Adding more "DL" to "IDL": towards more knowledgeable component inter-operability

Alex Borgida
Dept. of Computer Science
Rutgers University
Piscataway, NJ 08854 USA

Prem Devanbu
Dept of Computer Science
University of California
Davis, CA 95616 USA

Abstract

In an open component market place, interface description languages (IDLs) such as CORBA's provide for the consumer only a weak guarantee (concerning type signatures) that a software service will work in a particular context as anticipated. Stronger guarantees, regarding the *intended semantics* of the service, would help, especially if formalized in a language that allowed static checking of compatibility between the server and the client's service descriptions.

We propose an approach based on a family of formalisms called *description logics* (DLs), providing three examples of the use of DLs to augment IDL: (1) for the CORBA Cos Relationship service; (2) for capturing information models described using STEP Express, the ISO standard language used in the manufacturing domain (and a basis of the OMG PDM effort); and (3) constraints involving methods.

DLs, while being less powerful, do offer certain advantages over traditional formal specification techniques. First, they typically support decidable (sometimes even efficient) *reasoning algorithms*. Second, DLs excel at modeling *natural domains*, and are thus well-suited for specifying application- and domain-specific services.

1 Introduction

The software component market is here, thanks to the widely accepted standards such as CORBA and COM [27]. Now, a vendor can build a software component (such as a a financial calculator) and sell it either to end-users or to integrators. In addition, independently developed systems running on different platforms can inter-operate via published interfaces. Although the details vary with the specific standard, a key enabler of this new drive towards heterogeneous system integration is *the ability to describe interfaces in a programming language independent manner.* The core capability needed here is some *interface description language*, or IDL [14], which allows a component vendor or user to describe the interface through which two bits of software can interact. For example, a dictionary component may support the following interface:

```
interface dictionary {
    exception notFound (String key);
    exception unInitialized();
    exception badEntry(String key, String translation);
    void enter(in String key, in String translation)
            raises (badEntry);
    void setup();
    void lookup(in String key, out String translation)
            raises (notFound, unInitialized);
}
```

With this published interface, and the use of various stub and skeleton generators, and "on-the-wire" RPC protocols, it is possible for dictionaries to be implemented and (independently) client software to be built. In principle, dictionaries will become a freely replaceable and tradeable unit of software composition. Besides domain-independent components such as dictionaries and relationship servers, domain specific services are also possible, with interfaces defined specifically to support applications for manufacturing, health, education, etc. (OMG[1] has ongoing standardization efforts in some general domains). We focus more on this second family of applications.

IDL descriptions of a service resemble ordinary type descriptions, and play several roles: (i) representing agreement on the basic domain terminology/identifiers; (ii) guiding the use of the services in terms of the interface types and the exceptions; (iii) enabling automatic and static compatibility checks between the client's requirements

---

[1] Object Management Group (OMG) is the standards body controlling CORBA.

and the services offered by a provider (again, viewable as type checks). For languages with weak type systems, one might even generate run-time tests to validate arguments.

But in practice, IDLs leave many questions unanswered. For example, in the above specification: Can more than one value be associated with the same key? Can one add new words after the `setup` procedure sets up an efficient access structure, if one calls `setup` again? This is the problem of service specification. Currently, some of this additional information is provided informally, using comments in natural language. In addition to the standard problems of informal specifications (ambiguity, etc.), this leaves item (iii) above totally unsupported. It is therefore natural to consider some *formal* approach to augmenting IDL service specifications. In particular we imagine adding the following kinds of information to an IDL interface: (a) data invariants (particularly useful for database-like "integrity constraints"), (b) procedure pre- and post-conditions, (c) object behaviour models of dynamics.

In addition to expressing formally the domain semantics, the hope would be that the specifications of a client and supplier can be *matched* by some computer program (type checker/theorem prover) just as the IDL is matched.

What would the ideal formal service specification language be? Many program specification formalisms (Predicate Calculus, object oriented Z [26], Larch CORBA[15], Anna [16]) are highly expressive, but pay for this expressiveness by giving up decidable reasoning (item iii). Without effective reasoning, one cannot automatically determine if two specifications are compatible, leaving the unpleasant possibility of run time exceptions and errors.

Complete specifications may, in any case, be too onerous and difficult to obtain for real-world large products. But *benefits can be obtained even with partially characterized services*; any increment over the current IDL description provides a corresponding benefit! This suggests formalisms that are expressively limited, with decidable reasoning problems. Some languages investigated in the formal methods literature (e.g., Wright [2], Inscape [22], etc.) fall in this category. However, a third desirable feature of service specification languages is that their basic 'ontology' should match the 'object-oriented' nature of current IDLs, and they should be well-suited to describe application domains in the natural world. The aforementioned formalisms are, however, better suited for capturing mathematically clean abstractions and special notions such as concurrency.

