Object oriented data concepts for SDL

Dipl.-Inf. Martin v. Löwis of Menar
Dipl.-Inf. Ralf Schröder
Humboldt-Universität zu Berlin
Axel-Springer-Straße 54a
10117 Berlin
Germany

tel.:	*(+49) 30 20 181 321*
fax:	*(+49) 30 20 181 234*
e-mail:	*{loewis|r.schroeder}@informatik.hu-berlin.de*

Abstract

With the 1992 revision of SDL [1], object oriented structuring concepts were introduced into the language. However, some object oriented concepts known from modern languages are still missing, e.g. exception handling and references, which are related to polymorphism as well as late binding of operators. The main reason for these lacks is the formal base of data types ACT ONE [2]. This algebraic calculus makes it more difficult to introduce those concepts.

Today‘s applications of SDL are embedded in system designs together with other specification techniques, e.g. IDL [10]. The interaction with these additional techniques requires the extension of SDL with modern language concepts. Humboldt University developed solutions which are implemented in the SDL Integrated Tool Environment (SITE). These solutions are being presented to the ITU-T for standardization in the next revision of SDL.

Keywords
object oriented data, references, exception handling

1 EXCEPTION HANDLING

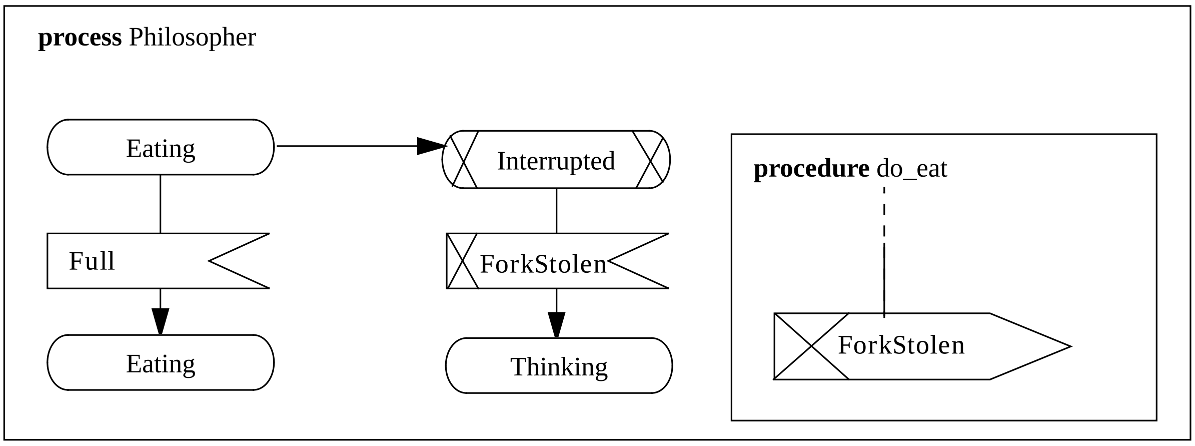
When an SDL system is executed, it is possible that a system state is reached where a useful continuation cannot be expressed with the current syntax. Such situations are caused by errors in expression elaboration (e.g. division by 0) or by dead-locks in remote procedure calls. The exception concept provides a new control flow element for SDL allowing the user to manage those situations. If an error occurs, the erroneous action is interrupted by throwing an exception, an suitable exception handler is determined and called. An untreated exception within a procedure interrupts the call of the procedure and raises the same exception for the caller. This is applicable to remote procedure calls, too. If the exception is not caught by a process or service, the further behaviour of the system is undefined.

Exceptions are a new syntactic entity with a name and an optional sort parameter list, e.g.:

```
exception ForkStolen;
```

Declared exceptions can be raised explicitly as a final action in an SDL transition, i.e. an exception instance is created with suitable sort parameters. Resuming the interrupted action is not supported. An exception handler can catch a thrown exception:

```
exceptionhandler interrupted;
    catch ForkStolen;
        task ”DoSomethingAboutIt”;
        raise -;   /* reraise the current exception with the same arguments */
```



The graphic syntax is demonstrated with the process „Philosopher“. The syntactic notation of an exception handler corresponds to state and input constructions:

- it has to be defined where a state can be specified,
- catch clauses can be specified as virtual,
- asterisk notations are possible with the same rules as for states and inputs.

