Implementing Articulation Rules for Object Request Brokers as an Extension to Production Systems

David A. Maluf
Department of Computer Science, Stanford University
maluf@db.stanford.edu

Abstract

When combining data from distinct sources, there is a need to share meta-data and other knowledge about various source domains. Due to semantic inconsistencies, problems arise when combining knowledge across domains when the knowledge is simply merged. Also, knowledge that is irrelevant to the task of interoperation will be included, making the result unnecessarily complex. An algebra *over* domain ontologies *has been proposed to support disciplined manipulation among heterogeneous systems. This paper describes the domain ontology algebra and the use of* articulation rules *to link* declarative interfaces*. In particular, it discusses the articulation's implementation as part of CLIPS production system capable of operating over the ontologies described by the interfaces of objects registered in multiple* CORBA *servers.*

1 Introduction

Many designers, developers, and users realize that information in private and public software applications, as on the Internet, provides increasing opportunities to enhance productivity. Understanding the content of the available information requires the use of knowledge-based systems. However, effective use of knowledge to support problem solving also requires use of multiple domain knowledge. Simply taking the union of multiple knowledge sources derived from distinct domains creates several problems. One problem is due to the differing representations of knowledge obtained from different sources. Furthermore, the terms used to represent knowledge from diverse domains will have semantic inconsistencies. These inconsistencies occur because the knowledge-content will differ both in semantics and in compositional granularity. A union of multiple knowledge bases includes irrelevant knowledge and the result will be large, and disproportionally costly to process.

For intelligent access to heterogeneous information, the focus should be on the intersection of the domains, since intersection will define the required articulations. The term *articulation* refers to the linkages which join concepts across domains [9].

Linking domains using knowledge bases extends and generalizes the identification of the articulation to a set of manipulations, such as selecting, combining, extending, specializing, and modifying components from diverse common and domain-specific sources. To deal with most of these issues, an algebra over ontologies has been proposed in [31] which is intended to support disciplined manipulation of knowledge resources. The representation of vocabularies and their structure is termed an *ontology* whereas the operations that combine and partition structures in a sound and well-behaved manner are termed an *ontology algebra*. The basic algebra consists of three operations, namely intersection, union and difference.

The objective of an ontology algebra is to provide the capability for interrogating multiple sources, which are largely semantically disjoint, but where *articulations* have been established that enable interoperation. The emergent need to define articulations between data resources has been demonstrated and described in [15] and [32]. The algebra is driven by the articulation rules. The foundation of articulating a domain algebra lies in partitioning the domains. A *partition* establishes a subset of the concepts in a domain.

The idea of combining knowledge-bases [24] with declarative interfaces is complementary: declarative interfaces are primarily about specifying component syntax and distributed implementations [26], whereas research in knowledge-based systems has addressed issues of component design, component binding, and component semantics [2]. This paper describes the role of a domain ontology algebra in declarative interfaces. It demonstrates the use of an algebra that provides users and system developers with the ability to intelligently manipulate components in real time.

1.1 Related Work

An introduction to Domain Ontology Algebra is presented in references [31][32]. In these papers, the advantage of a domain algebra is described. Some suggested interoperation functionalities of the domain algebra are presented in [19]. Also, there has been a significant amount of research in the interoperable systems community. The representative literature in semantic interoperations are presented in [28][7][6]. Much of this work conforms to describing interfaces which mirrors the effort in the database community [5][1] that addresses the problem at the schema level [17][33]. There are a number of similarities with the database and knowledge-base community and the proposed work, e.g., the concept of articulation [9][15], translating heterogeneous information into a meta level model [22][30], active database [29] and associating constraints and triggers with objects [11], etc., hence the adaptation of methods from the heterogeneous database literature, mediation and integration aspects to the problem of disciplined manipulation of information sources across networks, languages and platforms.

Considering that declarative interfaces are primarily about specifying syntax and component implementation, it appears that it may be better to adapt previously devised technology to the problem of interoperation such as object algebras [16]. However, there are two main differences with the current technology: (i), the heterogeneity of the interfaces, and (ii) the autonomy of the interfaces. The first problem relates to the problem of semantic mismatch and granularity incompatibility. The second problem is that interfaces are defined by resources available at a compile time and hence may not fully cooperate in runtime. Some of these problems have been discussed with respect to the modern concept of coordinating distributed objects in declarative interfaces as in [13][23].

The problem of interoperation among heterogeneous systems is central to the area of integration, as represented in [34]. We are using parts of the research devised for the abstraction of representation [19] in the current project.