In this paper, we propose to augment IDLs with Description Languages/Logics (DLs), whose roots are in AI knowledge representation and reasoning. DLs have several features that make them good candidates for enhancing IDLs. First, decades of research have identified a variety of DLs that admit decidable, even highly efficient reasoners. There are several refined and mature such reasoners available, and more under development. Second, the basic ontology of DLs involves notions such as "object", "concept" (unary predicate), "attribute" (binary predicate), which are entirely compatible with the basic ideas of object-centered IDLs such as CORBA's. Moreover, as detailed in [3], DLs resemble in many ways type systems, with DL judgments such as "concept subsumption" and "individual recognition" corresponding to "type subsumption" and "type inference" respectively. Therefore, DLs are natural candidates to extend the type system of IDLs to capture more details. Finally, DLs arise from the long tradition of semantic nets and frame systems [7], which are strongly biased towards capturing *real-world domain* knowledge for use in AI systems. So, while DLs are ill-suited for modeling mathematical and algorithmic knowledge (e.g., Fast fourier transforms and locking protocols), they are by design and tradition well-suited for application- and domain-specific services.

We provide three examples of the use of DLs to augment IDL component interfaces with additional semantics: (1) for the CORBA Relationship service; (2) for capturing information models described using STEP Express — the ISO 10303 language developed to describe product data in the manufacturing domain; (3) for describing constraints on procedural components. The first two examples are motivated by the OMG effort to develop product data management enablers in the manufacturing domain [19], while the last one is meant to show that DLs are useful for reasoning about meta-procedural, as well as meta-data aspects. But first, we introduce briefly Description Logics. (For further details, see, e.g., [5] and the DL web site [12].)

2 Concept Description Languages/Logics

DLs have an object-centered ontology. DL objects (which have immutable identity) may be related to other objects via *attributes* and may be grouped into *concepts*. For example, `C45` (an instance of concept `CAR`) can be related to its maker, `Gm` (an instance of

MANUFACT and US-CORP) by having `gm` be a filler of the `madeBy` attribute.[2] By default, an object may be related to zero or more other objects by an attribute – e.g., `ownedBy` for `C45` may include several entities, indicating multiple ownership. Attributes that can have at most one value are singled out, and will be called *features* here.

DLs start from primitive concept and attribute names, and combine these into *composite concepts* using **concept constructors**. Some concept constructors are the familiar boolean connectives: (**and** CAR BLACK-OBJECT) denotes the set of cars that are also black objects. Other concept constructors are more specially suited to represent conceptual models of application domains. For example, one can express cardinality constraints on attributes using constructors **at-least** and **at-most**, so that (**at-least 2** ownedBy) denotes objects with at least two owners. The values of attributes can be restricted in various ways, including by universal quantification, as in (**all** ownedBy PERSON) which denotes objects owned only by persons. For readability, **all** is replaced by **the** when dealing with features: (**the** madeBy MANUFACT)

The concept US-CORP may itself have been *defined* as the conjunction of CORPORATION and (**fills** incorporatedIn Usa), where the latter expression denotes objects that include Usa among the fillers of the attribute incorporatedIn. A useful property of DLs is the ability to nest composite descriptions, so that

```
(all madeBy (and MANUFACT
          (the postedProfit (minimum 2.5))))
```

describes objects made by MANUFACTs who posted a profit of at least 2.5 (billion dollars) — (**minimum k**) denoting the set of numbers greater or equal to k. Other concept constructors can specify relationships between (chains of) attributes:
(**and** CAR (**same-as** madeBy hasEngine.madeBy))
describes cars whose maker is the same as the maker of their engine.
One can also define composite attributes, using attribute constructors. For example, (**inverse** madeBy) would be the natural definition of the attribute makerOf, if madeBy and makerOf were inverses of each other. Other useful attribute constructors include relation composition: (**compose** p q), and transitive closure of an attribute p: (**transClosure** p)

Table 1 contains a subset of DL constructors (mostly ones used in this paper) from the ARPA KRSS standard collection [21][3].

CONCEPT TERM	TRANSLATION TO FOL $\psi(x)$ **is shown for** $\lambda x.\psi(x)$
TOP	$True$
BOTTOM	$False$
(**and** C D)	$C(x) \wedge D(x)$
(**all** p C)	$\forall z\ p(x,z) \Rightarrow C(z)$
(**at-least** n p)	$\exists z_1,...,\exists z_n$ $p(x,z_1) \wedge \ldots \wedge p(x,z_n) \wedge$ $z_i \neq z_j$
(**at-most** n p)	$\forall z_1,...,z_n, \forall z_{n+1}$ $(p(x,z_1) \wedge \ldots \wedge p(x,z_{n+1}) \Rightarrow$ $(z_1 = z_2 \vee \ldots \vee z_n = z_{n+1})$
(**same-as** p q)	$\forall z\ p(x,z) \Longleftrightarrow q(x,z)$
(**fills** p I)	$p(x,I)$
(**set** $I_1,\ldots,I_m$)	$x = I_1 \vee \ldots \vee x = I_m$
(**minimum** k)	$x \geq k$
(**maximum** k)	$x \leq k$
(**not** C)	$\neg C(x)$
(**overlap** p q)	$\exists z\ p(x,z) \wedge q(x,z)$
(**notsameas** p q)	$\exists z\ \neg(p(x,z) \Longleftrightarrow q(x,z))$