Moreover, an exception can be defined and used as context parameter. An exception handler has to be installed for a control flow element. This is possible for

- complete procedure, operator, service, or process graph,
- state or even an exception handler,
- transition, i.e. input or catch notations,
- single actions.

An exception handler for a thrown exception is searched with decreasing locality. Because exceptions can occur within an exception handler, too, there are at most 6 levels for an active SDL entity (action, catch clause, exception handler, transition, state, graph).

1.1 Exceptions and remote procedures

Remote procedures are defined with a transformation model based on signal exchange (cf. [1] chapter 4.14 Remote procedures). If the procedure of a server produce an exception which is not caught by the procedure itself, the caller as well as the server has to be informed about that situation. Therefore the exception is transformed by the server process to a signal, which is sent to the caller. Nevertheless, the exception handler of the remote call is searched and executed. The transformed client specification receives that signal and throws a local exception. If the exception is uncaught in either client or server, the further behaviour of the system is undefined.

All exceptions, which can be raised by a procedure have to be specified in the procedure signature, and the remote procedure declaration, respectively. It is possible to check the correct signature statically, i.e. a client is able to see all possible exceptions of the remote call. This syntax extension is analogous the OMG-IDL [10].

If a process exports multiple procedure, sometimes a certain order of calls has to be guaranteed. Erroneous client calls should be rejected without calling the procedure of the server side. Therefore a new syntactical construct is introduced in addition to save and input constructions for remote procedures: an rpc reject clause. This clause can reject the call with a specified exception for the client. Next, the specified action of the server is executed similar to an input clause:

```
state Booting;
    input procedure SetHostName;
        nextstate -;
    save procedure QueryInformation;
    input procedure Reboot raises ConnectionReset;
        task 'Log attempt';
        nextstate -;
```

Sometimes a remote procedure call remains unanswered for a long time or is even dead-locked, e.g. if the server does not exist. Here it is desirable to abort the call locally. The solution is to set a timer and to connect this timer syntactically with the remote procedure call. The transformed client specification now is able to receive the timer signal alternatively to remote procedure answers and remote procedure exceptions. As action an exception with the name of the timer is thrown:

...
set (now+100, MaxRPC);
call AnRPC **to** Object **timer** MaxRPC; **onexception** timeout;
...
exceptionhandler timeout;
    catch MaxRPC;
        ...

This complex syntax of the remote procedure call will be changed in the final language proposal.

1.2 Predefined exception

During the execution of an SDL system there are situations, where the further behaviour of the system is undefined. Some of those situations cannot be avoided by using a special control flow, e.g. the access to undefined variables. In such situations a predefined exception is thrown. The following exception are predefined:

- OutOfRange – A syntype check fails; this exceptional condition also covers attempts to access an array outside its index range. This is because of the underlying ACT-ONE model and the way the Array generator is defined. Probably, this exceptions is replaced by an range check operator in the new SDL version.
- UndefinedValue – Attempt to access a variable with an “undefined” value (Z.100, 5.4.2.2), to export or import a remote variable with an “undefined” value (Z.100, 4.14/5.4.2.2), to view a revealed variable which has an “undefined” value (Z.100, 5.4.4.4/5.4.4.2).
- NoRevealer – No revealer found when evaluating a **view** expression (Z.100, 5.4.4.4).
- InvalidReference – Wrong access to a term of a reference (cf. next section).

The discussion about the set of predefined exceptions is not finished yet.

1.3 Mapping from IDL to SDL

One motivation for the exception proposal was to simplify the mapping from OMG-IDL to SDL which is proposed in [9]. IDL exceptions can now be mapped directly to SDL exceptions instead of signals. CORBA system exceptions should be predefined in an SDL package „CORBA“. Within ITU-T there are developments under progress to combine an improved IDL version with the new SDL version.

