In: Proc. OOPSLA'96

A Functional Layer for Description Logics: Knowledge Representation Meets Object-Oriented Programming

Ralf Möller

University of Hamburg, Computer Science Department
Vogt-Kölln-Str. 30, D-22527 Hamburg, Germany
moeller@informatik.uni-hamburg.de

Abstract: The paper motivates the facilities provided by Description Logics in an object-oriented programming scenario. It presents a unification approach of Description Logics and object-oriented programming that allows both views to be conveniently used for different subproblems in a modern software-engineering environment. The main thesis of this paper is that in order to use Description Logics in practical applications, a seamless integration with object-oriented system development methodologies must be realized.

1. Introduction

Object-oriented modeling is now a standard technique for application development (see the discussions in [8] and [9]). In addition to object-oriented modeling, Description Logics (DL) have been proven to be a useful formalism for modeling knowledge in a specific domain (see e.g. [33] for a commercial example application). A DL provides a relational rather than an object-oriented way of modeling ([4], [32]). The main advantages of the DL perspective are that (i) inferences about domain objects can be formally modeled and (ii) A DL can deal with incomplete "conceptual" information about objects.

Although the logical semantics of DL modeling constructs is a big plus, to ensure decidability of the subsumption problem, the expressiveness of the logical language must be limited ([23], [32]). The consequence is that in a real application without toy problems, currently not all aspects can be formally modeled. This means that programming is still necessary. The question is, how this should be treated. One approach is to emphasize the logical model and to avoid talking about the additional procedural programming parts. This can be called the Dr. Jekyll and Mr. Hyde approach. In my opinion, from a software engineering perspective, it is more advantageous to make the best of the two worlds. When a subproblem of a complex application can be adequately solved with Description Logics, this formalism should be used. However, the DL solution must be integrated into the whole system development approach which, at the current state of the art, uses object-oriented modeling. Object-oriented modeling (or programming) is intimately related with incremental development and reuse. When a DL is to be integrated into such an environment, the "OO-mechanisms" should also be applicable to the DL part. In this paper, it is shown how both representation and modeling approaches, logical modeling and object-oriented programming, can complement each other. The paper presents a unification approach that allows both views to be conveniently used for different subproblems in a software-engineering environment.

Object-oriented programming is only a very vague term for a variety of approaches to software structuring and system development ([8], [9]). In this paper, the Common Lisp perspective will be used [30]. The document is written for readers with an object-oriented programming background. It assumes basic knowledge about CLOS ([14], [15], [24]). A short introduction to the main ideas behind Description Logics is given with several examples. "Description Logic" is also a very general name for different theories and practical systems [32]. One implementation

of a DL is the CLASSIC system ([1], [3], [25], [29]) which will be used for the examples presented in this paper.[1]

The integration of CLASSIC and CLOS requires that generic functions and methods can be written for CLASSIC objects (called individuals for historical reasons). The procedural parts of an application should be able to use CLASSIC individuals just like CLOS instances. The "services" of an object are accessed only by the use of functions. This means that the relational part of CLASSIC should be hidden behind a functional layer because, from a software engineering point of view, dealing with individuals and relations can be quite cumbersome for at least two reasons: First of all, from a functional programming point of view it is irrelevant whether a result of a function call is defined by merely looking up a set of instances in a relation table or by actually computing instances with complex algorithms and auxiliary data structures.[2] Second, using generic accessor functions for retrieving the objects that are directly set into relation to a specific individual allows a lot of error checking code to be automatically generated. A unified interface for accessing the services that an object provides hides many representation details which are irrelevant from a more abstract perspective.

The advantages of a logical representation system like CLASSIC, i.e. its ability to deduce implicit information, will be presented with a small object-oriented software-engineering scenario. Using three common programming problems (also called design patterns [10]) it will be shown that a Description Logic provides solutions *within* the language itself, i.e. there is no need to find specific implementation tricks or programming conventions to adequately implement the design patterns. We will discuss the following patterns:

- Define an interface for creating objects but let subclasses specify which classes should actually be used to instantiate objects and the corresponding subobjects (design pattern "Factory Method" in [10]). This problem is also called the "Make isn't Generic Problem" (the term has been coined by Kiczales [17]).
- Automatically change (or adapt) the class of an object to a certain subclass when the object is associated with another object.
- Deal with conceptual assertions for individual objects at runtime (e.g. the "type" of associated objects, or the number of objects associated with an object using a certain relation).

2. The Use of DL in an OOP Scenario

Let us assume, that in a press office, information about an unknown ship is gathered because of an SOS signal being received. The kind of the ship determines the importance of the SOS signal. For instance, for the next newspaper issue, there will be last minute changes to the banner headline whenever a *passenger* ship is found to be in distress. Thus, initially, there is a ship instance and step by step additional information about this ship will be added. The information that a certain ship is a passenger ship might not be given explicitly. The information source for the press office might not know that "passenger ship", which serves as a trigger for additional processes here, is an important piece of information.

2.1. Example Domain: Modeling Ships

The world model for this example domain consists of a few concepts and roles which will be presented using the KRSS syntax for CLASSIC [26]. Concepts can be compared to classes and can be described with superconcepts and restrictions for role fillers (necessary and sufficient conditions). A primitive concept (declared with `define-primitive-concept`) is only partially described with necessary conditions. If also sufficient conditions are given, a "defined" concept is described (declared with `define-concept`).

1. CLASSIC can be licensed from AT&T Bell Labs. The code for the extensions presented in this paper is available from the author [19].

2. Sometimes so-called "virtual relations" (or virtual slots) are used whose fillers are computed by functions that access the fillers of other relations (or slots). However, conceptual assertions about virtual relations are not supported (see also Section 2.6) and therefore, these relations are somewhat asymmetric.

The complete set of concept definitions is called TBox. The root of the concept hierarchy is `classic-thing` (compare this to `standard-object` in CLOS). Our initial TBox is defined as follows:

```
(define-primitive-concept person classic-thing)
(define-primitive-concept cargo-object
  classic-thing)
(define-primitive-concept container
  cargo-object)
(define-primitive-concept passenger
  (and person cargo-object))
```

Roles to relate individuals must be declared as well (for the term "role", the term "relation" is often used as a synonym). In a similar way as concepts, roles can be hierarchically related as well (`nil` is used for "no parent").[3] The inverse of a role can be declared as well.

```
(define-primitive-role has-ship nil)
(define-primitive-concept captain person)

(define-primitive-role has-cargo-object nil)
(define-primitive-role has-captain nil
   :inverse has-ship)
(define-primitive-role has-position nil)

(define-primitive-concept pos classic-thing)
(define-primitive-role has-x-coordinate nil)
(define-primitive-role has-y-coordinate nil)

(define-primitive-concept ship
  (and (all has-cargo-object cargo-object)
       (at-least 1 has-position)
       (at-most 1 has-position)
       (all has-position pos)
       (at-most 1 has-captain)
       (all has-captain captain)))
```

The concept `ship` will be explained in detail. The declaration specifies that all cargo objects (relation `has-cargo-object`) must be instances of `cargo-object`, i.e., in DL terminology, they must be subsumed by `cargo-object`. All fillers of the role `has-captain` must be captains. For each `ship`, there will be exactly one filler for the role `has-position` (which must be subsumed by the concept `pos`) and at most one filler for `has-captain` (which must be subsumed by `captain`).

