application paradigm to demonstrate how our methodology can be used in real world situations. Instant messaging applications, better known as IM are One of the popular applications used by thousands of people all over the world. There are numerous types of IM applications implemented by different companies where the basic ability offered by all of them is real-time communication that requires an instant messaging client to connected to an instant messaging service. Nowadays, due to popularity of some messaging services, it possible for a variety of clients to connect and use these messaging service. In our context, for sake of simplicity we assume that there two messaging clients, `client A` and `client B` with different implementation that are connected to the same messaging service server.

In client-server scenarios, one of the concerns is how to update a client in case server needs to be changed and the change also affects client. For example, if server changes its login method from an unsecured method to an encrypted method, the clients also need to be updated to accommodate this change. In our example, as the clients have heterogeneous implementation, their login methods can have different names and implementations with different time complexities, making it difficult to recognize that they both provide the same functionality.

```
class  DynamicPolicyAspect{

      hook DynamicPolicyHook {

         DynamicPolicyHool(method(..args)) {
            execution(method); }

           before() {
            getBeforeAdvice();}

           around() {
            getAroundAdvice();}

           after() {
            getAfterAdvice();}

         //To be implemented in Connector
         refinable void getBeforeAdvice(){}
         refinable void getAroundAdvice(){}
         refinable void getAfterAdvice(){}
      }
   }
```

Fig. 10. Generic hook defined by client

In this case, Onspects can provide a solution for dynamic update in such a heterogeneous environment, using annotation template, as shown for `loging()` method of client 1 in fig. 11, to save space we avoid demonstrating other diagrams. Fig. 12, fig. 13 also show pointcut definition for new login policy that should be applied to those clients.

```
@BehaviorDescriptor(
        name          = "Add Entry to Database",
        complexity    = "log(n)",
        Targets       = {"Personnel Database","Record"},
        Inputs        = {"Key", "Record"},
        Outputs       = {"Return Status"},
        is_a          = "Database Query",
        accesses      = {"Personnel Database","Hash Index"},
        manipulates   = "Buffer")
public int addPerInfo(int k, Rec r)
 {
     int status = 0;
```

Fig. 11. Example: annotation of query methods

This example shows how our method can be used for dynamic on-fly updates in a distributed system.

VI. Conclusion and Future Works

In this paper, we provided an approach for defining semantic pointcuts in a distributed environment based on ideas from semantic web and ontology modeling in order not to rely on underlying syntax for triggering a crosscutting concern. Besides using JAsCo runtime facilities, we showed how a dynamic update can be applied to the clients. However as a first approach to model semantic pointcuts, current solution

```
class LoggingAspect {

   hook LoggingHook {

        LoggingHook(method(..args)) {
            execution(method);
        }

        around() {
            System.out.println(" Logging ..");
        }
    }
}
```

Fig. 12. Example: hook definition

```
connector LoggingConnector {

   LoggingHook h =
       new LoggingHook(
              (behavior.isA == "Database Query") &&
               behavior.accesses("Personnel Database"));
}
```

Fig. 13. Example: connector definition

has its own limitation and shortcoming; extension of ontology model to provide further semantic information about program and also developing standard contents for such ontology model is considered as future work; besides providing and extending a robust toolkit with an easy to use interface is part of future work.

REFERENCES

[1] T. C. Howie, J. C. Kunz, K. H. Law. Software Interoperability; Center of Integrated Facility Engineering *Stanford University* 1996.

[2] Jack C. Wileden and Alexander L. Wolf and William R. Rosenblatt and Peri L. Tarr. Specification-level interoperability *communication journal, ACM, New York* 1991, vol. 34, pages 72–87

[3] T.R. Gruber. Towards Principles for the Design of Ontologies used for Knowledge Sharing. *Journal of Human-Computer Studies,* 1993, pages 907–928.

[4] http://www.w3.org/RDF/

[5] Deborah L. McGuinness, Richard Fikes, James Hendler, and Lynn Andrea Stein. DAML+OIL: An Ontology Language for the Semantic Web. In IEEE Intelligent Systems, Vol. 17, No. 5, pages 72-80, September/October 2002

[6] http://www.w3.org/TR/owl-ref/

[7] Herre, H.; Heller, B.; Burek, P.; Hoehndorf, R.; Loebe, F. Michalek, H. General Formal Ontology (GFO): A Foundational Ontology Integrating Objects and Processes. Part I: Basic Principles. *Onto-Med Report Nr. 8. Research Group Ontologies in Medicine (Onto-Med), University of Leipzig.* Version 1.0, 2006.

[8] Matuszek, Cynthia, M. Witbrock, R. Kahlert, J. Cabral, D. Schneider, P. Shah and D. Lenat. Searching for Common Sense: Populating Cyc from the Web. *In Proceedings of the Twentieth National Conference on Artificial Intelligence,* Pittsburgh, Pennsylvania, July 2005.

[9] J. Hendler. On beyond ontology. *Keynote talk, International Semantic Web Conference,* 2003.

[10] Xiang Su1,Guojin Zhu1, Xiaoli Liu2,Wenqin Yuan Presentation of Programming Domain Knowledge with Ontology. Proceedings of the First International Conference on Semantics, Knowledge, and Grid (SKG), 2005.

[11] Miller, G. WordNet: A Lexical Database for English *Communications of the ACM* November 1995 pp.39-41

[12] BATEMAN, J. A., MAGNINI, B. and RINALDI, F. The generalized Italian, German, English upper model *Proceedings of the ECAI94 Workshop: Comparison of Implemented Ontologies. Amsterdam* 1994.

[13] K. Knight and S.K. Luk. Building a Large-Scale knowledge Base for Machine Translation. *the AAAI94, AAAI Press Proceedings of* Seattle, pages 773–778.

[14] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming *In Proc. 11th European Conference on Object-Oriented Programming,* 1997, pages 220-242

[15] W. Cazzola, J. Jezequel, A. Rashid Semantic Join Point Models: Motivations, Notions and Requirements *ACM Press* 2005.

[16] D. Suvee and W. Vanderperren. JAsCo: An aspect-oriented approach tailored for component based software development. *In AOSD Proc.,ACM Press,* 2003, pages 21-29.

[17] W. Vanderperren and D. Suvee. Optimizing JAsCo dynamic AOP through HotSwap and Jutta. *DAW: Dynamic Aspects Workshop, pages* 2004, pages 120-134.

[18] http://www.daml.org/ontologies/.

[19] SchemaWeb directory. http://www.schemaweb.info/

[20] Li Ding, Tim Finin, Anupam Joshi, Yun Peng, Rong Pan, Pavan Reddivari, "Search on the Semantic Web," Computer, vol. 38, no. 10, pp. 62-69, Oct., 2005.

[21] Paul Buitelaar, Thomas Eigner, Thierry Declerck OntoSelect: A Dynamic Ontology Library with Support for Ontology Selection *Proceedings of the Demo Session at the International Semantic Web Conference, Hiroshima, Japan* November 2004

[22] IODT, AlphaWorks, IBM http://www.alphaworks.ibm.com/tech/semanticstk.