2 Interface Definition and Specification Systems

A number of object-oriented technologies have been developed to support large-scale interoperation among distributed applications [28]. However, managing large-scale interoperation of objects remains a task which requires many levels of expertise and an adherence to standards. With the advent of distributed computing, applications use interface definition standards to support interoperation in client-server and mediated architectures.

Many existing systems have strong notions of interfaces. These interfaces allow binding to services provided by their corresponding architectures if their interfaces can be described. Systems like Object Request Broker (ORB) [25] and Inter-Language Unification (ILU)[10] promote interoperability via interfaces between modules. *Module* is the generic term used to denote whatever units of program structure are desired. Modules could be parts of one process, all written in the same language; they could be parts written in different languages. A module could be a distributed system implemented by many programs on many machines. Calls across interfaces involve only as many mechanisms as necessary for the calling and called modules to interact. Modules are known by their interfaces. A module interface is specified in the system's object-oriented interface definitions language. In most systems, *Interface Definitions Languages* have been designed to facilitate such description (e.g. as Xerox ILU, OMG ORB, and OSF DCE Remote Procedure Call (RPC) [18]).

2.1 CORBA: A Brief Overview

To narrow down its scope to an implementation level, this paper relates directly to the Common Object Request Broker Architecture (CORBA) often known as *The CORBA Standard* [21]. In particular the selected CORBA implementation is the IONA implementation of CORBA.

The CORBA is a system of standards and specifications that describes how software components can interoperate across networks, languages and platforms. CORBA allows for client-server interaction between heterogeneous objects distributed over a wide-area network. CORBA makes meta information describing the objects in a system and their interfaces available so that it can access other objects. Any object connected to an Object Request Broker (ORB) can play simultaneously the role of client and server and hence Objects can initiate calls and respond to requests. ORB is the part of CORBA which facilitates client-server communication and interaction between distributed objects.

To reach object interoperability and for objects to plug and play, clients have to know exactly what they can expect from every object they might call upon for a service. In CORBA, the services that an object provides are described to interface between the object itself and the rest of the system. The objective of the interface is twofold: (i) it informs clients of the services that the objects provide as well as the access method to invoke these services and (ii) it informs the communications infrastructure of the format of the access methods.

2.2 The Interface Definition Language

CORBA *Interface Definition Language* (IDL) is defined as a language for describing the interfaces of software objects. An interface is a description of the set of possible operations that a client may request of an object [25]. An IDL interface specification contains declarations of types, exceptions and constants. IDL is independent of programming languages, and may be used to describe objects implemented using a variety of programming languages, compilers or operating systems. For instance, the IDL specification of an object `Bicycle` and class `Shop` can be described as follows:

```
interface Shop {
   readonly attribute long parent;
   void set(in long value);
   long get();
};
```

```
interface Bicycle : Shop {
   readonly attribute short size;
   readonly attribute short color;
   short getSize();
   short getColor();
};
```

The information represented by the IDL specification for any objects connected to a CORBA server is compiled and stored in the *Interface Repository* service which the server provides. The interface repository can be examined by objects on the server in order to ascertain what other objects are connected to the server and what interfaces they provide. This allows an object to request services from other objects on the server without having prior knowledge of the other objects or their interfaces.

3 Knowledge and Domains

This section discusses the implication of perceiving interface descriptions as domain knowledge. Perceiving interfaces as domain knowledge addresses several objectives, namely resolving implementation differences, interpretation and partial information.

A primary role of acquiring interfaces as domain knowledge is to resolve domain *implementation differences*. As some of the work on designing client-server applications focuses on designing software models, one can realize the different possibilities in the modeling considered in their design. For an interoperation among heterogeneous distributed systems, one must focus on relating different models. There are significant differences which make the task of relating the semantics of the models difficult. These differences are due to their interface compositions.

A second aspect of acquiring interfaces as domain knowledge is to establish correct semantic *Interpretation*. To permit objects to interoperate with other heterogeneous systems, it is not sufficient to simply merge the information on the basis of the semantics. Simply matching terms does not correspond to matching meanings. On the other hand, considering the linkages or articulation rules among these domains would be a better tradeoff in the interoperation. For each of these domain interfaces, the corresponding articulation rules are defined among the needed partitions. Interoperability can hence occur in a sound manner.

Another aspect of acquiring interfaces as domain knowledge is to handle *partial information*. Knowledge bases can handle incomplete information [12]. This problem in object oriented technology is typified in most hierarchical trees by their inability to handle partial information. For example there is no way to assert an object in the hierarchy without a reference to a root.