3. A subrole (or subrelation) identifies a subset of the relation tuples defined by the superrole.

From a data representation point of view, this small domain model could also be represented with CLOS (or any other object-oriented representation mechanism). So, what are the advantages of Description Logics? The answer is that a DL can be used to *deduce implicit information*. In the following sections, we will discuss three simple examples which indicate why deductive capabilities of Description Logics go far beyond object-oriented programming. Before we discuss the examples, I would like give a brief overview on the semantics of description logics.

2.2. Semantics of Description Logics

Concept terms are mapped onto logical formulas. For instance, (`all r C`) is mapped onto the formula:

$$\lambda x \cdot \forall (y \in D \cdot\ r(x, y) \Rightarrow C(y))$$

where D is the domain (universe of discourse), r is a relation and C is a concept term. Cardinality restrictions like (`at-least 1 r`) are mapped onto formulas with existantial quantifiers.

$$\lambda x \cdot \exists (y \in D \cdot\ r(x, y))$$

The `and` concept term constructor is mapped onto a logical conjunction etc. Statements with (`define-primitive-concept C term`) and (`define-concept C term`) are mapped onto the following logical axioms, respectively.

$$\forall (x \in D \cdot C(x) \Rightarrow term(x))$$

$$\forall (x \in D \cdot C(x) \Leftrightarrow term(x))$$

The first declaration defines only necessary conditions for a concept C (implication) while the second defines also sufficient conditions (bi-implication). A concept C is a superconcept D iff $C(x) => D(x)$. The superconcept-subconcept relationship is also called the *subsumption* relationship. A detailed description of the semantics of the CLASSIC representation language is beyond the scope of this paper (see [2]).

The logical axioms provide the basis for logical deductions to derive implicit information. A DL reasoner can be interpreted as a sound and complete inference engine for a subset of First-Order Predicate Logic. Thus, in contrast to the semantics of programming languages, the semantics of the DL language

also defines what has to be computed by a DL reasoner given a set of input formulas (i.e. there is no need to define an operational semantics). For instance, the set of concept and relation definitions (TBox) can be checked for consistency.

Furthermore, implicit subsumption relations between defined concepts are automatically detected. For example, the TBox could be extended by the following concept definitions:

```
(define-concept ship-with-captain
  (and ship
       (at-least 1 has-captain)))

(define-concept ship-with-cargo
  (and ship
       (at-least 1 has-cargo-object)
       (at-least 1 has-captain)))
```

Though not explicitly stated, it is evident that `ship-with-cargo` is also a subconcept of `ship-with-captain` and the TBox reasoner adds `ship-with-captain` as a superconcept of `ship-with-cargo`. However, in general, finding these implicit subsumption relations in the TBox can be very difficult (see [23], [32]).

2.3. Dynamic Classification

Computing the subsumption relation and doing consistency checking in the TBox is one kind of service a DL reasoner offers. Consistency checking is also done for individuals in the ABox. For example, only one `captain` can be set into relation to a `ship` via the `has-captain` relation. More interesting than consistency checking is the "dynamic classification service" for ABox individuals. The ABox incrementally derives the concepts of an individual from its relations to other individuals (forward reasoning). This is possible for "defined" concepts which are described with necessary *and sufficient* conditions (see above). Concrete individuals are created in the ABox with the following statements:

```
(define-distinct-individual s1)
(state (instance s1 ship))
(define-distinct-individual c1)
(state (instance c1 captain))
```

Instances can be related to each other by an additional ABox statement:

```
(state (related s1 c1 has-captain))
```

After this statement has been asserted, the ABox of CLASSIC will deduce that `s1` is not only a `ship` but must also be a `ship-with-captain`. Note that the additional concept `ship-with-captain` is not explicitly stated for `s1`. It is automatically derived from the relation of `s1` to other individuals (in this case `c1`). If `c1` were not known to be a `captain`, this would also be inferred as a by-product (see the restriction `(all has-captain captain)` in the definition of `ship`).

Why is this an important service which an object system definitely should provide? The answer is that the first design pattern of the introduction (Factory Method) can be elegantly implemented using the DL facilities. In other words, it solves the "Make isn't Generic Problem". This problem will become apparent a larger software development scenario. I will explain it using our ship domain.

2.4. Dynamic Classification as a Solution to the "Make Isn't Generic Problem"

At the heart of object-oriented system design is the notion of reuse. Therefore, let us assume a large software package for handling ships has been developed by a software company Seasoft. A shipping firm Ocean-Trade will buy the product if the local software development team of Ocean-Trade can adapt the software library to the needs of the firm. Obviously, Seasoft will not give away the source code and Ocean-Trade gets an object-oriented software library with classes and methods in binary format only.

Surprisingly, object-oriented development alone does not guarantee enough flexibility even for simple programming problems. A simple example shows where standard OO techniques fail or where they are more complicated than necessary. The software system of Seasoft contains classes (or, as we will see, even better: concepts) that are used for modeling ships, positions, persons, passengers, captains etc. However, the

need to represent a `ship-with-captain` has not been anticipated (`ship-with-captain` serves as a placeholder for something to be incrementally added by Ocean-Trade here). Like many large software systems, the Seasoft library generates objects internally. For instance, for planning purposes, additional ships might be created by the Seasoft logistics module which consists of a set of ship classes. We assume that a set of generic functions have been defined to solve a logistics planning problem. For the predefined Seasoft `ship` classes, methods are written as usual. The output of the planning system might be a report to propose further investments etc.

