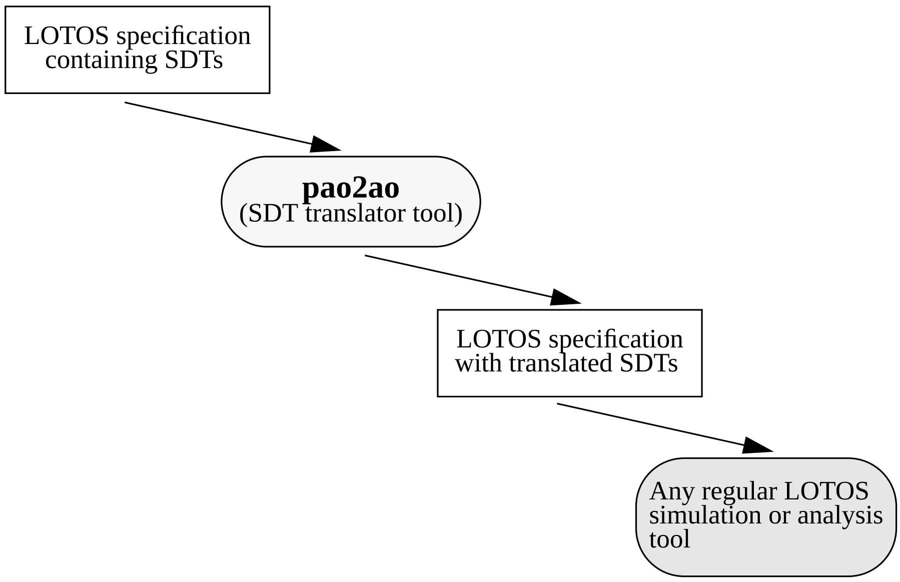
The Structured Data Type Translator ‘pao2ao’

SILKE STORP
BERNARD STEPIEN

ACT ONE is an abstract data type language that is very flexible, because it allows to define any function indepedently from any application. This advantage is unfortunatly undermined by the fact that abstract data type has to be systematically constructed from scratch, thus requiring sometimes considerable work and thus reducing its usability. Extensions to the language have never taken place due to the objectives of stability defined by the various ISO work group. Experience by various LOTOS community users has shown that some frequently used concepts should be implemented in ACT ONE in a standard way. This has resulted in several proposals [SS91] [UT92]. Both consisted in defining an extension to the language with semantics to translate these extensions into traditional ACT ONE. The following work consisted in writing a translator that would take a LOTOS specification where new Structured Data Types (SDT) are present both as definitions, but also as value expressions in actions, process instances value actualizations and guards.

There were several strategies possible. The decision to build a stand alone translator was based on several factors. Among them is the fact that several LOTOS tools exist, and a translator would have avoided to modify all of the existing tools. Also, the Structured Data type are presently experimental and are not included yet in the standard. The translator is then considered as allowing a user to reason in a higher level of abstraction independent from ACT ONE itself. In this case the resulting ACT ONE translation is considered as strictly an executable version similar to assembly language in traditional computer languages. (fig.1)

FIGURE 1. Tool chain with SDT translation tool 'pao2ao'



STRUCTURED DATA TYPE DESCRIPTION

There are six structured data types that can be translated:

- **enumerations**: a finite set of constants is defined, which can be subdivided into 'subclasses'
- **sort unions**: a new sort is built by unifying certain sorts, which can be subdivided into 'subclasses'. Mapping- and selection-functions of the sorts are provided
- **sets**: a set over an element sort is defined, together with special value declarations
- **sequences (strings)**: a list over an element sort is defined, together with special value declarations
- **arrays**: an array over an element sort is defined, which has a fixed size and is one- or multi-dimensional. It is possible to declare special values
- **records**: a record consists of various fields, which are selected by names and have various sorts. Each field may be optional or may have an initial (default) value

USAGE OF 'pao2ao'

To call the translator 'pao2ao' the shell variable $PATH has to contain the bin directory of the SEM group:

/net/jupiter/usr85/SEM/bin

The specification with structured data types must be in a file have the suffix '.sdt'. The suffix is different or there is no suffix, then the tool will look for a file 'name.sdt'. The tool generates a file 'name.lot'. **Caution**: the tool does not check whether there exists already a file with that name.