ATTRIBUTE TERM	TRANSLATION TO FOL $\psi(x,y)$ **is shown for** $\lambda x,y.\psi(x)$
IDENTITY	$x = y$
(**inverse** p)	$p(y,x)$
(**compose** p q)	$\exists z\ p(x,z) \wedge q(z,y)$
(**role-or** p q)	$p(x,y) \vee q(x,y)$
(**transClosure** p)	*transitive closure* *of binary predicate p*

Table 1: Some concept and attribute constructors and their semantics in Predicate Calculus

DLs (just like UML, for example) can express information about a domain of discourse; but DLs are also logics that provide reasoning services. A basic operation is determining if some concept C is subsumed by another, D ($C \sqsubseteq D$). For example,
(**all** ownedBy (**set** Alicia Roma)) $\sqsubseteq$
(**at-most** 2 ownedBy)

---

[2]Notational conventions: concept names are all-caps, individuals have first letter capitalized, and attributes begin with lower case letters. Keywords are bold-face

[3]The list of constructors in [21] was arrived at empirically, in efforts to express the meaning of natural language sentences and other Artificial Intelligence tasks. Subsets of its constructors have been used to develop a number of substantial ontologies, in areas such as medicine, botany, etc. — see [12].

In some situations, the question of subsumption is asked in the context of a number of other stated "declarations" of subsumption and equality (called a knowledge base/ontology). For example, (**and** A C) $\sqsubseteq$ (**and** B E) holds in the context of A $\sqsubseteq$ B and C $\doteq$ (**and** E F).

Another logical deduction is determining whether a concept or set of concepts is coherent (an incoherent concept can never have an instance — it corresponds to the predicate FALSE). And, though not relevant to this paper, there is more logic dealing with individuals, facts, and their connection to concepts.

The various DLs investigated so far have three kinds of implementations: CLASSIC, LOOM and BACK[12], the three most widely used, are implemented using a "normalize then compare" approach, which is quite different than standard theorem proving. CLASSIC has been used, among others, in industrial configuration applications, while LOOM has a wide user-base in AI. This class of DLs can also support interesting lattice-based operators on concepts such as "difference" and "least common subsumer", as well as semantic pattern-matching, and the ability to explain its reasoning (a feature often absent from theorem-proving techniques based on model-checking or tableaux). A second family of DLs, typified by CRACK[12], is implemented with tableaux-like refutation theorem proving techniques. Recently, several powerful decidable DLs have been identified, that are closely related to Converse Propositional Dynamic Logic [10]; hence, these can rely on extensions of the theorem provers for the latter.

A key focus of KR&R, and specially DL research has been the trade-off between expressiveness of a KR language and the feasibility of reasoning with it. Specifically, there are detailed results about the computational complexity of reasoning with various subsets of constructors (and restrictions of them). Of interest are subsets for which reasoning is *decidable* (a variety of subsets of constructors from KRSS are known to result in logics in which reasoning is undecidable), or even computable in polynomial time. Through judicious choice of concept constructors (versions of the first 11 in the first table on page 3) standard CLASSIC [6] has a complete subsumption algorithm that is $o(n^3)$. The tableaux techniques appear to function quite fast in practice, though having worst-case exponential-time complete subsumption. Finally, LOOM, which offers a much wider set of constructors, has an in-complete reasoner since subsumption is undecidable for several different subsets of its language.

Since the full KRSS DL is still known to express only a very limited subset of Predicate Calculus (the "3 variable subset"), for practical purposes systems such as CLASSIC and LOOM offer "escape hatches": ways of describing concepts using either procedures (**test-concepts**) or full first order logic ((**satisfies** ψ)). Their reasoners tend to treat these as black boxes.

Let us turn now to applications of DLs in describing the semantics of interfaces.

3 The CORBA Relationship service: `Cos-Relationship`

The OMG common object services (COS) framework defines a *relationship service* (called `Cos-Relationship` [1]) which is used to model entities and relationships that arise in application software, in the style of Entity-Relationship models. The service is widely referenced because it is compatible with other CORBA services such as naming, life-cycle, transaction, persistence, security, etc.; it also includes useful operations for traversing the graph of a relationship.