Let us return to Ocean-Trade now. In our scenario, the planning module must be adapted to match the needs of Ocean-Trade. In the spirit of object-oriented programming, subclasses of predefined classes will be created and methods might be "overwritten". Due to an object-oriented system design, there should be no problems. For example, for ships with captains, Ocean-Trade would like to add additional functionality to the Seasoft planning module. Let us assume, the Seasoft system uses a certain generic function for which a method will be added that dispatches on `ship-with-captain` (the subconcept of `ship` defined above). This method is more specific than the predefined method dispatching on `ship`. In this case, the problem is that *instances are created in the body of "old" methods written by Seasoft*. Seasoft methods create instances of `ship` rather than instances of `ship-with-captain`. The instantiation functions called in the inherited methods do not use the intended subclass `ship-with-captain`, i.e. the code is not easily extensible (hence the term "make is not generic"). In the example discussed above, the class of a new instance depends on some other instances being related to it (or, as Kiczales puts it, which give rise to the creation of the instance [17]).

A first attempt to find a solution for Seasoft would be to insert a function which is used compute the class of internally created instances. This solution has been proposed by Gamma et al. as the "Factory Method" design pattern [10]. Seasoft must anticipate the desire for extensions here and should add an additional generic function to their protocol which is used to dynamically compute the classes used by their planning module. The idea might be to let customers (i.e. Ocean-Trade) write more specific methods for this function. But, on which object(s) should this function dispatch? It must be an object that Ocean-Trade will have to provide in order to avoid an infinite recursion of the problem!

The software system of Seasoft will become clumsy if too many of these unwieldy "extension hooks" are provided.[4] This is where Description Logics come into play. If CLASSIC were used for the development of the Seasoft system, the concept `ship-with-captain` could have been defined by the Ocean-Trade software team. Instead of using a primitive concept, Ocean-Trade can define a concept with sufficient conditions just as in the example presented above. With the extended generic functions presented in this paper, it is possible for Seasoft to write methods that dispatch on CLASSIC individuals. Ocean-Trade can "specialize" a certain generic function provided by Seasoft. Now, when other methods defined by Seasoft create an individual which is subsumed by `ship` and which is related to a `captain`, this individual will be *automatically* classified as a `ship-with-captain`. There is no need to use a procedure to compute the class at instance creation time. Whenever generic functions are applied to this `ship-with-captain` in old methods defined by Seasoft, the intended behavior (which is defined by Ocean-Trade) is automatically available.

The small example presented above is no "constructed problem". Kiczales presents several other problems of the same category [17] and proposes a solution that uses an extension to instance creation. In his solution, the creation of an instance (e.g. a ship) can depend on other instance (e.g. a captain). The main idea is to let the software customer (Ocean-Trade) define dependencies in terms of class paths (called "traces"). By "traces" new superclasses can be "inserted" at instance creation time. Thus, Seasoft provides default classes and Ocean-Trade can force

4. Other approaches like "delegation" also fail in this scenario.

subclasses to be used. However, Seasoft must anticipate a possible "trace" and has to use a more generic make function for instance creation. The solution of Kiczales only works when the relation of a ship to a captain is *known at instance creation time* (i.e. the captain must be an initialization argument to the ship). But, what happens when the captain is not known at instance creation time? With Description Logics, this is no problem either. Reclassification will occur even if the ship individual is created first and a captain will be added afterwards! It is even possible to define a concept `ship-without-captain`.

```
(define-concept ship-without-captain
  (and ship
       (at-most 0 has-captain)))
```

Specific methods can be written for this concept, too. In this example, the "dependency" solution of Kiczales with "traces" at instance creation time also fails because, in this case, there is no "dependency" at all.

The declarative way of specifying defined concepts can be of great importance in modern software engineering scenarios. It can help to open up implementations (see the notions of open implementations [16] or glass-boxes [28]) and supports encapsulation at the same time (no hooks for class computations). The examples presented in this section indicate that reasoning about concepts with necessary and sufficient conditions is more powerful than object-oriented programming techniques provided by CLOS (and other OO systems). ABox reasoning also goes beyond the capabilities of standard frame systems which only support consistency checking as an inference service. But DL reasoning offers even more.

2.5. Dynamic Classification as a Solution to the Concept-by-Relation Problem

Continuing our ship example, I would like to add new concepts and relations.

```
(define-primitive-concept ship-in-shipyard
  ship)

(define-primitive-role has-ship-in-repair-dock
  nil)

(define-primitive-concept shipyard
  (all has-ship-in-repair-dock
       ship-in-shipyard))
```

For ships which are subsumed `ship-in-shipyard`, changing the position might be forbidden. This might be achieved by the definition of specific methods.

In some circumstances a concept should only be a superconcept of an instance when the instance is set into relation to another instance. Thus, a `ship` should only be subsumed by this concept when it is set into relation to a `shipyard`. However, it would be very inconvenient to directly create an individual of `ship-in-shipyard` or to explicitly change the class of the individual. A similar situation occurs when "persons" become "customers" when they are set into relation to a "bank". This dynamic classification is no problem when DL concepts are used. The ship example is continued with the following assertions.

```
(define-distinct-individual s1)
(state (instance s1 ship))
(define-distinct-individual yard1)
(state (instance yard1 shipyard))
(state (related yard1 s1
                has-ship-in-repair-dock))
```

Even though `s1` is created as a `ship`, the dynamic reclassification mechanism of the ABox forces `s1` to be also an instance of `ship-in-shipyard` just because `s1` is set into relation `has-ship-in-repair-dock` to `yard1`. The classification is dynamic because `s1` will no longer be a `ship-in-shipyard` when the `related` statement is retracted.

It should be noted that in dynamic OOP languages the class of an instance might be changed (e.g. in CLOS: `(change-class a-ship 'ship-with-captain)`). However, the new class itself has to be determined in beforehand (possibly with procedural code) and will not be determined automatically by considering relations to other objects. The so-called Concept-by-Relation problem is not adequately solved in object-oriented programming languages. Another important topic is the treatment of generalized conceptual assertions for individuals.

2.6. Beyond OOP: Generalized Conceptual Assertions for Individuals

Object classification can also depend on conceptual assertions about role fillers (concept restrictions, cardinality restrictions). This will be illustrated with the example from the press office. The individual `s1` known from above serves as a representative for the ship in distress in this example. Let us further assume, a few other concepts for ships are declared in the TBox. Concepts for passenger ships as well as container ships will be defined with sufficient conditions.

```
(define-concept passenger-ship
  (and ship
       (all has-cargo-object passenger)))

(define-concept container-ship
  (and ship
       (all has-cargo-object container)))
```

Let us further assume, that in our example domain, incomplete information about `s1` is announced to the press office (e.g. by an incoming fax). As time passes it turns out that there are passengers on the ship. Although there might also be some containers on the ship, this is unimportant for the press office and, as an additional assumption, it is asserted that *all cargo objects are passengers*. In formal terms, this is expressed as follows.

```
(state (instance s1
          (all has-cargo-object passenger)))
```

Together with the given TBox, implicit information can be inferred by ABox reasoning. The individual `s1` is reclassified (or "subclassified") as a `passenger-ship`. In this case, the inference step is justified by the *conceptual* assertion `(all has-cargo-object passenger)`. Together with the concept `ship` the conditions for `passenger-ship` are satisfied (see the concept definition for `passenger-ship`). In contrast to the `ship-in-shipyard` example in the previous section, no concrete passenger instances are known. Conceptual information about the cargo objects suffices to deduce that `s1` is a `passenger-ship`. Knowing the fact that `s1` is a `passenger-ship` might trigger processes which change the newspaper headline etc.