A help message, saying how the tool is called and which options are possible, can be obtained by using the '-h' option:

```
> pao2ao -hef

   usage:  sdt2ao inputfile[.sdt] [-o outputfile] [-g]
            where [-g]:    generation of visualisation annotations
                           and Demon graphic template program
      or:  sdt2ao -h
           which prints this message
```

A successfull translation produces the following output. (Please notice that the file containing the specification with structured data types needs to have the suffix '.sdt'. The suffix can be omitted):

```
> pao2ao test

 Gesellschaft fuer Mathematik und Datenverarbeitung
 Forschungszentrum fuer Offene Kommunikationssysteme
 Structured Data Type to Act One Translator

 (C) 1993

 Translation of file: test.sdt starts !

 Parsing and syntax checking start !
 Parsing and syntax ckecking were successful !

 Unparsing starts !
 Output is written to file: 'test.lot'

 Unparsing successful !

 Translation of Structured Data Types was completely successful !!!
```

Interpreting Error Reports of the Static Semantic Checker

The translator ‘pao2ao’ only checks the correctness of the syntax of a specification. It does not check the static semantic. This has to be done after the translation by using commonly available LOTOS tools, like for instance TOPO that is distributed together with the *lite* tool. The interpretation of the errors, reported by these tools, is sometimes a bit tricky because they point to the translated specification.

In the following we will list some typical error messages and give guideline how to interpret them. Of cause the list can not be complete and there are many, many other cases possible. We have used the TOPO compiler for the static semantic check.

In most cases, the best strategy is look only at the errors reported at the first 1 or 2 lines and try to correct them. The following are probably inferences of the first errors.

TYPE NOT FOUND

type “Set” not found at library
type “NaturalNumber” not found at library
type “String” not found at library

Library must be changed. The structured data types from the library. The library must provide these three types and order relation on the terms. For lite the mod-is library is necesary (see annex for the required signature of the three types from above).

sorts of value expression are different / undefined operation

test.lot:165: lsa: sorts of value expressions are different
test.lot:165: lsa: undefined operation: array
test.lot:165: lsa: undefined operation: a

The operation ‘a’ is not defined.

```
test.lot:82: lsa: sorts of value expressions are different
test.lot:82: lsa: undefined operation: array
```

The number of arguments of the array is wrong.

```
ina.lot:1012: lsa: undefined operation: overlay
```

This message indicates that the record name might be wrong. Each record is translated into a set of record fields which is overlayed over a set of initialization values.

Signatures of Translated Structured Data Types

Structured data types use four library data types from the '**mod-is**' library: String, Set, NaturalNumber, Boolean. These types are imported from the library, but only if a type is used, and no type exists in the specification that has the same name. The translator tool also generates two other types:

1. **DecimalNaturalNumber**: enrichment of 'NaturalNumber' by functions which are mapped on successors of 0 (used for the array indexes)

1, 2, 3, 4, 5, 6, 7, 8, 9:		->	Nat	
_ . _:	Nat, Nat	->	Nat	

2. **sequence_SDT**: enrichment of 'String' by the functions

rpush, lpush:	Element, String	->	String	(* add element to right *)
				(* left end of sequence *)
rpop, lpop:	String	->	String	(* right/left sequence with *)
				(* one element removed *)
right, left:	String	->	Element	(* right/left element *)
_lt_:	String, String	->	Bool	(* compares elementwise *)

formal operation:
_lt_: Element, Element -> FBool

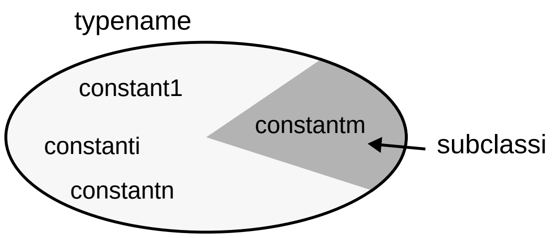
The syntax of the terms of the structured data types that can be used in the specification can be found in the values definitions of the structured data types.