2 REFERENCES

ACT ONE as formal background of SDL data types (cf. [2]) has a value semantics. This implies that an automatic implementation of an SDL specification in a target language often requires copy operations of values. A good optimization can reduce the number of copy operations but is not able to avoid all copies. The designer of SDL systems knows the problem but cannot solve it with the current SDL semantics. Moreover, it is desirable to specify list and graph structures with SDL, and to have polymorphic properties. A possible solution is a reference concept for SDL.

The main idea is, that for each SDL sort an implicit reference sort is introduced. The name of the reference sort is the name of the sort prefixed with „^“. The character „^“ is deleted from the set of characters for SDL names. Reference sorts can be used like other data types, e.g. as types for variables as well as type of structure fields. One is followed the style of syntax descriptions in Z.100 [1], the notation is:

<reference identifier> ::= [<qualifier>] ^ { <sort name>|<syntype name>}
<sort> ::= <sort identifier> | <syntype> | <reference identifier>

There is also a semantic model which is based on the current SDL semantics and a set of transformation rules. The main idea is, that terms of references are kept in a storage of a separate SDL process, called the reference process of a sort. This storage is addressed by references. Each sort which is not derived from an other sort has such a process. The SDL specification of this process is here not given in detail. The access (creation, modification) of a reference is a remote procedure call, formally. However, it is not expected, that such a call changes the active expression „sender“. The reference process can never be addressed in a user specification!

An abstract reference sort can be introduced analogous to the predefined sort „PId“:

```
newtype REFERENCE
    literals Nil;
    operators unique! : REFERENCE -> REFERENCE;
    axioms
        for all r in REFERENCE ( unique!(r) /= Nil;);
        for all r1,r2 in REFERENCE (unique!(r1) = unique!(r2)  ==  r1 = r2;);
    default Nil;
endnewtype;
```

Let „Base“ be an arbitrary data type and „Sort“ a derivation, e.g.

```
newtype Base endnewtype Base;

newtype Sort inherits Base operators all; endnewtype;
```

References of sorts without inheritance relation should not be assignment compatible. Therefore different derivations of the sort „REFERENCE“ are introduced according to the following rules:

1. If the sort does not contain an inheritance construction, then the reference sort is defined as derivation of "REFERENCE":

```
newtype ^Base inherits REFERENCE operators all; endnewtype ^Base;
```

2. If there is an inheritance construct then the reference sort is defined as syntype of the base type reference:

```
syntype ^Sort = ^Base endsyntype;
```

These sorts are called reference sorts of the sorts „Base“ or „Sort“, respectively. Consequently, all reference sorts of an inheritance hierarchy are assignment compatible according to the syntype model of SDL. This is not desirable. An enhanced syntype construct can restrict the assignment compatibility. The restrictions for references are similar to those of other object oriented languages:

- References of a base sort cannot be assigned to references of a derivation if the value of the reference is not at least a value of the derivation. The thrown exception would be „OutOfRange“.
- References of different inheritance branches can be not assigned. This is a static property.

The syntype concept has to be extended to allow these static or dynamic checks. There are additional operators to manipulate references. These operators are explained in the next sections.

For the following examples it is assumed to have variables „**dcl** int Integer, int_ref ^Integer;“ and a derived sort of „Integer“ with variable „dcl new_int NewInteger, new_int_ref NewInteger;“.

2.1 Creation

A reference is created by calling the implicitly defined operator

```
operators New : Sort -> ^Sort;
```

It is available for each SDL sort. This operation binds a copy of the argument to the returned reference value, which is different to all other reference values of the system. The call has to be qualified (e.g. "**type** Integer new(42)"), if the type of the argument is not unique. Formally, the operator call is transformed to an remote procedure call based on the declaration

```
remote procedure Sort_MAKE; fpar in Sort; returns ^Sort;
```

The implementation is provided implicitly within the reference process of „Sort“. It uses the hidden operation „unique!“ of the reference sort to provide new references. Alternatively, references can be reused by some kind of storage managment.The given value, called the assigned term, and its type information is stored in a table (e.g. an SDL array) together with the reference.