Reasoning about concepts is required when concrete objects are not known, i.e. when the information available about objects is incomplete. Dealing with incomplete, conceptual information *at runtime* is currently not supported by OOP languages. In strongly typed Functional Programming languages (e.g. Haskell [27]) and strongly typed OOP languages (see e.g. [6]), *at compile-time* a type inference mechanism may be used but, at runtime, inferences like those presented above are not supported.

There are other examples where the concept of an individual is important. For example, in a graphical user interface, the drawing function for a ship might depend on the ship's concepts. User interface programming is one of the best examples for the application of object-oriented programming techniques. For rapid user interface development however, an existing UIMS *must be reused* (for Common Lisp, this can be CLIM [7], [22]). UIMSs like CLIM provide powerful programming abstractions which are modeled with the object-oriented representation techniques of CLOS (e.g. different classes for gadgets and output streams etc.). There is no way to rebuild these software libraries with CLASSIC or any other DL in a reasonable time. So what can be done? Copying information associated with a DL object into a CLOS object which is used for UI part of an application is inadequate as well. Unfortunately, managing multiple "copies" of the same object is a direct contradiction to the principles of object-oriented programming. Thus, for rapid application development, object-oriented programming techniques must be made available to CLASSIC individuals. The next chapter discusses an approach that demonstrates how this can be achieved with an extension to CLASSIC that uses CLOS-like generic functions to access information about an individual.

3. Integrating OOP and DL

The main features of object-oriented programming are:

- defining the structure of instances (in terms of slots or "instance variables" and inheritance),

- defining object behavior (partial function definitions with methods and inheritance),
- realizing encapsulation by hiding the structural layer behind the behavioral layer.

CLOS separates the structure definition (slots) from the behavioral definition (generic functions and methods). Encapsulation is realized by the module mechanism of Common Lisp (packages). The separation of these software-engineering dimensions has many advantages. One of the advantages is that it is possible to define a behavioral layer for an existing structural layer. In our case, the internal object structure can be handled by CLASSIC and will be hidden behind the behavioral layer of generic accessor functions as is usual in CLOS.

3.1. Accessors: A Functional Interface to a Knowledge Base

CLASSIC itself provides a relational interface for retrieving and adding role fillers. Assume, there exists a ship `s1` with captain `c1`. Given the ship, the captain can be retrieved: `(cl-fillers @s1 @has-captain)`.[5] In our example, this will a return a set of fillers (`@c1`). For all ships, at most one captain will be returned (see the definition of the concept `ship`). However, the result will always be a set (actually a list) and the function `first` must be applied to get the list element itself. To hide the repeating access to the first element, an additional function will have to be written. Furthermore, in some circumstances, it will be considered as an error if the filler is not known. Unfortunately, additional code must be written to check this. If `nil` (the empty set) is returned, an error is likely to occur in subsequent function calls when a `captain` individual is expected. Again, code must be written to avoid this. Instead of writing this code manually, a more general mechanism is advantageous. The way to access individuals should be declaratively defined using generic functions and corresponding low-level code for methods should be automatically generated. The declaration form `define-accessors` has been introduced to specify the access to individuals in that way.

```
(define-accessors <concept-name>
  (<role-name> <accessor-name>
               [ :single-value-p <boolean> ]
               [ :error-if-null <boolean> ] )
  ...)
```

For each role description mentioned after the concept name, a reader method and a setf writer method for the generic function `<accessor-name>` is generated. If they do not already exist, corresponding generic functions are automatically generated.[6]

The first role option `:single-value-p` specifies whether a single value or a set of values should be returned by the role reader function. The other option `:error-if-null` is used to insert code for error checking to avoid an empty set to be returned.[7]

In our example, the following definitions are used.

```
(define-accessors captain
  (has-ship captains-ship :single-value-p t))

(define-accessors ship
  (has-cargo-object ship-cargo-objects)
  (has-position ship-position
                :single-value-p t
                :error-if-null t)
  (has-captain ship-captain
               :single-value-p t
               :error-if-null t))

(define-accessors pos
  (has-x-coordinate position-x
                    :single-value-p t
                    :error-if-null t)
  (has-y-coordinate position-y
                    :single-value-p t
                    :error-if-null t))
```

5. `@` is read as "individual quote" and is used to get the individual with name `s1` as an object.

6. The explicit declaration of a generic accessor function is necessary, for instance, when a special method combination that differs from the standard method combination is to be used [30].

7. Default values for role options could be inferred from concept definitions. For instance, in a `define-accessors` definition for a concept `c`, "`:single-value-p t`" might be automatically inserted as a role option for a role `r` if the concept `c` was subsumed by `(at-most 1 r)`. On the other hand, if `c` was subsumed by `(at-least 1 r)` the option "`:error-if-null t`" could be used. In my opinion it is better to define the options explicitly (for readability reasons). Furthermore, in some cases it is convenient to use a set in subsequent function calls even if its cardinality is one.

One of the main advantages of dispatched access to information about individuals via methods is error checking. When, by accident, `position-x` is applied to a `person`, the condition `no-applicable-method` will be signalled. An error is indicated right at the wrong function call. If CLASSIC's retrieval functions for role fillers were used, possibly `nil` would be returned. Though *not being inconsistent*, a person might never intended to be (directly) related to a number via the role `has-x-coordinate`. It would be very inconvenient to restrict this on the logical side (e.g. by forcing an inconsistency). Even if this were done, CLASSIC would happily return the empty set (`nil`) as the set of role fillers for `has-x-coordinate`. From a logical point of view, such a query for a role filler is well defined. On the procedural side, the return value `nil` might cause an error. If ever, the error might be detected in subsequent function calls which do not expect `nil` to be a valid return value. With generic functions and methods, the missing information (on the logical side) that persons are not intended to be related to numbers via `has-x-coordinate` can be added.

The functional access layer has been deliberately separated from the concept definitions (TBox). These mechanisms can be considered as completely independent layers. As in CLOS, for accessing information of an object, `define-accessors` defines methods for generic functions. Additional methods might be written by the programmer (e.g. around methods or after and before methods).