ENUMERATION

The enumeration type is used to introduce a finite set of constants. The constants of an enumeration type can be used directly by their name. The constants are ordered by the sequence in which they are declared. The first one is the least. Constants are used in a specification

- as **basic building elements**, which have no finer structure, like for example in the OSI environment the informations about the kind of connections (e.g. negotiated or not) or the result of service invocations (e.g. ok or failure reason)
- as **abstractions of complex data structures**. In the initial stage of specification evolution, the internal structure of data elements is not important. The constants are later transformed into complex types in implementation directed specifications. Furthermore constants can be used instead of complex data structures to get clearer simulations of process behaviour.

Constants can be grouped into subclasses of a type. For each subclass an additional predicate is specified. These predicates have the form ‘is_subclassi(x)’, where x is a variable or a term.



Enumeration Scheme

enumtype	typename	
is		
	|{ constant1, constant2, …, constantn }|	
subclass	subclass1 |{ constantj, …, constanti }|	(* OPTIONAL *)
	subclass2 |{ constantl, …, constantk }|	(* OPTIONAL *)
	…	
	subclassn |{ constantm, …, constantm }|	(* OPTIONAL *)
endtype		

Types

typename

Library Types

NaturalNumber (Import)

Sorts

typename

Operations

constant1:		->	typename	(* constructors *)
constant2:		->	typename	
…				
constantn:		->	typename	
next:	typename	->	typename	(* successor-operation *)
min_typename:		->	typename	(* minimum&maximum *)
max_typename:		->	typename	(* of enumeration *)
is_constant1:	typename	->	Bool	(* predicates *)
is_constant2:	typename	->	Bool	
…				
is_constantn:	typename	->	Bool	
_eq_:	typename, typename	->	Bool	(* equality & order rel’s *)
_ne_:	typename, typename	->	Bool	
_lt_:	typename, typename	->	Bool	
_le_:	typename, typename	->	Bool	
_gt_:	typename, typename	->	Bool	
_ge_:	typename, typename	->	Bool	
is_subclass1:	typename	->	Bool	(* subclass predicates *)
is_subclass2:	typename	->	Bool	(* constant is in *)
…,				(* subclass
is_subclassn:	typename	->	Bool	

Auxiliary Operations:

h:	typename	->	Nat	(* mapping of constants *)
				(* on Nat for order rel’s *)

UNION

Union types construct a new data type as the union of given data types (called element types). In contrast to the type combinations of standard LOTOS, they generate a new sort that collects the data objects of the unified types. They also provide the necessary operations to manage the union like test the type of an element, select an element and set the value of an element. Usual applications of union types can be found in the field of specifying OSI-Communications, where the protocol services are grouped according to their different functionalities.

LOTOS does not include hierarchical sorts / classes in its basic mathematical model. An example of such sort hierarchies is the specification of natural numbers as a sub sort of integers. The integer sort is assignment compatible with the natural numbers, which means that a natural value can be assigned to an integer variable. LOTOS can just simulate hierarchical sorts by providing type conversion functions and predicates to test the subsort membership of values. This is done by union types.

Union types could be seen as superclasses of other types. A superclass unifies data objects that share some properties. Within a union of objects, the objects can be grouped into subclasses.

The operations that can be performed on unions are the selection of union elements and the setting of the values of union elements. It is possible to test whether a union component is of a certain type. The conceptual framework provides equality and type testing predicates and order relations on unions.

Union Scheme

```
uniontype   typename
is          import_type1, import_type2, …, import_typen
            |{   constructor1:    import_sort1,
                 constructor2:    import_sort2,
                 …,
                 constructorn:    import_sortn }|
subclass    subclass1 |{ constructorj, …, constructori }|       (* OPTIONAL        *)
            subclass2 |{ constructorl, …, constructork }|       (* OPTIONAL        *)
            …
            subclassn |{ constructorm, …, constructorm }|       (* OPTIONAL        *)
endtype
```

Types

typename

Library Types

Boolean (Import)