3.1 Terminology, Definitions and Assumptions

Partitioning: In a given application, there is an interest in certain *knowledge* rather than the complete composition of the domain. For instance, one may consider the single concept of `Wheel` from a domain `Factory` . Partitioning is equivalent to the production of simpler approximations of domain knowledge bases. When knowledge bases involve a large vocabulary, partitioning is also the process of aggregating the knowledge model to another involving smaller vocabulary and fewer constants. Often the aggregation is performed by translating the declarative knowledge predicates and grouping the vocabulary and constants into arbitrary expressions. Hence, in a `Factory` model, one can group

$$\{\mathtt{Spoke}(\mathtt{x}) \wedge \mathtt{Wheel}(\mathtt{y}) \wedge \mathtt{Frame}(\mathtt{z})\}$$

as the components of a bicycle. In the `Factory` model, the concept of *Bicycle* is not necessarily defined, hence the classes `Spoke`, `Wheel` and `Frame` are not related to each other. □

Context: Another factor of partitioning is context. Context has been proposed as a means of defining the validity of a sentence relative to a situation. Formalizing contexts [20][4] develop the notion of context which allows predicate axioms for fixed situations to be "lifted" to more dynamic contexts or when situations change. The context formalism is an extension to first-order logic in which sentences are valid within a context. To this end, the denotation of $\mathtt{ist}(c; p)$ is used to denote that a formula of a proposition p which is true in a context c. For example given a context `Factory` for the bicycle components, one can write

$$\mathtt{ist}(\mathtt{Factory}; \mathtt{Spoke}(\mathtt{x}) \wedge \mathtt{Wheel}(\mathtt{y}) \wedge \mathtt{Frame}(\mathtt{z})).$$

□

Articulation Rules: The idea of using context directly relates to the notion of partitioning in this paper; however manipulating partitions presents another scope, namely *Articulation*. The term *articulation axioms* has been established in [15] but refers to the rules that are used for translating concepts across domains. In this paper we refer to these rules as the *articulation rules*. On the other hand, the partitions are the axioms upon which rules can be established from one domain to another. A main difference between the partitions and the articulation rules is that the partitions relate to specific domains whereas the rules are maintained separately from the domain [32]. For instance, the bicycle components of a `Factory` match the concept of `Bicycle` in a toy `Shop`. Hence one can write

$$\mathtt{ist}(\mathtt{Factory}; \mathtt{Spoke}(\mathtt{x}) \wedge \mathtt{Wheel}(\mathtt{y}) \wedge \mathtt{Frame}(\mathtt{z})) \\ \Leftrightarrow \\ \mathtt{ist}(\mathtt{Shop}; \mathtt{Bicycle}(\mathtt{z})).$$

The articulation rule's specification is the part of the domain algebra which describes the linkages that handle interoperation between the independent systems. The articulations allow server-to-server interactions between heterogeneous objects distributed over a wide-area network. □

3.2 A Domain Algebra

Automatic reasoning about the interfaces requires a more formal approach to transformation and manipulation of their equivalences. Hence a set of operations are established for the needed manipulations. These operations describe the domain algebra.

The domain algebra is symbolically composed of two types, namely partitions and operation symbols. The domain partitions are atomic elements of articulation rules. The Articulation rules themselves are the atomic elements of the algebra. On the other hand, the symbols of operations, such as $\bigcap$, $\bigcup$ and $-$, stand for the algebra operations. For multiple domain ontologies, the complete operations among domains are:

Operation	Symbol	Semantics
Intersection	$\bigcap$	Create sharable expressions
Union	$\bigcup$	Create all expressions
Difference	$-$	Create not shared expressions

- *Intersection:* The intersection is the first concept of the domain algebra since it allows the algebra to bring together two domain ontologies. It is equivalent to an `AND` operator. The concept of intersection is not exactly like the predecessor algebras: the intersection is hand crafted and reflects the articulation rules.

- *Union:* The union concept allows the algebra to bring together two domain ontologies to form a new one. It is equivalent to an `OR` operator. However the algebra lacks a formal approach to eliminate redundant knowledge that is common to both. This leads to several ways of establishing the unions of multiple domains. It is convenient to think of knowledge as not being redundant if not explicitly specified by the articulation rules. Similarly to the natural join in relational databases, the domain algebra union joins interfaces when they link through shared articulation rules. The union is restricted only to the knowledge that the rules relate to. In object oriented models, inheritance and class ownership are typical relations that an articulation rule can relate to.

- *Difference:* The difference concept completes the algebra and its presence compensates for the absence of negation [31]. The difference operation retrieves the elements in domains that are `NOT` covered by another. Hence, the difference operation results in asymmetrical results and is not commutative.