CLASSIC individuals require another dispatch mechanism which is realized by extended generic functions which will be explained in the next section.

3.2. Generic Functions and Methods

The extended generic functions presented in this paper can dispatch on CLASSIC concepts or CLOS classes or both. An example for a generic function that indirectly accesses information stored for an object is given below. The form `define-generic-function` is used to define a generic function with dispatching extended to CLASSIC individuals.

```
(define-generic-function ship-position-xy
  ((ship :classic)))

(define-method ship-position-xy ((ind ship))
  (let ((pos (ship-position ind)))
    (values (position-x pos)
            (position-y pos))))
```

When the function `ship-position-xy` is applied, the internal role structure used for representing position information is transparent. `(ship-position-xy s1)` just returns two values. Hiding the internal role structure is quite important because the role structure might be subject to change. Generic functions realize important encapsulation principles. The relational interface of CLASSIC is weak in this respect.

The argument list of `define-generic-function` indicates which arguments expect CLASSIC dispatch and which arguments use standard CLOS dispatch.

```
(define-generic-function <function-name>
  (<dispatched-argument-description> ...
   [ <other-argument> ... ] )
  [ <option> ...] )
```

A description for a dispatching argument is a list consisting of an argument name and a dispatch indicator (either `:clos` or `:classic`). Just as `defgeneric` from CLOS, `define-generic-function` also supports ordinary arguments without specializer (called `other-arguments`). The options for `define-generic-function` are the same as for `defgeneric`. Note that method combinations are also supported.[8]

Methods can be defined with the form `define-method`.

```
(define-method <function-name> [ <qualifier> ]
    ( <dispatched-argument> ...
     [ <other-argument> ... ] )
  ... )
```

The syntax of a dispatched argument in a method parameter list is identical to the syntax of arguments

8. In CLOS, a generic function is automatically generated when the first method definition is evaluated and the generic function is not yet known. This is currently not supported by `define-method`.

for `defmethod` of CLOS. The `<qualifier>` indicates the kind of method combination. In addition to names for CLOS classes, CLASSIC concept names can be used as specializers.

In our ship domain, we will use a generic function for printing information about ships on a certain output stream. As usual, multimethods should be defined that dispatch on the kind of ship and on the kind of output stream (e.g. `textual-output-stream`, `graphical-map-output-stream`). The example emphasizes the requirement that both, CLASSIC and CLOS arguments must be dealt with during method dispatch. Nobody would try to reinvent the wheel and represent streams with CLASSIC concepts.

```
(define-generic-function print-ship-info
    ((ship :classic)
     (stream :clos))
  (:documentation "Demonstration function for ~
                   method dispatch."))
```

The generic function `print-ship-info` might be used by the Seasoft software. Methods are defined for `ship` and its subconcepts and a CLOS class `textual-output-stream`.

```
(define-method print-ship-info
    ((ship ship)
     (stream textual-output-stream))
  (format stream "~%Ship ~S." ship)
  ...)

(define-method print-ship-info :after
    ((ship container-ship)
     (stream textual-output-stream))
  (format stream
          "~%~S is even a CONTAINER ship."
          ship)
  ...)

(define-method print-ship-info :after
    ((ship passenger-ship)
     (stream textual-output-stream))
  (format stream
          "~%~S is even a PASSENGER ship."
          ship)
  ...)

(print-ship-info s1 *text-output-stream*)
```

When the function `print-ship-info` is applied to a ship individual and a stream instance, the composition of the effective method depends on the ship's concepts and the stream's class. Initially, for `s1` only the first method will be applied because `s1` is classified as a `ship`.

The main advantage of CLASSIC is the feature of dynamic object classification by ABox reasoning. Let us assume, that in our example domain, the following is asserted (see above).

```
(state (instance s1
          (all has-cargo-object passenger)))
```

By ABox reasoning `s1` is classified as a `passenger-ship`. A call to `print-ship-info` will result in different output after the assertion has been added. An additional `:after` method will be added to the effective method for subconcepts of `ship` (standard method combination).

Methods can also be written for generic accessor functions that are created with `define-accessors`. For instance, the position of a captain can be directly associated with the position of his ship.

```
(define-method position-x ((c captain))
  (let ((ship (captains-ship c)))
    (if ship
        (position-x ship)
      nil)))

(define-method position-y ((c captain))
  (let ((ship (captains-ship c)))
    (if ship
        (position-y ship)
      nil)))
```

In this context, the liberal use of dynamic OOP systems like CLOS should be emphasized. There is no need to define a method within the scope of a class definition. Why not writing a separate method for `print-ship-info` that dispatches on captains?

```
(define-method print-ship-info ((c captain)
                                (s stream))
  (let ((ship (captains-ship c)))
    (unless (null ship)
      (print-ship-info ship s))))
```

These methods for existing generic functions might be written by another programmer who has no access to the source code for the definition of `captain`. In languages like C++ a subclass of `captain` has to be created to add such a method. Additional subclasses

even increase the need for design patterns like “Factory Method”. In CLOS this can be avoided because a method is associated with a generic function and not with a class.[9] The service of printing information about a ship can very well be reached via a captain object.

A common pitfall is illustrated with the following example. In order to avoid the test whether the call to `captains-ship` returns `nil` in the method presented above, it might be a good idea to define an additional concept.

```
(define-concept captain-with-ship
  (and captain
       (at-least 1 has-ship)
       (at-most 1 has-ship)))
```

According to the definition of `has-captain` as the inverse of `has-ship`, the statement

```
(state (related s1 c1 has-captain))
```

implies that `(related c1 s1 has-ship)` also holds. Therefore, the ABox also concludes that `c1` is a `captain-with-ship` because there is at least one ship (`s1`) set into relation to `c1`.

A “simplified” method might be defined for the generic function `print-ship-info`.

```
(define-method print-ship-info
    ((c captain-with-ship)
     (s stream))
  (print-ship-info (captains-ship c) s)))
```

The accessor function `captains-ship` can also be used for an instance of `captain-with-ship` because this is a subconcept of `ship`. It seems to be that the test whether `captains-ship` returns `nil` can be omitted because there should be exactly one filler for the role `has-ship`. Everything runs fine as long as `captain-with-ship` is inferred via ABox reasoning about the role fillers of `has-ship` (and the inverse `has-captain`). But, what happens when a `captain-with-ship` is created directly?

```
(define-distinct-individual c2)
(state (instance c2 captain-with-ship))
```

The instance `c2` is an instance of `captain-with-ship` by definition. However, there is no filler *known* for `has-ship`. The logical semantics of `(at-least 1 has-ship)` merely says that it is *consistent* that a `captain-with-ship` is associated with a `ship` and iff there is a filler for `has-ship`, the captain will also be a `captain-with-ship`. There is *no need to actually create a filler* when an instance is a `captain-with-ship` by definition. Thus, a runtime error (no applicable method) is likely to occur when the method defined above is executed because there is no `print-ship-info` method for `nil` (and a stream).