Example:

```
task new_int_ref := new(42); /* new oparation unique because of the return type */
task int_ref := new_int_ref;  /* the assigned term is at least an Integer */
task int_ref  := type NewInteger new(0);  /* qualified use of new */
task int_ref  := type Integer new(1);      /* has to be qualified, too */
task new_int_ref := int_ref; /* throws InvalidReference, the value is Integer */
```

2.2 Access to the assigned term

A copy of the assigned term of a reference is returned by the implicitly defined prefix operator „^“ with the same precedence as „not“ and „-“:

operators "^" : ^Sort -> Sort;

In most cases, the operator call is unique because of the return type. Otherwise the call has to be qualified. References of inherited sorts are restricted assignment compatible. If the type of the reference term is not equal to the return type of the operator, a predefined exception "InvalidReference" is thrown. The operation is formally transformed to a remote procedure call

remote procedure Sort_EXTRACT; **fpar in** ^Sort; **returns** Sort; **raises** InvalidReference;

where the type context is attended. The implicit implementation of the procedure within the reference process checks the stored type information and returns the assigned term, or throws an exception, respectively.

Example:

```
/* continue with the assignments form the previous section */
task int := ^int_ref  /* possible */
task new_int := ^int_ref  /* possible, but throws an exception because is is: */
task new_int := call NewInteger_EXTRACT(int_ref); /* wrong term! */
```

2.3 Modification of the assigned term

The assigned term of a reference can be modified by using a new syntactic variable notation:

```
<variable> ::= <variable identifier> | <idexed variable> | <field variable>   /* Z.100 */
             | ^ <variable>       /* reference extension */
```

An assignment with the new reference notation is transformed to an remote procedure call with the signature:

remote procedure Sort_MODIFY; **fpar in** ^Sort, **in** Sort; **raises** InvalidReference;

where the sort of the value is considered a type context for the operator resolution. The sort of the value has to have an inheritance relation to the sort of the reference (static assignment property). The implicit implementation of the remote procedure within the reference process replaces the term without type check against the old value. This can be changed within the semantic model if a more strong type concept is desired.

Example:

```
task ^int_ref := type NewInteger 128; /* previous value was an Integer */
task ^new_int_ref := ^int_ref; /* overwrite 42, exception prevented from assign */
```

2.4 Determining the sort of the assigned term

In some situations it is important to know, that the value of a reference is of a certain type. It is proposed to introduce two operators:

```
operator
    Sort_VERIFY : ^Sort -> Boolean;
    Sort_SUBTYPE: ^Sort -> Boolean;
```

which returns the value „True“ if the assigned term is a term of the sort „Sort“ or at least a term of the sort „Sort“, respectively. The call is transformed to an remote procedure call, too. Syntactical notation as well as the semantic properties are left open yet.

Example:

```
decision call NewInteger_VERIFY(int_ref);
    /* returns True, because 128;but note, that */
decision call Integer_VERIFY(int_ref);
   /* returns "False". The result of the next operation remains True: */
decision call Integer_SUBTYPE(int_ref);
```

2.5 Release of references

A reference, except „Nil“, always has an assigned term. If the reference value is not used in the SDL system anymore, e.g. the reference variable is reassigned, the binding to the term could be released and the reference value could be reused. This storage management is out of scope of SDL. A tool provider may define a „delete“ operation in combination with a tool specific storage management.

In the example the reference to the „NewInteger“ value „128“ disappears if „int_ref“ is assigned the „Nil“ value. Now this references could be reused.

3 INHERITANCE

There are data structures with a well known semantics. These are sorts like „Integer“ and „Boolean“, but also constructions like strings or structures. Here SDL defines transformation rules to map syntactic notations to a correct algebraic specification. The language definition of SDL provides these rules for structures, index access, and in context of Z.105 for ASN.1 constructions. It is syntactically impossible to use these constructions in combination with inheritance. Therefore the syntactic rules for sort definitions are changed as follows:

- Generators are deleted from the language. Predefined generators are replaced by sorts with context parameters (sort templates).
- Inheritance can be applied for each sort, i.e. it is not part of the extended properties.
- Alternatively, extended properties are: literals, structures, choices, enumerations. Inherited sorts have to use the extended property of the base sort.