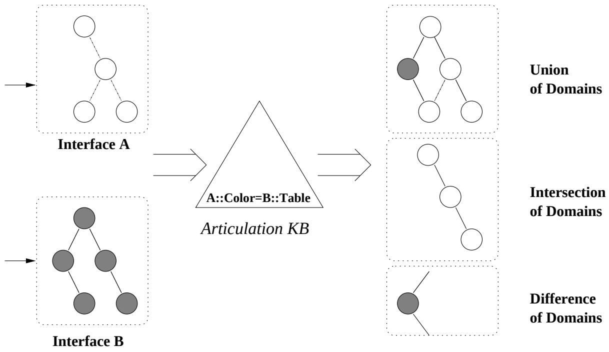

Figure 1. When interoperating among multiple loosely coupled distributed objects, articulation rules establish the links between domains and drive a domain ontology algebra.

Such an algebra can provide a basis of interrogating multiple interfaces which are semantically disjoint, but where a shared knowledge base has been established.

3.3 Partitioning Domains

The domain algebra scales partitioned domain ontologies, given some application objectives. The partitioning of a domain ontology is treated by first order logic formalism. This traces back to early work on algebras [8] which were demonstrated by first order predicate calculus. The two identified roles in translating a domain are *partitioning* and *articulation*. Partitioning separates the domain into propositions or *partitions*. Context definitions maintain the partitions within the corresponding contexts where *Articulation Rules* can be established. These articulation rules are used for linkages across domains. We consider these rules are declared and maintained by the domain experts independently from the domain sources.

Figure 1 illustrates the two functionalities of the translating of an interface: (i), the partitioning of interface repositories (Interfaces A and B) to a corresponding first order representation, and (ii), the articulation rules which are maintained in a knowledge-based system (Articulation KB).

4 Implementation

This section provides an overview of the implementation of the system capable of operating over the ontologies described by registered interfaces in CORBA interface repositories. The system is based on interfacing CORBA, since CORBA is most likely to become the industry standard for client-server and distributed systems. The implementation is more of a system integration process than a complete coding. The software component that incorporates the domain algebra and articulation rules is CLIPS. CLIPS is fully integrated in the implemented system, that is all the software components of CLIPS are used. To facilitate a quick prototype implementation, the system was designed to meet numerous objectives. One of the objectives is to build an extensible prototype – a system that supports non-traditional applications and can serve as a environment for future innovations and improvements in information integration technology.

4.1 CLIPS

CLIPS is a productive development and delivery expert system tool which provides a complete environment for the construction of rule and object-based expert systems [27]. CLIPS has been developed by the Software Technology Branch (STB), NASA/Lyndon B. Johnson Space Center. CLIPS has been designed for full integration with other languages such as C/C++ and Ada. CLIPS is the acronym for C Language Integrated Production System.

CLIPS provides a cohesive knowledge representation tool for handling a variety of knowledge with support for three different programming paradigms: rule-based, object-oriented and procedural. The rule-based, are primarily intended for heuristic knowledge based on experience and hence articulation rules. On the other hand, the object-oriented programming in CLIPS is referred to as the CLIPS Object Oriented Language (COOL). COOL allows complex systems to be modeled as modular components. The five generally accepted features of object oriented programming are class definitions, message handlers, abstraction, encapsulation, inheritance and polymorphism. COOL will be used to mirror the declarative interfaces. The rules can interact and match objects. The procedural programming capabilities provided by CLIPS are similar to capabilities found in languages such as C/C++, Pascal, Ada, and Lisp.

CLIPS includes a number of features to support the verification and validation of articulation rules, including support for modular design and partitioning of a knowledge base, static and dynamic constraint checking of slot values and function arguments, and semantic analysis of rule patterns to determine if inconsistencies could prevent a rule interaction with the objects.

CLIPS is based on the *Rete* pattern matching algorithm. The efficiency of this algorithm is based on the assumption that data changes slowly over time. This assumption matches perfectly the nature of declarative interfaces that also slowly change over time.

4.2 The Dynamic Invocation Interface Functions

The Dynamic Invocation Interface (DII) allows requests to be built up and invoked dynamically to CORBA servers. Initially, clients need to know interface-related information only at the invocation time. A DII request, like a static request, is composed of an operation name, an object reference, and a parameter list.

In the current implementation, three functions were integrated in the system to support the DII requests. The objective is that given an object reference, the object's type and all information about that type can be determined at runtime by calling functions defined by the Interface Repository. In particular, these functions can determine: the module in which the interface was defined, if any, the name of the interface, the interface's attributes and their definitions, the interface's operations and their definitions, including parameter, context and exception definitions, and the inheritance specification of the interface.