Sorts

typename

Operations

```
constructor1:      sort1                ->  typename    (* constructors:        *)
constructor2:      sort2                ->  typename    (* map import_sort to   *)
…                                                       (* union_sort           *)
constructorn:      sortn                ->  typename

is_constructor1:   typename             ->  Bool        (* predicates           *)
is_constructor2:   typename             ->  Bool
…
is_constructorn:   typename             ->  Bool

get_constructor1:  typename             ->  sort1       (* selectors:           *)
get_constructor2:  typename             ->  sort2       (* converts union_sort   *)
…                                                       (* back to import_sort  *)
get_constructorn:  typename             ->  sortn

_eq_:              typename, typename   ->  Bool        (* equality             *)
_ne_:              typename, typename   ->  Bool
_lt_:              typename, typename   ->  Bool        (* lexicographical order *)

is_subclass1:      typename             ->  Bool        (* subclass predicates: *)
is_subclass2:      typename             ->  Bool        (* term has one of the  *)
…                                                       (* sorts described by   *)
is_subclassn:      typename             ->  Bool        (* the constructor set  *)
                                                        (* of that subclass     *)
```

SEQUENCE

Sequences are, like arrays, ordered sequences of elements where all elements have the same sort. All components of a sequence can be read but only the outermost component can be modified. The modification of inner components is only possible by first removing one by one all components before the component can be accessed.

Sequences are used in LOTOS specifications, in case the number of components is not fixed from the beginning. An example for sequences in the OSI range are the sequences of acquaintances.

The operations on sequences are append elements to the sequence, select its head or its tail, select the *n*th element of a sequence concatenate two sequences and get the length of a sequence. The conceptual framework provides equality and type testing predicates and order relations on sequences.

Sequence Scheme

seqtype	typename	
is	import_type	
elements	sort	
values	constant1 = |(expressioni, …, expressionj)|;	(* OPTIONAL *)
	constant2 = |(expressionk, …, expressionl)|;	(* OPTIONAL *)
	…	
	constantn = |(expressionm, …, expressionn)|;	(* OPTIONAL *)
endtype		

Types

typename

Auxiliary Types:
typename0, typename00

Library Types

sequence_SDT (Actualized and Renamed)

Sorts

typename

Operations

<>:		->	typename	(* constructors *)
_+_:	typename, sort	->	typename	
_+_:	sort, typename	->	typename	
typename:	sort	->	typename	(* seq. of only one elem *)
rpush:	sort, typename	->	typename	(* add element to right/ *)
lpush:	sort, typename	->	typename	(* left end of sequence *)
rpop:	typename	->	typename	(* right/left sequence *)
lpop:	typename	->	typename	(* with one element *)
				(* removed *)
right:	typename	->	sort	(* right/left element *)
left:	typename	->	sort	
_++_:	typename, typename	->	typename	(* seq. concatenation *)
Length:	typename	->	Nat	
Reverse:	typename	->	typename	
_eq_:	typename, typename	->	Bool	(* equality *)
_ne_:	typename, typename	->	Bool	
_lt_:	typename, typename	->	Bool	(* lexicographical order *)
constant1:		->	typename	(* constants for seq *)
constant2:		->	typename	(* values *)
…				
constantn:		->	typename	

SET

Sets are collections of elements where all elements have the same sort. Unlike sequences, sets are not ordered.

The usual operations on sets the test of membership for values, the union and intersection of sets and the test if one set is a subset of another one.

Set Scheme

settype	typename	
is	import_type	
elements	sort	
values	constant1 = |{ expressioni, …, expressionj }|;	(* OPTIONAL *)
	constant2 = |{ expressionk, …, expressionl }|;	(* OPTIONAL *)
	…	
	constantn = |{ expressionm, …, expressionn }|;	(* OPTIONAL *)
endtype		

Types

typename

Auxiliary Types:
typename0, typename00

Library Types

Set (Actualized and Renamed)

Sorts

typename

Operations

{}:		->	typename	(* constructors *)
Insert:	sort, typename	->	typename	(* element in set, but *)
				(* only if it is new *)
Remove:	sort, typename	->	typename	(* remove element *)
IsIn:	sort, typename	->	Bool	(* tests whether an *)
NotIn:	sort, typename	->	Bool	(* element is in a set *)
Union:	typename, typename	->	typename	(* union of 2 sets *)
Ints:	typename, typename	->	typename	(* intersection of 2 sets *)
Minus:	typename, typename	->	typename	(* substraction of 2 sets *)
_eq_:	typename, typename	->	Bool	(* equality *)
_ne_:	typename, typename	->	Bool	
_lt_:	typename, typename	->	Bool	(* lexicographical order *)
Includes:	typename, typename	->	Bool	(* set1 ⊇ set2 *)
IsSubsetOf:	typename, typename	->	Bool	(* set1 ⊆ set2 *)
Card:	typename	->	Nat	(* # of elements in a set *)
constant1:		->	typename	(* constants for set *)
constant2:		->	typename	(* values *)
…				
constantn:		->	typename	