Because the syntactical representation of a new SDL language is not so important here, only an incomplete grammar is given:

```
<partial type definition> ::= newtype <sort name>
                                  [ <formal context parameter>]
                                  [ <inheritance rule> ]
                                  [ <extended properties> ]
                                  [ <operator list> ]
                                  [ <operator definitions> ]
                                  [ <default assignment> ]
                              endnewtype [ <sort name> ]

<extended properties> ::=     <literal lists>
                          |   <structure definition>
                          |   <choice definition>
                          |   <enumerated defininition>
```

The next sections discuss the special constructs in detail. The final goal is to prohibit ACT ONE based sort definitions in specifications and to enforce the use of structural data descriptions.

3.1 Literals

Literal definitions are considered as structural properties to avoid a mixture with other structural constructions. Literals have the same semantics as enumerations but without complex operations. There are implicit defined and visible conversion operators to realize explicit conversion of a derived value to an base value and vice versa. The transformation of an added literal to the base sort raises an exception.

3.2 SDL structures

A structure is a special syntactic construction in SDL, e.g.:

```
newtype Base struct
    i Integer;
endnewtype Base;
```

This is an abbreviation for several operators, which allows the initialization and the access of structure fields. Formally, the access to fields is impossible if the structure is not initialized completely. Hence, the inheritance of structures has also a problem with new fields if there is some kind of assignment compatibility. Therefore a formal change of the "Make!" operators is proposed. The points are:

- fields can be initialized separately,
- structures are equal if all initialized fields are equal,

- structure initialization with „(. ... .)“ can omit expressions analogous to output expression lists. However, the final ’,’ cannot be omitted because structure values cannot be qualified if the value is not unique.

The semantics of these structures is the same semantics as an Z.105 [4] based ASN.1 SEQUENCE with optional fields without „Present“ operation but a default initialization „{}“. Structure access with references can be formally defined by operations with corresponding signatures:

```
operators
    iModify! : virtual ^Base, Integer -> ^Base;
    iExtract!: virtual ^Base -> Integer;
operator iModify!; fpar ref ^Base, i Integer; returns ^Base;
    dcl base Base := ^ref;
    start; base!i := i, ^ref = base; return ref;
endoperator;
operator iExtract!; fpar ref ^Base; returns Integer;
    dcl base Base := ^ref;
    start; return base!i;
endoperator;
```

The „^Base“ arguments are tagged as virtual to allow the access even for references with assigned terms o derivations. This concept is explained in chapter "Virtual operators". With these operators, the standard transformation rules of SDL for structure access can be used. A sort with structure properties also provides conversion operators to and from a derivation:

```
operators
    NewBase : Base -> NewBase;
    Base : NewBase -> Base;

operator NewBase; fpar base Base; returns newbase NewBase;
    start;
        task newbase := (. base!i , .);
        return;
endoperator;

operator Base; fpar newbase NewBase; returns base Base;
    start;
        task base := (. newbase!i .);
        return;
endoperator;
```

3.3 Choices

Choice constructions are proposed to be a built-in construct analogous to the SDL structures with the same semantics like ASN.1 CHOICE construction in context of Z.105 [4]. Inheritance of CHOICE constructions is analogous to inheritance of SDL structures. The example demonstrates the proposed syntax:

```
Base ::= choice { i Integer }    /* optional support for  Z.105 like syntax */
newtype Sort inherits Base;
    adding choice
        b Boolean;
endnewtype;
```

The difference to SDL structures is the implied enumeration constructed from the field names. If a choice value is used polymorphic, some fields can be unknown. This problem can be solved with a small modification of the Z.105[4] mapping: a special, possibly hidden literal „unknown!“ is returned if the assigned fields is not introduced in the base type definition. The example demonstrates such a situation:

```
dcl ref ^Base := <<type Sort>>new( { b True } );
...
decision ref!present;
    (i) : ...
    else : ... /* would be reached here */
enddecision;
```