$\boxed{\textbf{GetCorbaInterface}}$ *server-name*

This function is to get all the interfaces from the Interface Repository of a CORBA server, whose name is the parameter of this function. It will get the necessary information through calls to the Interfaces Repository. Moreover, this function will then define all the interfaces locally in CLIPS. Although all the attributes and operations of the remote objects will also be put into the definition of the local objects, they are not used to store or retrieve values at this point. For instance, one may interrogate interfaces for their corresponding contents, given a `Factory` domain locates at IP address `171.64.75.95`,

```
(defcontext FACTORY)
(GetCorbaInterface "171.64.75.95")
```

and a `Shop` domain at an IP address `171.64.75.15`,

```
(defcontext SHOP)
(GetCorbaInterface "171.64.75.15")
```

Browsing the corresponding surrogates in COOL one might have

```
DOMAIN FACTORY

FACTORY
    FRAME
        Dimension
        Color-table
    SPOKE
    WHEEL
```

```
DOMAIN SHOP

SHOP
    BICYCLE
        Size
        Color
        stock-number()
    SUPPLIER
```

After *GetCorbaInterface* acquires the interfaces from the Interface Repository of the CORBA server and maps them locally, the system is ready to interoperate and create objects of the interfaces. More than one object of the same class can be created and each of them actually has its own context. The `defcontext` specification is equivalent to the module definition in CLIPS.

To keep track of all the references to CORBA objects created that were dynamically created, each local object references back the actual CORBA object to be able to retrieve attributes and invoke operations. This also maintains the independency and evolution of the servers. Hence a new attribute in every local object that corresponds to a remote CORBA object is created. This attribute is called `CorbaObjectNum`, which is a mapping between a local object and a remote object.

MakeCorbaInstance *interface-name object-name*

This function is needed for making instances of CORBA objects, and the new information is stored in the local objects. This information is called `CorbaObjectNum` and provides the mapping between CLIPS objects and CORBA object references on the server. The full interface description will be put in a string, which is also stored as an attribute. This function takes two parameters: the class name and the object name. For instance one may interrogate the class `BICYCLE` in context `SHOP`, or more precisely `SHOP::BICYCLE` as

```
(MakeCorbaInstance SHOP::BICYCLE
[Bontrager])
```

and hence a local surrogate object `[Bontrager]` is created. Each *MakeCorbaInstance* request updates the object references through `CorbaObjectNum`.

InvokeCorbaOperations *CorbaObjectNum operation return*

This function makes remote function calls to the objects residing on the CORBA server. It takes in the `CorbaObjectNum`, which provides the mapping between local CLIPS objects and remote CORBA object references on the server; the name of the operation that the user wants to invoke, and finally a multi-field value. This multi-field contains first the return type; then if the return is `OBJREF`, the return new object instance name and class name; and finally a list of type/value pairs of parameters. Types can be `OBJREF`, `SHORT`, `LONG`, `USHORT`, `ULONG`, `FLOAT`, `DOUBLE`, `BOOLEAN`, `CHAR`, or `STRING`. *InvokeCorbaOperations* function calls the necessary CORBA functions to perform the operation dynamically. For instance, one may invoke a CORBA operation

```
(InvokeCorbaOperations
   (send [Bontrager] get-CorbaObjectNum)
   "get-stock-number"
   (FLOAT)).
```

The specific operation

```
(send [Bontrager] get-CorbaObjectNum)
```

gets the index into the CORBA object array to get the object reference. `"Get-stock-number"` is the desired operation to invoke on the object. In this example, `stock-number` computes the amount of items in stock. The multi-field `(FLOAT)` stores the information of the return type of the operation and information about the parameters, in this case a float value.

4.3 Articulation Rules: Bi-directional production Rules

In expert systems, a production rule facility allows definitions of operations that are executed whenever specific events occur or certain conditions are met. In general, a production rule takes the form of:

If [*condition*] **then** [*action*]

The approach taken by the production rules community has been to provide rules that take the activity in one direction, namely Left Hand Side (LHS) where conditions and patterns are described to the Right Hand Side (RHS) where the corresponding actions are listed [3]. On the other hand, articulations are equivalence rules and a new mechanism was adopted to reference them. In general, an articulation rule takes the form of:

[*condition-action*] **equivalent** [*condition-action*]

The syntax of the writing of articulation rules is based on an extension of CLIPS production rule construction. An articulation rule is parsed in two directions and hence it becomes equivalent to two production rules. The current implementation of the articulation rule system includes three commands for defining and manipulating rules, namely *define-articulation*, *delete-articulation* and *modify-articulation*.