The service is formally specified in pure CORBA IDL; but the limitations of IDL force many modeling decisions (e.g., about cardinalities) to remain implicit in the implementation of the clients and servers. In this section, we briefly introduce the service using the example of a public school, and illustrate how DL-based modeling can explicate such decisions, and (through reasoning) statically determine whether there are incompatibilities that could rear up during actual inter-operation.

Consider a system dealing with public schools. Informally, a *school* is located in a *district*, and *persons* reside in a district. *Students* can be enrolled in a school. To enroll in school, the student must reside in the same district that the public school is located in. To model this one starts with classes for PERSON, SCHOOL, DISTRICT, and then establishes relationships according to a general approach illustrated in Figure 1. An object participates in relationships, such as **enrollment**, via a *role* (e.g., the PERSON object is connected with a role object **student** to the **enrollment** binary relationship.) A fragment of the corresponding IDL would be

```
module PS {
    interface PERSON { . . . }
    interface SCHOOL { . . . }
    interface DISTRICT { . . . }
    interface student :
```
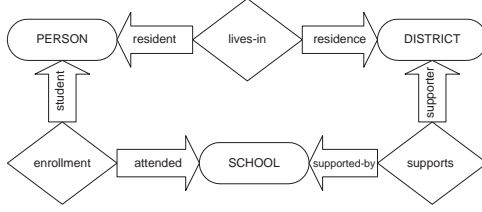
Figure 1: Example binary relationships

```
        CosRelationship::Role
interface attended :
        CosRelationship::Role
interface enrollment
        CosRelationship::Relationship;
    etc.
```

Let us examine in detail the process of enrolling a child in a school, using the above service. First, the `createRole` method of an implementation of a `CosRelationship::RoleFactory` is invoked to create a `attended` role for the `SCHOOL` object. The semantics of the domain require that there must be between 50 and 200 students attending the school. If the latter upper bound is violated, the `MaxCardinality-Exceeded` exception (which is predefined for `Cos-Relationship`) is signalled. Second, a `student` role is created for the child, who is required to be an instance of `PERSON` having age under 21. Moreover, the student must live in the same district that supports the school. These translate into checks made by the software – for example, that the `student` role has an associated `PERSON` object who `lives-in` the same `DISTRICT` that `supports` the `SCHOOL`. The violation of these constraints may be signalled by raising additional kinds of exceptions. Finally, the `createRelationship` method of a `CosRelation-ship::RelationshipFactory` object is invoked to create the `enrollment` relationship between the above two roles.

The problem is that while IDL can represent the existence of the exceptions, it cannot specify the precise conditions under which these exceptions are raised. The constraints noted above can however be expressed in DL (CLASSIC) by declaring concept identifiers for interfaces (`SCHOOL, DISTRICT, PERSON`), attribute and feature identifiers for the roles (`resident, residence, student,...`, and then adding restrictions on the concepts:

PERSON ⊑ (**the residence DISTRICT**)

CHILD $\doteq$ (**and** PERSON
         (**the age** (**maximum** 21)))

STUDENT ⊑ (**and** CHILD
    (**same-as residence attended.supporter**))

SCHOOL ⊑ (**and**
    (**at-most** 200 **attended**)
    (**all student** CHILD)
    (**the supporter** DISTRICT))

These concept descriptions can be explicitly entered into the interface description as annotations (perhaps in comments, as in [23]). These assertions amount to an explicit representation of domain knowledge that would otherwise be implicit in the implementation, or would appear in comments. For example, the interim OMG PDM Enabler proposal [20] uses stylized comments to capture cardinality constraints.

Consider two software systems desirous of communicating over a school module, such as the one above. As long as their IDL formulation is the same, they can communicate. In addition, each IDL formulation will now include a model of the constraints on the various relationships in the form of DL concepts. In this case, using DL inferences, the two models can be statically compared for consistency using the subsumption interface.

For example, assume that two components define an interface `COMPLIANT-SCHOOL` differently. Assuming the following additional semantics

SCHOOL ⊑ (**the class-size** Integer))

supporter $\doteq$ (**inverse** supportedBy)[4].

RICH-DISTRICT $\doteq$ (**and** DISTRICT
 (**the** supportedBy
   (**the class-size** (**maximum** 30))))

suppose that one district defines `COMPLIANT-SCHOOL` as: (**and** SCHOOL (**the class-size** (**maximum** 30))) while another describes it as:

(**and** SCHOOL (**the supporter** RICH-DISTRICT))

CLASSIC will quickly infer that these two definitions are in fact equivalent, and declare the components compatible; updates to the various relationships can be carried out between these components, without

---

[4]If **inverse** is not
available, and the attributes are features, this can be captured
as (**same-as** supporter.supportedBy IDENTITY).

fear of `CosRelationship` cardinality or other exceptions.

We note that to deal with non-binary relationships, recursive constraints, and to detect certain kinds of modeling errors (models that cannot be satisfied in finite databases, only infinite ones), it may be advantageous to use a more expressive DL for modeling [8].