3.3. Individual Creation and Initialization

Creating individuals using the primitives supplied by CLASSIC (or KRSS) is somewhat crude. From a software engineering point of view, a protocol for individual initialization is needed. For individual creation, a function `create-individual` with parameters `(concept-name &optional (ind-name (gensym)) &rest initargs)` has been supplied.

An example would be

```
(setf c2 (create-individual 'captain-with-ship
                            'c2)).
```

When an individual is created with `create-individual`, the generic function `initialize-individual` is automatically called. A method for this function can be defined as follows (compare this to `initialize-instance` from CLOS).

```
(define-method initialize-individual :after
      ((ind captain-with-ship) &rest initargs)
  (unless (captains-ship ind)
    (setf (captains-ship ind)
          (create-individual ‘captain))))
```

Initialization arguments can also be given to `create-individual`. The list of “initargs” is a sequence of role names and corresponding sets of initial fillers.[10]

```
(setf s1 (create-individual 'ship 's1
           'has-captain c1))
```

9. Associating a method with a specific class is impossible when multiargument dispatch is supported (“multimethods”).

10. The set of initial fillers for a role is represented by a list. If a non-list is used, a singleton list is automatically created.

For `ship` another initialization method might be defined.

```
(define-method initialize-individual :after
      ((ind ship) &rest initargs)
  (let ((captain (ship-captain ind)))
    (setf (captains-ship captain) ind)
    (unless (ship-position ind
                           :error-if-null nil)
      (setf (ship-position ind)
        (create-individual
           'pos (gensym "POS")
           'has-x-coordinate 0
           'has-y-coordinate 0)))))
```

A CLASSIC individual can be used just like a CLOS object. For instance, `(ship-position s1)` returns a single value: a position individual. As the body of the method indicates, defaults for options given in the `define-accessors` definition can be overridden for a specific accessor call.

3.4. A Functional View on Conceptual ABox Assertions?

So far we have seen that a functional layer with extended generic functions, methods and automatically generated accessors can be smoothly integrated with a relational DL system. In this context, it is interesting to consider whether conceptual assertions (see the example in Section 2.6) can also be stated from a functional point of view.

In our example, the ship `s1` will be used as an argument to `ship-cargo-objects`. The individual `s1` is subsumed by `ship`. When applied to `s1`, the function `ship-cargo-objects` is expected to return a list of passengers and therefore, `s1` must be an instance of `passenger-ship`. The assertion presented above could be denoted using a functional syntax, for instance, the following declaration could be used:

```
(assert-result-type (ship-cargo-objects s1)
                    (passenger))
```

The effect of this assertion would be a reclassification of `s1` as a `passenger-ship`. From a Functional Programming perspective, it turns out that the ABox (of a DL system) allows reasoning about types of specific instances based on information about function calls. Note that this kind of reasoning happens at runtime rather than at compile-time. However, type restrictions on function calls could only be defined for accessor functions because for generic functions with general Common Lisp methods no type calculus exists. In order to avoid a mismatch between accessor functions and general functions, only the relational syntax for conceptual assertions is supported.

3.5. Computation of the Concept Precedence List

The TBox defines a partial order relation between concepts (subsumption relation). In order to define how method dispatch is handled, the multiple inheritance lattice must be serialized by a *concept precedence list* which represents a total order between concepts. A concept precedence list is used for the same purposes as a CLOS class precedence list, it defines how an effective method for a specific function call is computed (see the detailed introduction in [15]). A valid concept precedence list is any total ordering that obeys all partial orders defined by the TBox. However, by this requirement only a small set of constraints are defined. There are still several different approaches to serialize a concept lattice. In CLOS the notational order of superclasses in a class definition defines a set of additional order constraints. However, from the viewpoint of Description Logics, the direct superconcepts (the least general subsumers) are unordered. Therefore, in the approach presented in this paper, the relation of parents with respect to method dispatch is left undefined. Procedural code must not depend on any notational order between concept parents.

4. Implementation of Method Dispatch

To allow experiments with the functional layer to CLASSIC, a straightforward implementation for method dispatch with individuals has been provided. CLASSIC dispatch is reduced to CLOS dispatch.

4.1. Reducing CLASSIC dispatch to CLOS dispatch

The implementation of generic functions and method dispatch for CLASSIC is quite simple.[11] The form

`define-generic-function` is used to declare which parameters are handled as ordinary CLOS instances and which parameters are CLASSIC individuals. As a side effect of this declaration, a new function is created (a simple Common Lisp function). This "wrapper" function calls another function with the same name concatenated with the suffix `METHOD`. This function internally represents the generic function and implements the method dispatch. For instance, the macro form:

```
(define-generic-function print-ship-info
     ((ship :classic) (stream :clos)))
```

expands into

```
(PROGN
  (SETF (GET 'PRINT-SHIP-INFO
             :ARGUMENT-SIGNATURE)
        '(:CLASSIC :CLOS))
  (DEFGENERIC PRINT-SHIP-INFO-METHOD
        (IND #:TYPE6806 STREAM))
  (DEFUN PRINT-SHIP-INFO (IND STREAM)
    (FUNCALL (FUNCTION PRINT-SHIP-INFO-METHOD)
             IND
             (COMPUTE-TYPE-ARG IND)
             STREAM)))
```

The internal function `print-ship-info-METHOD` is applied to the same arguments as the wrapper function, but for each parameter which uses CLASSIC dispatch, an additional parameter is inserted (for `ind` this will be `#:type6806`). For each CLASSIC individual, an associated CLOS instance is computed with `compute-type-arg`. In a method definition, the additional arguments are used for the "real dispatching". Note that normal CLOS arguments are treated as usual. The method definition

```
(define-method print-ship-info
    ((ship ship)
     (stream textual-output-stream))
  (format stream "~%Ship ~S." ship)
  ...)
```

expands into

```
(DEFMETHOD PRINT-SHIP-INFO-METHOD
    ((IND T)
     (#:TYPE6807 SHIP)
     (STREAM TEXTUAL-OUTPUT-STREAM))
  (FORMAT STREAM "~%Ship ~S." IND)
  ...)
```

The method is defined for the "real" generic function with suffix `METHOD`. In the example, the specializer `ship` has been "moved" to the second parameter. For the original parameter no specializer is defined. It specializes on `t`, the most general type in Common Lisp, and therefore, this parameter has no "discriminating power". Nevertheless, the original instance must be passed as an argument. In the body of the method, the CLASSIC individual must be bound to `IND`. The corresponding additional parameter is used only for dispatching (its system-generated name is uninterned). Since the substitute specializer must be a CLOS class (here the class `ship` is used), for every named CLASSIC concept (either defined by `define-concept` or `define-primitive-concept`) a corresponding CLOS class is automatically created. The set of superclasses of such a class is computed on the basis of the TBox classification process. Note that the list of superclasses of a class might dynamically change when a defined concept is automatically inserted into the subsumption hierarchy by TBox classification.