A production rule is activated when the condition is matched and the actions are executed. On the other hand, the actions modify the working memory according to the rule specifications. Since CLIPS is purely pattern-based, the triggering events are directly linked to the objects manipulated with the Dynamic Invocation Interface Functions, namely *GetCorbaInterface*, *MakeCorbaInstance* and *InvokeCorbaOperations*.

define-articulation *articulation-name partition* $\Leftrightarrow$ *partition*

define-articulation is a CLIPS function that creates rules using CLIPS `defrule` construct. This function effectively creates two production rules. For instance, a new rule, namely `new-rule`, can be defined as

```
(define-articulation new-rule
   (object (is-a FACTORY::FRAME)
      (Dimension ?x := (map ?x "10 01-04;..")))
   <=>
   (object (is-a SHOP::BICYCLE)
      (Size ?x := (map ?x "01-04 10;.."))))
```

The `is-a` constraint is a native CLIPS and is used for specifying class constraints. This constraint also encompasses subclasses of the matching classes. On the other hand, the `:=` symbol is a new notation and is a predicate return value constraint operator. When needed, it is possible to use the return value of the external function `map` to modify the value of a field. In the conversion of an articulation rule to production rules, the predicate return value constraint operator is translated in the *action* list.

Redefining a currently existing articulation rule causes the previous rule with the same name to be removed. This is due to the implementation of CLIPS.

delete-articulation *articulation-name*

The *delete-articulation* removes a previously defined articulation rule. Since an articulation is effectively two production rules, the deletion of an articulation rule is equivalent to the deletion of two production rules. The previously created rule `new-rule` can be deleted as

```
(delete-articulation new-rule)
```

The delete action can also be accomplished at the CLIPS level using `undefrule`.

modify-articulation *articulation-name partition* $\Leftrightarrow$ *partition*

The *modify-articulation* action allows the partitions of articulation rules to be modified. The partitions of an articulation rules can be changed after the rule has been defined. However, this requirement is not enforced since the modify action is actually a deletion followed by the new definition. CLIPS does not support production rules modification. The *modify-articulation* is meant to complete the set of defining and manipulating the articulation rules.

```
(modify-articulation new-rule
   (object (is-a FACTORY::FRAME | DEPOT::FRAME)
      (Dimension ?x))
   <=>
   (object (is-a SHOP::BICYCLE)
      (Height ?x)))
```

The ``|'' constraint is a connective constraint. Native CLIPS syntax allow three connective constraints, namely ``&'' (and), ``|'' (or) and ``~'' (not). The ``&'' constraint is satisfied if two adjoining constrains are satisfied. The ``|'' constraint is satisfied if either of the two adjoining constrained are satisfied. The ``~'' constraint is satisfied if the following constraint is not satisfied (Further details on predicate connective constraint and other CLIPS features can be found in the CLIPS 6.0 programming guide).

The two examples introduced in the *define-articulation* and *modify-articulation* cases demonstrate the mechanism of relating partitions. Since a partition may include more that one condition, conjunctive conditions can be listed with no constraint operator. The reason of listing two partitions with different contexts in the articulation rule as in the modify action examples demonstrate the multiple inheritance capability of CLIPS.

4.4 Operating over Domains

This section illustrates three examples of the domain ontology algebra relating to three operations, namely intersection, union and difference. The articulation rule examples introduced in Section 4.3 can be alternatively edited in a user interface through tables. The intersection, union and difference examples in the current section refers to the set of articulation rules described in Table 4.4. Reading Figure 4.4 from right to left, the articulation rules column displays the rule number. An interface name and class name are possible contexts for an entry. The predication occurs at the attribute level. Mapping functions are expanded to their contents.

1. *Domains intersection:* A `FACTORY` can query the domain `SHOP` and acquire the number of `BICYCLE` sorted by color and amount in stock.
2. *Domains union:* A consulting agency can verify for the domain `FACTORY` and acquire the number of `BICYCLE`, sorted by color and amount in stock, for more than one shop. This reflects a union over multiple domain intersections.
3. *Domain differences:* A `FACTORY` can query the domain `SHOP` and acquire the components or accessories that the `SHOP` relates to the `BICYCLE`, but `FACTORY` does not manufacture.

5 Status, Conclusions and Future Work

The current implementation is currently written for the 'C' Language Integrated Production System 6.0 (CLIPS) [14][27]. Since user interface functions and data access functions are separated out into other components, the domain algebra consists mainly of rules.