3.4 Index access

The definition of operators

```
operators
    Extract! : Sort,Index -> Item;
    Modify! : Sort, Index, Item -> Sort;
```

allows the use of left or right hand index expressions. The access via a reference requires the virtual operators to be analogous to structures:

```
operators
    Extract! : virtual ^Sort,Index -> Item;
    Modify! : virtual ^Sort, Index, Item -> ^Sort;
operator Extract!; fpar ref ^Sort, index Index; returns Item;
    dcl sort Sort := ^ref;
    start; return sort(index);
endoperator;
operator Modify!: fpar ref ^Sort, index Index, item Item; returns ^Sort;
    dcl sort Sort := ^ref;
    start;
        task sort(index) := item, ^ref := sort;
        return ^sort;
endoperator;
```

The definition of these operators has to be included in the package „Predefined“[3] for „String“ and „Array“ templates.

3.5 Enumerations

Enumerations are a special case of sort definitions. There are well defined properties of enumerations, if SDL is combined with ASN.1. The idea is to indicate this property in an SDL signature, too:

```
newtype Base
    enumerated { one, two }
endnewtype;


newtype Sort inherits Base
    adding enumerated { three }
endnewtype;
```

ASN.1 allows/requires the binding of numbers to the literals

```
Base ::= ENUMERATED { one(1), two(2) }
```

which are available by calling the operator „Num“. The operator „Num“ as well as the syntactical mapping to SDL-sorts is defined in Z.105 [4]. Inherited enumeration sorts have conversion operators similar to literal sort.

4 VIRTUAL OPERATORS

Virtual operators perform actions, depending on an argument. It is possible, that the called action is not visible in the calling scope. The example demonstrates a simple "print" operation, which returns a type description string. The used syntax is an first approach only:

```
newtype base_msg
    operators print : virtual ^base_msg -> Charstring;
    operator print; fpar ref ^base_msg; returns Charstring;
        start; return ‚base_msg‘;
    endoperator;
endnewtype;
```

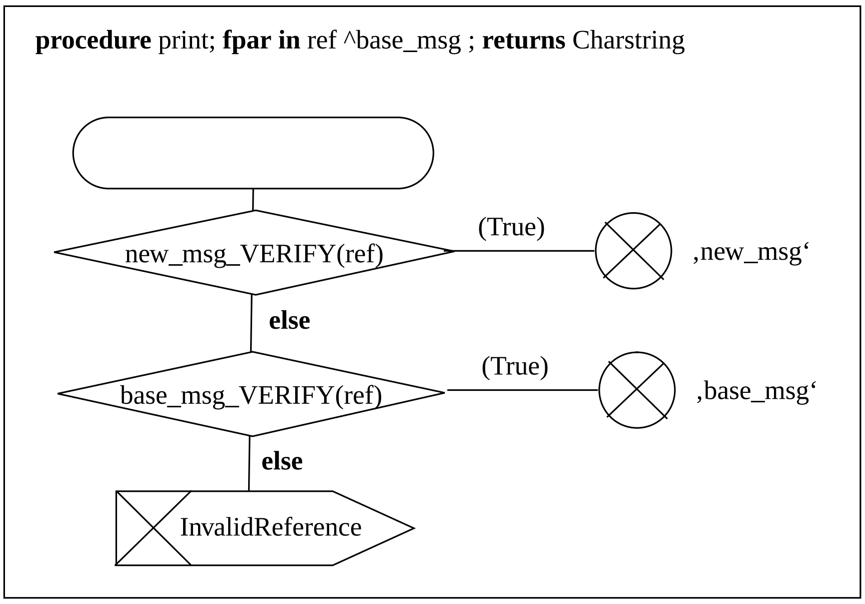
A redefinition replaces the reference to the type of the term, which is expected now:

```
newtype new_msg inherits base_msg operators all;
    operators print : redefined ^new_msg -> Charstring;
    operator print; fpar ref ^new_msg; returns Charstring;
        start; return ‚new_msg‘;
    endoperator;
endnewtype;
```

Virtual calls with reference arguments can be modelled with SDL procedures. This procedure implements a late binding according to the type of the term of the reference with a simple decision.

```
procedure print;
    fpar in ref ^base_msg ; returns Charstring;
    start;
        decision new_msg_VERIFY(ref);
            (True): return ‚new_msg‘;
            else:
        enddecision;

        decision base_msg_VERIFY(ref);
            (True): return ‚base_msg‘;
            else: raise InvalidReference;
        enddecision;
endprocedure;
```



This approach assumes, that all sorts are visible. Alternatively the virtuality concept of procedures can be used. Each derived procedure jumps to a label of the base procedure if the own decisions fail, instead of raising the exception. It is possible, that a virtual operator is visible if a reference of a sort is used where its operator is not yet defined. The call of this operator with such an reference should be excluded statically (not conform to the current syntype concept).