4 Capturing EXPRESS in DL

The Object Management Group (OMG) standardization efforts for manufacturing software[19] have strong ties to the established ISO Standard for the Exchange of Product Model Data (STEP), which helps manufacturers share product data. A key element of STEP is the language EXPRESS, used to present "information models" for different kinds of applications (e.g., CAD designs, electrical products). Hardwick et al [13] provide an introduction to STEP and EXPRESS. They also describe a mapping from EXPRESS features to IDL, which can be used to support distributed information sharing. EXPRESS [25] is a powerful object-centered data specification language. It is more expressive than most object-oriented modeling languages and supports several special features. We show a mapping from EXPRESS to DLs to demonstrate the latter's suitability for application-specific domains. In addition, our mapping provides a formal semantics for part of EXPRESS which is not yet available elsewhere. We use part of an example from [25] to illustrate EXPRESS, and describe its translation into DL. This allows partial reasoning with EXPRESS.

The example concerns the registration of cars, and is summarized informally thus:
Cars are objects, which have (single-valued) attributes such as model (which is a car model object), serial number, ownedBy. The car is made by a manufacturer, who is said to make the model as well, and the manufacturer of the car plus its serial number distinguish this car from all other cars.
A car also can have up to 100 options, and a history of transfers. Each transfer records from which owner to which owner the item was transfered, and the date when this occurred. Each car's transfers are organized in a sequence, so that the previous owner is the seller, and the dates are in non-decreasing order.

The following is the EXPRESS syntax capturing this information.

```
1. entity CAR;
2.  model        : CAR-MODEL;
4.  options       : set [0:100] of CAR-OPTIONS;
```

```
5.  serialNo      : INTEGER;
6.  ownedBy       : OWNER;
7.  history       : list of TRANSFER;
  derive
8  madeBy         : MANUFACT := model.madeBy
  where
9.  datesOK(self.history)   // checking history
10  exchangeOK(self.history) // for correct ordering
  unique
11     (madeBy, serialNo)
end  entity;


entity TRANSFER;
   item         : CAR;
   from         : OWNER;
   to           : OWNER;
   when         : DATE;
 end entity;
```

The only non-self-explanatory material are lines 9 and 10, which illustrate EXPRESS's "escape mechanism": for material that cannot be captured using the dictions of EXPRESS, there is a full-fledged programming language for writing functions.

To begin with, here is the limited model that could be captured in OMG IDL:

```
interface CAR {
  attribute CAR-MODEL model;
  attribute set<CAR-OPTIONS> options;
  attribute OWNER ownedBy;
  attribute seq<TRANSFER> history;
  attribute MANUFACT  madeBy;
}
```

The DL translation of the EXPRESS model is more accomodating. First, we use DL primitive attribute names for EXPRESS attributes that have aggregation domains, like SET or LIST (`options` and `history` in this case); the rest become features, having at most one value. Also, all entity names are translated into primitive concept names. Types (not shown) are translated to corresponding concept definitions. For example, the EXPRESS **type** GENDERS = **enumeration of** ('masc,'fem); becomes
`GENDERS` $\doteq$ (set 'masc 'fem).

The basic attribute domain and cardinality constraints (lines 2 to 7) are immediately translated to DL as subsumption constraints on concept `CAR`:

```
CAR ⊑ (and
   (the  model CAR-MODEL) (at-least  1 model)
   (all  options CAR-OPTIONS)
   (at-most  100 CAR-OPTIONS)
   (the  serialNo INTEGER) (at-least  1 serialNo)
   (the  ownedBy OWNER) (at-least  1 ownedBy)
```

```
(all history TRANSFER)
(all madeBy MANUFACT) (at-least 1 madeBy) )
```
In addition, EXPRESS has **derive** and **where** constraints. **derive** provides a way to 'compute' the value of the attribute. Many of the simple **derives** constraints can be expressed directly in DL: *e.g.,* line 8 adds (**same-as madeBy model.madeBy**) to the above. (The EXPRESS line ⟨ madeBy: MANUF := Ford ⟩ would add (**fills madeBy Ford**).) The remaining kinds of derivations are encoded in the catch-all
(**test-concept** xpr-derive madeBy "≺expression≻")
As in EXPRESS these cannot usually be reasoned with statically, but can be checked dynamically in a particular model.

The EXPRESS **where** conditions are used to state complex boolean constraints. In fact, line 8 could have also been stated as ⟨ **where** madeBy :=: model.madeBy ⟩. Again, some **where** constraints can then be captured by DL constructs; and concept constructors **and** , **or** , etc. can be used to combine them to get complex boolean expressions. Lines 9 and 10 become DL terms like (**test-concept** datesOK history). The final constraint, **unique**, is unfortunately very common in computer-based information systems and man-made worlds (in the natural world, there are no globally unique external identifier). Recently, CLASSIC has been extended with a concept constructor that can capture this as (**key** [madeBy,serialNo] CAR). In general, though, this would have to be represented as (**test-concept** key [madeBy,serialNo] CAR).