The function `compute-type-arg` (see the expansion of `define-generic-function`) computes a CLOS placeholder for a CLASSIC individual. The idea behind `compute-type-arg` is to get the concept of a CLASSIC individual (`classic:cl-ind-parents`), to derive a corresponding CLOS class, and to use the class prototype of this class. One problem is that a CLASSIC individual may be subsumed by more than one named concept, i.e. `classic:cl-ind-parents` returns a list of concepts. When this happens, a new anonymous CLOS class with corresponding superclasses must be created on the fly. The prototype object of this class will then be used. A memoization scheme (with a hash-table `*class-table*`) is used to avoid inflationary class creation.

11. The main idea is inspired by the implementation of presentation type dispatch in CLIM (`define-presentation-generic-function` and `define-presentation-method`).

```
(defun compute-type-arg (ind)
  (or (classic::di-clos-instance ind)
   (let ((class-names
          (mapcar #'classic:cl-name
               (classic:cl-ind-parents ind))))
     (if (null (rest class-names))
         (find-class-prototype
           (find-class (first class-names)))
       (let ((class (gethash class-names
                             *class-table*)))
         (if class
             (find-class-prototype class)
           (let* ((class-name (gensym))
                  (class (find-class
                           class-name)))
             (ensure-clos-class
               :name class-name
               :superclasses class-names)
             (setf (classic::di-clos-instance
                      ind)
                   (find-class-prototype
                      (find-class type-name)))
             (setf (gethash class-names
                            *class-table*)
                   class)
             (find-class-prototype class)))))))
```

With access to the internal data structures of CLASSIC (`classic::di-clos-instance`), an individual can be directly associated with its CLOS counterpart, i.e. the procedure `compute-type-arg` is used only when the individual is reclassified. CLASSIC has been extended to reset the association between an individual and its CLOS representative when the individual is reclassified.

In the following, we have a look at the performance of the current implementation for CLASSIC dispatch.

4.2. Performance considerations

The definition of `compute-type-arg` indicates that the straightforward implementation of CLASSIC dispatch comes at a certain cost. In addition to static costs for the definition of CLOS classes for named CLASSIC concepts, there are some initial dynamic costs:

- some calls to retrieval functions (`classic:cl-name`, `classic:cl-ind-parents`),
- a complex hashing operation over a list of symbols,
- possibly a dynamic creation of a CLOS class,
- the access to the CLOS class prototype,
- and an additional CLOS dispatch step for the substitute argument.

Furthermore, a lot of garbage is created (`mapcar`). Measurements on a Symbolics MacIvory-Model-3 indicate that a dispatched access to a relation with a generic function created by `define-accessors` takes less than one *milli*second. This is approximately three times slower than directly using CLASSIC's retrieval functions on the same processor. Note that accessing a slot of a CLOS instance takes less than a *micro*second, i.e. CLASSIC itself is inevitably slow compared to CLOS. Thus further optimization of CLASSIC (we used Version 2.2) is required.

5. Related Work

Generic functions for Description Logics have also been developed in the Loom System [5]. Loom offers a more powerful Description Logic than CLASSIC though it is incomplete. Method dispatch for individuals is provided by specific generic functions which dispatch on ABOX object but not on CLOS objects. Loom also supports CLOS classes for the implementation of ABox individuals but only a limited sort of reasoning is implemented on these instances (no dynamic reclassification by forward inferences).

With the substitution scheme presented in this paper, method dispatch will be handled by the CLOS system. From the viewpoint of CLOS, extending the object-oriented system can be considered as programming at the metalevel. CLOS itself can be extended using a predefined set of classes and generic functions. The facilities are known as the CLOS Metaobject Protocol [15]. Though some features of the MOP have been used (e.g. `ensure-clos-class` creates a new class at runtime and `find-class-prototype` accesses the prototype instance of a class), the whole system architecture has not been defined in the spirit of the MOP. Using the MOP it might have been possible to avoid the definition of new macros like `define-generic-function` or `define-method`. The MOP idea

enforces an “open system implementation” that avoids the introduction of additional (possibly incompatible) layers for software specification [16]. The original mechanisms provided by CLOS (`defgeneric`, `defmethod`) would have been extended rather than “shadowed”. For several reasons, the MOP has not been used for implementing the extensions defined in this paper. The first reason is simplicity. It is not very easy to find the right entry points into the MOP for a specific implementation problem (but see [12] and [13] for several examples). The other reason is that the MOP is not standardized and is not coherently supported by all Common Lisp systems. The MOP however, might help to find a more optimized implementation. The solution presented in this paper with new macro form definitions is straightforward and the main points of the implementation are easy to understand.

Starting from a different background, the dynamic reclassification or subclassification of objects as a software modeling principle has also been considered by Wieringa et al. [31]. They use a more general order-sorted dynamic logic with equality in the context of “class migration”. See also the extensive work of Goguen and Meseguer (e.g. in [11]). The programming problems presented in this paper can also be solved using the simpler Description Logic approach.

6. Summary and Conclusion

The examples have demonstrated that the facilities of Description Logics allow a system designer to tackle programming problems that cannot easily be solved with object-oriented programming alone. Nevertheless, the main thesis of this paper is that in order to use Description Logics in practical applications, a seamless integration with object-oriented system development methodologies must be realized. Extended generic functions and *multi*methods with CLASSIC dispatch not only allow an incremental way of software definition. In addition to this, they can even been seen as a form of defining assertions that enforce a safer system architecture also for the procedural parts (the same holds for CLOS [18]).

The paper has presented an approach that demonstrates how the integration of CLOS and CLASSIC can be achieved. The notion of generic functions and methods have been extended to define how CLASSIC individuals can be incorporated into the dispatch mechanism of CLOS. We have discussed a prototype implementation that is easy to understand and allows the integration to be tested in larger applications. With the extended dispatch mechanism for CLASSIC instances, a large system for generating interfaces has been implemented [20], [21].

References

[1] Borgida, A., Brachman, R.J., McGuiness, D.L., Resnick, L.A., *CLASSIC: A Structural Data Model for Objects*, in: Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, May-June, 1989.