The system as described in this paper is fully implemented and operational. It consists approximately of 4,000 lines of C and C++ code with comments (approximately 600 C and C++ comments). The actual coding took about three man-months, but the system was carefully designed before any implementation began. Advanced expertise with CLIPS 6.0 internal kernel code was required.

The fact that the system could be implemented in a short time reflects well on the integration aspects of CLIPS. This also underlines the intent of the author of system integration as opposed to coding. One intention of the implementation is that it can be used by researchers not involved in establishing domain algebras. The wrapper that translates CORBA interfaces to COOL objects is currently made available as a public software package and can be downloaded from (`www-db.stanford.edu/~maluf/ccnp/ccnp.html`). To this stage the software has been down-loaded to over 100 researchers or users across the Internet.

Predicate		Context		Articulation Rule
Translation Function	Attribute	Class	Interface	

`01-04 yrs = 10"`
`05-08 yrs = 14"`
`09-11 yrs = 16"`
	`Size`	`BICYCLE`	`SHOP`	`0123`

`12-15 yrs = 18"`
`16-18 yrs = 20"`
`18-65 yrs = 22"`
	`Dimension`	`FRAME`	`FACTORY`	

`#FC2334 = Black`
`#FC3365 = Red`
`#FC5467 = Blue`
	`Color`	`BICYCLE`	`SHOP`	`0125`

`#FC8934 = White`
`#FC2422 = Yellow`
`#FC2545 = Grey`
	`Color-table`	`FRAME`	`FACTORY`	

Figure 2. Articulation rules can be alternatively edited in a user interface through tables. From right to left, the articulation rules column displays the rule number. An interface name and class name are possible contexts for an entry. The predication occurs at the attribute level. Mapping functions are expanded to their contents. Hence, a mapping function that translates units of `age` to `inches` can be stated as facts.

This paper describes a system of allowing multiple declarative interface interactions between heterogeneous objects distributed over a wide-area network. The objectives set in this paper are to establish the articulation needed for a domain algebra and thus sustain interoperation at an ontological level. The main objective of the use of a domain ontology algebra is to combine partitioned structures in a sound and well-behaved manner. Users and system developers can translate objects that provide comprehensive coverage of topics of interest and object usability and re-usability by different applications in changing environments. The domain algebra can bring about a shift from designing interfaces to the manipulation, enhancement, and maintenance of domain ontologies.

The behavior of large sets of articulation rules can be difficult to understand and control. To this point, no tests have been accomplished with large sets of articulation sets. However in heterogeneous systems, interaction among domains is not likely to have large sets of rules. On the other hand, the only problem foreseen in potential large sets of rules is their maintenance.

The current research is a complementary approach to the current knowledge-based systems that support disciplined manipulation of large integrated sources.

6 Acknowledgments

The author would like to thank Gio Wiederhold who carefully reviewed the draft of the document and contributed to its revised content. Contributions from Marcus Chan at Stanford University has been particularly helpful in coding parts of the implementation as a partial fulfillment of a course requirement at Stanford.

References

[1] S. Abiteboul and A. Bonner, "Objects and Views"; in Proceedings of the ACM SIGMOD International Conference on Management of Data"; Denver, Colorado, 1991.

[2] A. Borgida, "From type systems to knowledge representation: natural semantics specification for describing logics"; Intelligent and Cooperative Information Systems, 1(1):93-126, 1992.

[3] L. Brownston, R. Farrell, E. Kant and N. Martin, "Programming Expert Systems in OPS5: An introduction to Rule-based Programming"; Addison-Wesley, Reading Massachussetts, 1985.

[4] S. Buvac and R. Fikes, "A Declarative Formalization of Knowledge Translation"; Knowledge Systems Laboratory, KSL-94-59, August 1994.

[5] R. Cattell (editor), "The Object Database Standard: ODMG-93"; Morgan-Kaufmann Inc., Los Altos, California, 1994.

[6] S. Ceri and J. Widom, "Managing Semantic Heterogeneity with Production Rules and Persistent Queues"; in Proceedings of the Nineteenth International Conference on Very Large Data Bases, pages 108-119, 1993.

[7] N. Cercone, M. Morgenstern, A Sheth and W. Litwin, "Resolving Semantic Heterogeneity"; in Proceedings of the Sixth International Conference on Data Engineering, 1990.

[8] E. F. Codd, "A relational Model of Data for Large Shared Data Banks"; Communication of the ACM 13(6):377-387, 1970.

[9] C. Collet, M. Huhns and W. Shen, "Resource Integration Using a Large Knowledge Base in Carnot"; IEEE Computer, 12(24), Dec. 1991.