A simple Prolog program can translate abstract syntax trees for EXPRESS declarations (using a slightly different grammar that generates the EXPRESS language) to DL terms; after some simple post processing, these produce a set of concept and role specifications for CLASSIC, for example. In fact, for the above example CLASSIC was sufficient, and hence reasoning is very efficient, as mentioned earlier.

Finally, suppose we would like to represent explicitly the list of transfers in the history (using an attribute **next** say), so that we could enforce constraints such as the fact that the buyer in a transfer needs to be the seller in the next one – (**same-as** dest next.src). This sounds quite manageable if we could define a concept corresponding to a linked list cell (CELL ⊑ (**the** next CELL)). However, this combination of language constructors — "recursive" constraints and **same-as** on feature chains

— can be used to encode the well-known undecidable "word" problem, and hence is not permitted in CLASSIC. There are DLs, such as CIQ [10], which can handle recursive concept declarations, and even fixed-point definitions. To ensure decidability of reasoning, these exclude the constructor **same-as**. There is no way out of this dilemma in general. The benefit provided by DL research is the a priori knowledge of what combinations are undecidable, so that one is aware that complete reasoning cannot be expected in principle.

We remark that the recent trend to extensible DL reasoners [4], allows specialized, possibly incomplete reasoners to be customized for applications. For example, probably the only important reasoning with keys is like (**key** [madeBy,serialNo,α) CAR) ⊑ (**key** [madeBy,serialNo) VEHICLE) for any attribute list α. The CLASSIC 2.3 release allows this kind of inference to be added to the basic implementation so that (**test-concept** key ≺FeatureList≻ ≺ConceptId≻) will behave appropriately.

The facility of DLs to state definitions is a major missing feature of EXPRESS, which can only declare primitive concepts, with necessary conditions, and then add sufficient conditions as assertions. So for example, the CLASSIC definition of ADULT $\doteq$ (**and** PERSON (**the** age (**minimum** 22))) would require the use of procedures or queries (which are equivalent to FOPC) in EXPRESS:
entity ADULT
  age/PERSON : INTEGER
where
  necessaryCond: (self.age $\geq$ 22);
  suffCond: **query**(p < * PERSON | p.age $\geq$ 22)
    ⊆ **query**(c < * CHILD | true);
end entity;

5 Modeling dynamics

The above examples illustrate the modeling of static aspects of a domain; we now turn to dynamic aspects. In AI, actions (verbs) are modeled using a "case frame" approach where the participants of the activity are linked to it via attributes. So, for example, one can have the primitive concept DELIVER, with features object, src, dest and when:
```
DELIVER ⊑
  (the object CAR) (the src MANUFACT)
  (the dest DEALER) (the time DATE)
```
corresponding to the IDL specification
```
 interface CAR{
 ...
   deliver(in CAR object, in MANUFACT src,
```

in DEALER dest, in DATE time)

...

}

However, to capture additional constraints on the participants (for example, that the `object` and `src` must be located in the same place at the beginning) one can no longer use IDL — the only trace of it would be an exception NOT-SAME-PLACE raised by the method. We can however also model the initial and final conditions associated with a method using DLs as follows: (i) For each method (e.g., `deliver`, `sell`) associated with an interface (e.g., `CAR`), declare a corresponding feature in the DL concept:

CAR $\sqsubseteq$ (**the** deliver DELIVER)

(ii) Express the preconditions as descriptions relating the parameter features, as in

;; object delivered must be co-located with the source

CAR **and** (same-as deliver.object.location deliver.src.location)

Preconditions can also involve the attributes of the concept itself (recall that in OO methods, the data members of a class can be accessed by a method); for example,

;; the car's maker delivers it to the dealer

CAR **and** (same-as madeBy deliver.src)

Similarly, it is the owner of the car who is supposed to sell it, so we need precondition

CAR **and** (same-as ownedBy sell.src)

The specification can also indicate that the latter condition is checked, by associating an exception with its violation; in contrast, if the condition does not raise an exception, then it is an assumption – a proof obligation.

Final conditions are treated similarly. So for example, the effect of selling the car is to give it a new owner

;; the destination of sell is the new owner

CAR **and** (same-as ownedBy sell.dest)

Some initial and final conditions may be too hard to express in DLs so they can be either stated informally, as comments, or as argument formulas to **satisfies** .