[2] Borgida, A., Patel-Schneider, P.F., *A Semantics and Complete Algorithm for Subsumption in the CLASSIC Description Logic*, Journal of Artificial Intelligence Research, No. 1, Morgan Kaufmann Publ., 1994, pp. 277-308.

[3] Brachman, R.J., *“Reducing” CLASSIC to Practice: Knowledge Representation Theory Meets Reality*, in: Proc. KR’92 Principles of Knowledge Representation and Reasoning, Nebel, B., Rich, C., Swartout, W. (Eds.) Morgan Kaufmann Publ., 1992, pp. 247-258.

[4] Brachman, R.J., McGuiness, D.L., Patel-Schneider, P.F., Resnick, L.A., *Living with CLASSIC: When and How to Use a KL-ONE-like Language*, in: Principles of Semantic Networks - Explorations in the Representation of Knowledge, Sowa, J. (Ed.), Morgan Kaufmann Publ., 1991, pp. 401-456.

[5] Brill, D., *Loom Reference Manual, Version 2.0*, USC/ISI, 4676 Admiralty Way, Marina del Rey, CA 90292, December, 1993.

[6] Bruce, K.B., Crabtree, J., Murtagh, T.P., Gent, R. van, Dimock, A., Muller, R., *Safe and Decidable Type Checking in an Object-Oriented Language*, in: Proc. OOPSLA’93, ACM SIGPLAN NOTICES, Volume 28, No. 10, October 1993, pp. 29-46.

[7] *Common Lisp Interface Manager: User Guide*, Franz Inc., 1994.

[8] Communication of the ACM, Special issue about “*Object-Oriented Experiences and Future Trends*”, October 1995, Vol. 38, No. 10.

[9] Computer - Innovative technology for computer professionals, Special issue about “*Object-Oriented Technology”*, October 1995.

[10] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[11] Goguen, J.A., Meseguer, J., *Unifying Functional, Object-Oriented and Relational Progamming with Logical Semantics*, in: Research Directions in Object-Oriented Programming, Shriver, B., Wegner, P. (Eds.), MIT Press, 1987, pp. 417-477.

[12] Haarslev, V., Möller, R., *Visualization and Graphical Layout in Object-Oriented Systems*, Journal of Visual Languages and Computing, Nr. 3, 1992, pp. 1-23.

[13] Haarslev, V., Möller, R., *A Framework for Visualizing Object-Oriented Systems*, in: Proceedings OOPSLA'90, SIGPLAN Notices, Volume 25, No. 10, October 1990, pp. 237-244.

[14] *Keene*, S.E., *Object-Oriented Programming in CLOS: A Programmer's Guide to CLOS*, Addison-Wesley, 1989.

[15] Kiczales, G., des Rivières, J., Bobrow, D.G., The Art of the Metaobject Protocol, MIT Press, 1991.

[16] Kiczales, G., *Towards a New Model of Abstraction in the Engineering of Software*, in: Proc. IMSA'92 Workshop on Relflection and Meta-level Architectures, 1992.

[17] Kiczales, G., *Traces (A Cut at the "Make Isn't Generic" Problem)*, in: Proceedings of ISOTAS'93, also available as: `ftp://parcftp.xerox.com/pub/openimplementations/traces.ps.Z`.

[18] Lamping, J., Abadi, M., *Methods as Assertions*, Xerox Palo Alto Research Center, available as: `ftp://parcftp.xerox.com/pub/openimplementations/methods-as-assertions.ps.Z`.

[19] Möller, R., *Extending CLASSIC with Generic Functions and Methods*, `http://kogs-www.informatik.uni-hamburg.de/~moeller/`, 1996.

[20] Möller, R., *Reasoning about Domain Knowledge and User Actions for Interactive Systems Development*, IFIP Working Groups 8.1/13.2 Conference on Domain Knowledge for Interactive System Design, Chapman & Hall, 1996.

[21] Möller, R., *Knowledge-Based Dialog Structuring for Graphics Interaction*, in: Proc. ECAI'96, Budapest, Hungary, August 1996.

[22] Möller, R., *User Interface Management Systems: The CLIM Perspective*, `http://kogs-www.informatik.uni-hamburg.de/~moeller/uims-clim/clim-intro.html`, 1996.

[23] Nebel, B., *Reasoning and Revision in Hybrid Representation Systems*, Lecture Notes in Artificial Intelligence, Vol. 422, Springer-Verlag, 1990.

[24] Paepcke, A., *Object-Oriented Programming - The CLOS Perspective*, MIT Press, 1993.

[25] Patel-Schneider, P.F., McGuiness, D.L., Brachman, R.J., Resnick, L.A., *The CLASSIC Knowledge Representation System: Guiding Principles and Implementation Rationale*, SIGART Bulletin, Vol. 2, No. 3, pp 108-113.

[26] Patel-Schneider, P.F., Swartout, B., *Description Logic Specification from the KRSS Effort*, `ksl.stanford.edu:/pub/knowledge-sharing/papers/dl-spec.ps`.

[27] Peyton Jones, S.L., Hall, C., Hammond, K., Partain, W., Wadler, P., *The Glasgow Haskell Compiler: A Technical Overview*, in: Proc. UK Joint Framework for Information Technology (JFIT), Technical Conference, Keele, 1993.

[28] Rao, R., *Implementational Reflection in Silica*, in: Informal Proceedings of ECOOP/OOPSLA'90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming, October, 1990.

[29] Resnick, L.A., Borgida, A., Brachman, R.J., McGuiness, D.L., Patel-Schneider, P.F., Zalondek, K.C., *CLASSIC Description and Reference Manual for the Common Lisp Implementation*, Version 2.2, 1993.

[30] Steele, G.L., *Common Lisp - The Language, Second Edition*, Digital Press, 1990.

[31] Wieringa, R., de Jonge, W., Spruit, P., Roles and Dynamic Subclasses: A Modal Logic Approach, in: Proc. ECOOP'94, Tokoro, M., Pareschi, R. (Eds.), Springer, LNCS 821, 1994, pp. 32-59.

[32] Woods, W.A., Schmolze, J.G., *The KL-ONE Family*, in: Semantic Networks in Artificial Intelligence, Lehmann, F. (Ed.), Pergamon Press, 1992, pp. 133-177.

[33] Wright, J.R., Weixelbaum, E.S., Vesonder, G.T., Brown, K., Palmer, S.R., Berman, J.I., Moore, H.H., *A Knowledge-Based Configurator That Supports Sales, Engineering, and Manufacturing at AT&T Network Systems*, AI Magazine, Vol 14, No. 3, 1993, pp. 69-80.