[10] A. Courtney, "Inter-Language Unification"; rel.1.5, Xerox PARC, 1994.

[11] N. Gehani and H. V. Jagadish, "Ode as an Active Database: Constraints and Triggers"; in Proceedings of the Seventeenth International Conference on Very Large Data Bases, pages 327-336, 1993.

[12] M. Genesereth and N. Nilsson, "Logical Foundations of Artificial Intelligence"; Morgan Kaufmann Publishers, Inc., Los Altos, California, 1987.

[13] M. Genesereth, N. Singh and M. Syed, "A Distributed and Anonymous Knowledge Sharing Approach to Software Interoperation"; Third International Conference on Information and Knowledge Management, 1994.

[14] J. Giarratano and G. Riley, "Expert Systems: Principles and Programming"; Second Edition, Boston, PWS Publishing Company, 1994.

[15] R. V. Guha. "Context: A Formalization and Some applications"; Doctoral Dissertation, Stanford University, 1991.

[16] G. M. Kuper and M. Y. Vardi, "The logical Data Model"; ACM Transcation on Database Systems, (18):379-413, 1993.

[17] W. Litwin, L. Mark and N. Roussopoulos, "Interoperability of Multiple Autonomous Databases"; ACM Computing Surveys, 22(3):267-293, 1990.

[18] H. W. Lockhart, "OSF DCE: Guide to Developing Distributed Applications"; McGraw-Hill, 1994.

[19] D. A. Maluf and G. Wiederhold, "Abstraction of Representation for Interoperation"; to appear in Tenth International Symposium on Methodologies for Intelligent Systems, Lecture Notes in Computer Science, Springer Verlag, 1997.

[20] J. McCarthy, "Notes on Formalizing Context"; In proceedings of the Thirteenth International Joint Conference on Artificial Intelligence, 1993.

[21] J. Mowbray and R. Zahavi, "The Essential CORBA"; John Wiley and Sons, 1995.

[22] National Information Standards Organization, "Information Retrieval (Z39.50): Application Service Definition and Protocol Specification (ANSI/NISO Z39.50-1995)"; NISO Press, Bethesda, MD, 1995. Accessible at `http://-lcweb.loc.gov/z3950/agency/`.

[23] N. Singh, and M. Gisi, "Coordinating Distributed Objects with Declarative Interfaces"; in Coordination Languages and Models, Lecture Notes in Computer Science, 1061, Springer, 1996, pages 368-385.

[24] R. Neches, R. Fikes, T. Finin, T. Gruber, R. Patill, T. Senator, and W. R. Swartout, "Enabling technology for knowledge sharing"; AI Magazine, 12(3):36-55, 1991.

[25] Object Management Group, "The Common Object Request Broker: Architecture and specification"; Accessible at `ftp://omg.org/pub/CORBA`, December 1993.

[26] R. Orfali, D. Harkey and J. Edwards, "The Essential Distributed Objects Survival Guide"; John Wiley and Sons, 1996.

[27] G. Riley, "CLIPS: An Expert System Building Tool"; Proceedings of the Technology 2001 Conference, San Jose, CA, December 1991.

[28] E. Sciore, M. Siegel and A. Rosenthal, "Using semantic values to facilitate interoperability among heterogeneous information systems"; ACM Transactions on Database Systems, 19(2):254-290, 1994.

[29] T. Sellis, C. Lin and L. Rashid, "Implementing Large Production Systems in a DBMS Environment: Concepts and Algorithms"; in Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, 1991.

[30] G. Wiederhold and M. Genesereth, "The Conceptual Basis for Mediation Services"; to appear in IEEE Expert, 1997.

[31] G. Wiederhold, "An Algebra for Ontology Composition"; Proceedings of 1994 Monterey Workshop on Formal Methods, U.S. Naval Postgraduate School, Monterey CA, pages 56-61, 1994.

[32] G. Wiederhold, "Interoperation, Mediation, and Ontologies"; International Symposium on Fifth Generation Computer Systems (FGCSOB94), Workshop on Heterogeneous Cooperative Knowledge-Bases, Vol 3, pages 33-48, ICOT, Tokyo, Japan, 1994.

[33] G. Wiederhold and X. Qian, "Modeling Asynchrony in Distributed Databases"; Third IEEE Computer Society Data Engineering Conference, Los Angeles, Feb. 1987.

[34] Y. Papakonstantinou, H. Garcia-Molina and J. Widom, "Object Exchange Across Heterogeneous Information Sources"; International Conference on Data Engineering, 1995.