Finally, DLs can also represent dynamic sequencing constraints on the methods of a service. It is not uncommon in the OO world to provide some description of the valid sequences of operations on the objects of a class. Rather than using state charts, or other formalisms which require introducing new identifiers (for state names), we use regular-expression based notation for valid operation sequences. In the `CAR` example, as-

sume methods to deliver it to the dealer, to sell a car, and to destroy it. The usual valid sequence would be captured by the regular expression *deliver* (*sell*)* *destroy* . Interestingly, the example in [25] actually states that "after destruction, earlier transfers (sales) can still be recorded". This means that the server might specify the regular expression *deliver* (*sell*)* *destroy* (*sell*)*, with suitable restrictions on the `sell` method's time.

A client desiring the first service could use the second one, but not vice versa. How can we verify this? As it happens, one large family of decidable DLs is based on a close correspondence with "converse propositional dynamic logic" [10]. The basis of this correspondence is model-theoretic: objects are states (hence concepts are state-predicates), attributes are state transitions (hence programs). Therefore, all we have to do is view the above regular expressions as complex attribute-expressions in a concept definition, e.g.,

(**at-least** 1 (**compose** deliver

(**compose** (**transClosure** sell) destroy)))

We can then use the usual subsumption engine. Moreover, the correspondence to PDL allows much more complex "programs" to be written, describing complex local conditions under which methods can be invoked.

Alternatively, one could use the plan-based DL CLASP [11], which can be added onto CLASSIC using the extensibility features [4]. Plans are constructed through sequence, alternation and looping from actions, with the additional benefit of checking the consistency of sequencing: preconditions of actions cannot conflict with postconditions of their predecessors.

6 Discussion

We have argued that IDLs, such as that of OMG, provide only limited type information as a basis of the agreement about the services provided or desired, and this limitation can easily lead to run-time errors (at best) or worse, unintended effects that remain undiscovered.

One solution is to increase the power of the service "type description language", which in the limit can be a powerful program specification formalism, such as Larch CORBA [15] or ADL [24] — the most closely related previous work to ours.

However, we also seek a decidable, or even tractable,

formalism to allow compile-time "type checking". We have argued that Description Logics are type-like formalisms (concepts are unary predicates after all), where subsumption can be used in ways similar to type signature matching. In fact, DLs seem to form a nice middle ground between signature matching [28] and specification matching [29]. DLs were designed for modeling application domains in the natural world. Many such domains arise from the standardization efforts of the OMG subcommittees in manufacturing, medicine, etc., where we have observed the need for capturing additional service semantics being met by the stylized use of comments. We note that an IDL service specification already provides us with a valuable commodity: a consensus on *primitive terms* (concept and attribute names for DLs), from which composite terms and assertions can be built using the DL constructors. This contrasts with the harder task of merging multiple data sources (e.g., heterogeneous database integration) — a problem to which DLs are also being applied assiduously (e.g. [9, 18]).

In specific terms, we have showed how DL concepts can be used to capture *some* of the (i) data invariants related to interface attributes for services that are data/relationship centered; (ii) pre and post conditions needed to describe methods; (iii) conditions under which exceptions will be raised; (iv) aspects of event dynamics. We have argued that DLs are well suited especially for the first task by using actual services (`CosRelationship`) or languages like EXPRESS, which have been originally developed to specify services in special domains. Also, the recently exploited connection between PDL and some expressive DLs makes it possible to make strong statements about dynamics *while staying in the same framework*!

In addition to checking for exact matching of the above 4 kinds of information between the client and server, the formal assertions and DL reasoning allow us to perform additional tasks:

Compatibility testing of the specifications:
Even if the client's requirements do not match the provider's specifications exactly, if the client's needs are stronger, (e.g., an extra initial condition), then the client can be warned, and it might even be possible for the language translation to automatically add "wrapper code" for run-time checks. This would require us to be given methods to access *primitive* concept and attribute instances, but would generate code for complex descriptions.
(Local) consistency checking:

Formal interface specifications can be checked for self-consistency. For example, the initial conditions of a method should not be inconsistent with the data invariants for its interface class.
More thorough treatment of exceptions:
By associating IDL exceptions with formal assertions we obtain, to begin with, some validation benefits if we discover assertions that did not have IDL exceptions, and vice-versa. For example, the `takesSpouse` $CosR$ relationship may specify that (a) it is anti-symmetric, (b) one person's spouses cannot overlap with another person's, (c) a person has at most one spouse, etc. For each of these constraints, a different exception would be specified.
Variability in services provided:
Interestingly, if the checking of some assertions is left to be generated *automatically* from the augmented IDL (as mentioned earlier), with the choice of which assertions are to be checked left to the client, but the code being generated/used at the server site, we get a much more flexible kind of inter-operation. For example, the `takesSpouse` relationship server could then be used in a polyandrous society, by omitting constraint (c).