By using the mapping above, operator calls with assigned terms of sorts without redefinition cause an exception. There could be an alternative mapping.

5 STATIC SORT VARIABLES

Other object oriented languages know the concept of static class variables. This is possible for SDL sorts by using the reference process of the sort. For example, a sort shall provide the name as a value of a class variable. This can be specified as follows:

```
newtype Sort;
    static out name Charstring := ‚Sort‘;
endnewtype;
```

The access is

```
task string_var := Sort!name;
```

An other application is to provide tag information for ASN.1 type, which can be accessed in SDL by using static sort variables. If the static sort variable is writable, this is a way to provide shared global variables. Such an application is the specification of reference counter for storage management.
Formally the specification above implies a variable „name“ of the sort „Charstring“ within the reference process of „Sort“. The access can be done with procedures, e.g.

```
exported procedure nameExtract!; fpar ^Sort; returns Charstring;
    start; return name;
endprocedure;
```

All definitions and the access notation are provided by syntactic transformation. If the keyword **out** is omitted or replaced by **in**/**out**, then a procedure „nameModify!“ is defined, too.

6 IMPLEMENTATION ISSUES AND SUMMARY

The concept of exception handling was implemented and tested with the SITE tools of Humboldt University [5]. On that base, the concept was proposed within ITU-T for standardization (detailed description in [7] and [8]) for the new revision of SDL. This proposal is stable, the next step is to do the editorial work for inclusion into the new language definition.

Object oriented data are an open issue within ITU-T. Therefore this proposal is seen as start point for the discussion of that big topic. Probably, ACT ONE becomes not the formal base of data in the new SDL version. Nevertheless, most of the concepts are implemented in experimental tool versions of SITE [6]. This implementation is used to show some consistency properties as well as for experiments. The formal background of this proposal gives ideas what happens, if some properties of the data concepts are changed. Such point are for example:
- references to local process variables,
- assignment compatibility for derived data types,
- forbid derivations of simple sorts like „Integer“ and
- storage management.

The next step could be the definition of a semantic built-in concept for SDL. This avoids some complex transformation rules and new properties like the expanded syntype check.

7 REFERENCES

[1] ITU-T: SDL - Specification Description Language, International Standard Recommendation Z.100, Genf,1992

[2] ITU-T: Annex C in SDL - Specification Description Language: Initial Algebra Model, International Standard Recommendation Z.100, Genf,1992

[3] ITU-T: Annex F in SDL - Specification Description Language: Formal Semantics, International Standard Recommendation Z.100, Genf,1992

[4] ITU-T: SDL in combination with ASN.1, Recommendation Z.105, Geneva, 1997.

[5] http://www.informatik.hu-berlin.de/Themen/SITE: SITE – SDL Integrated Tool Environment, Humboldt-University Berlin, 1998.

[6] http://www.informatik.hu-berlin.de/Themen/SITE/sdl2000.html: Actual research implementations of SDL with SITE, Humboldt-University Berlin, 1998

[7] ITU-T Q.2/10 Rapporteur: TD 56-E. New constructs for exception handling. SG 10 meeting, Geneva, March 19988.

[8] ITU-T Q.2/10 Rapporteur: TD-62E. Associated Timers. SG 10 meeting, Geneva, March 1998.

[9] ITU-T: Basic Reference Model of ODP: Architectural Semantics, specification techniques and formalisms, International Standard Recommendations X.90x, Geneva, 1997.

[10] Object Management Group: The Common Object Request Broker: Architecture and Specification, OMG document formal/98-02-33, 1998.