To conclude, we repeat that DLs provably cannot capture all the possible things one might want to say about a service – but then neither can anything else that is decidable. And for certain services (e.g., ones involving numeric computing, or concurrency) there are surely far better alternatives to DLs. That is why we have emphasized the kind of services that we see DLs being particularly well suited to deal with. Moreover, the technology of extensible DL reasoners holds the promise of being able to custom tailor DLs that are specially suited for certain application domains.

REFERENCES

[1] http://www.omg.org/corba/sectrans.htm

[2] R. Allen, D. Garlan, "A formal basis for architectural connection", *ACM TOSEM*, July 1997.

[3] A. Borgida, "From type systems to knowledge representation: natural semantics specifications for description logics," *Int. J. of Intelligent and Cooperative Information Systems 1*(1), 1992.

[4] A. Borgida, "Towards the systematic development of terminological reasoners: CLASP reconstructed", *Proc. KR'92*,

[5] A. Borgida, "Description Logics in Data Management", *IEEE Trans. on Knowledge and Data Engineering*, vol.7, no.5, October 1995, pp. 671–682.

[6] A.Borgida, R. J. Brachman, D. L. McGuinness, and L. A. Resnick "CLASSIC: a structural data model for objects," *Proc. SIGMOD'89* pp. 59–67.

[7] R. Brachman, H. Levesque, Eds, "Readings in Knowledge Representation", *Morgan Kaufman*, 1985.

[8] D. Calvanese, M. Lenzerini, and D. Nardi, "A unified framework for class-based representation formalisms", *Proc. KR'94*, pp.109–120.

[9] D. Calvanese, G. De Giacomo, M. Lenzerini, D. Nardi, R. Rosati. "Description logic framework for information integration" *Proc. KR'98*, pp. 2-13.

[10] G. De Giacomo and M. Lenzerini. "A uniform framework for concept definitions in description logics." *Journal of Artificial Intelligence Research , 6*, pp.87-110, 1997. (More details in ftp://www.dis.uniroma1.it/pub/degiacomo/phd-thesis95.ps.gz)

[11] P. Devanbu and D. Litman, "CLASP - a plan representation and classification scheme", *Artificial Intelligence*, 1996.

[12] *Description Logics*, http://www.ida.liu.se/labs/ iislab/people/patla/DL/

[13] Hardwick, M., D.L. Spooner, T. Rando, K.C. Morris. "Sharing Manufacturing Information in Virtual Enterprises", *CACM 39(2)*, February 1996.

[14] D. A. Lamb, "IDL: Sharing Intermediate Representations" *ACM TOPLAS*, July 1987

[15] G. Leavens and Y. Cheon, "Extending CORBA IDL to Specify Behavior with Larch", Iowa State University, *CS-TR-93-20*, 1993.

[16] D. C. Luckham, F. W. von Henke. "An overview of Anna, a specification language for Ada" *IEEE Software* March 1985.

[17] R.M. MacGregor, "A deductive pattern matcher", in *Proc. AAAI'87*, pp.403–408.

[18] E. Mena, V. Kashyap, A. Sheth, A. Illarramendi: "OBSERVER: An Approach for Query Processing in Global Information Systems based on Interoperation across Pre-existing Ontologies", *Proc. CoopIS 1996*, IEEE-CS Press, pp.14-25.

[19] OMG TC, "Product Data Management Enabler RFP", August 1996. http://www.omg.org/library/ schedule/PDM_Enabler_RFP.htm

[20] OMG TC, "PDM Enablers – Joint proposal to OMG in response to [19]", ftp://ftp.omg.org/-pub/docs/mfg/98-01-01.pdf

[21] Patel-Schneider, P. and W. Swartout, eds., "Description Logic Knowledge Representation system specification fromthe KRSS Group of the ARPA Knowledge Sharing Effort", http://www-db.research.bell-labs.com/user/pfps/papers/krss-spec.ps, November 1993.

[22] D. Perry, "The INSCAPE environment" *Proceedings, ICSE 1989.*

[23] D. Rosenblum "A Practical Approach to Programming with Assertions" *IEEE TSE*, Jan. 1995.

[24] S. Sankar, R. Hayes "Specifying and Testing Software Components using ADL" Sunlabs Technical Report, Sun Microsystems, Inc., TR-94-23 (April 1994) http://www.sunlabs.com/-techrep/1994/smli_tr-94-23.pdf, 1994.

[25] Schenk, D.A., and Wilson, P.R. *Informationb Modeling the EXPRESS Way*, Oxford University Press , 1994

[26] S. Stepney, R. Barden and D. Cooper (eds), *Object Orientation in Z*, (Workshops in Computing Series), Springer Verlag, 1992.

[27] C. Szyperski, "Component Software", *Addison-Wesley*, 1994.

[28] Zaremski, A., and Wing, J., "Signature matching: A tool for using software libraries." *ACM TOSEM*, 1995.

[29] Zaremski, A., and Wing, J., "Specification matching of software components," *ACM TOSEM 6(4)*, 1997.