Auxiliary Operations:

Insert_1:	sort, typename	->	typename	(* Implements a set as a *)
				(* sorted sequence *)

ARRAY

An array is a finite sequence of fixed length with elements over the same type. The single elements of an array are accessed by an index value, specifying the number of the desired element in the array. The number of elements of an array is specified by the range of the index values. Arrays can be nested. The depth of the nesting specifies the dimension of the array, which is also called an *n*-dimensional array.

Array types are used in the specification for mathematical applications, like matrixes, as coordinates of geometrical data elements, for the encoding of characters, for the control information of low level devices, etc.

The operations that can be performed on arrays are the selection of array elements and the setting of the values of array elements. The conceptual framework provides equality and order relations on arrays.

The index range '[n..m]' has a lower limit 'n' and an upper limit 'm'. The number of array elements is 'm-n+1'. The lower border must always be less or equal to the upper one. The n-dimensional array is specified by n index ranges ('n..m').

Array Scheme

arraytype	typename	
is	import_type [n .. m, … , q .. r]	
elements	sort	
values	constant1 = |⟨ expressioni, …, expressionj ⟩|;	(* OPTIONAL *)
	constant2 = |⟨ expressionk, …, expressionl ⟩|;	(* OPTIONAL *)
	…	
	constantn = |⟨ expressionm, …, expressionn ⟩|;	(* OPTIONAL *)
endtype		

‘expression’ could be any term; it could be a regular ACT ONE term, a structured data type term and especially also an array term. The only rule is that the array constant must correspond with its definition, i.e. the nesting and the number of elements must be consistent with the range definition. The last subrange specifies the range of the innermost array. An example: an array with the nesting level one has a range [1 .. 2, 1 .. 4] with elementsort ‘nat’, and could have this value: |⟨ |⟨2, 4, 6, 1⟩|, |⟨5, 2, 5, 2⟩| ⟩|

Types

typename

Auxiliary Types: typename{_sub}*_sub, typename{_sub}*0, the number of ‘_sub’ is equal to the nesting level
The types ‘typename{_sub}*_sub’ define the arrays of the respective nesting with select and set operations and equality predicates. The types ‘typename{_sub}*0’ contain the definitions of select and set operations with more than one index.

Library Types

DecimalNaturalNumber

Sorts

typename

Auxiliary Sorts: typename{_sub}$^{+}$

Operations

array:	typename_sub_sub*, ..., typename_sub_sub*	->	typename_sub*	(* constructor *)
array:	sort, ..., sort	->	typename_sub*	(* for the depest nesting *)
nth:	typename, Nat, ..., Nat	->	sort	(* element selector *)
setn:	typename, Nat, ..., Nat, sort	->	typename	(* set value of array-elem *)
_eq_:	typename, typename	->	Bool	(* equality *)
_ne_:	typename, typename	->	Bool	
_lt_:	typename, typename	->	Bool	(* lexicographical order *)
constant1:		->	typename	(* constants for array *)
constant2:		->	typename	(* values *)
...				
constantn:		->	typename	

Auxiliary Operations:

nth:	typename_sub^{+}, Nat	->	typename_sub_sub^{+}
setn:	typename_sub^{+}, Nat, typename_sub_sub^{+}	->	typename_sub^{+}
_eq_:	typename_sub^{+}, typename_sub^{+}	->	Bool
_ne_:	typename_sub^{+}, typename_sub^{+}	->	Bool
_lt_:	typename_sub^{+}, typename_sub^{+}	->	Bool
nth:	typename_sub^{+}, Nat, ..., Nat	->	sort
setn:	typename_sub^{+}, Nat, ..., Nat, sort	->	typename_sub^{+}

RECORD

A records is a collection of a fixed number of components that may be written down in any order. Each record component has its own type. The components of a record are accessed by field selectors that are associated to them. Records can have optional components, this means that these components might be omitted in the definition of a record. A field can have a default value, which is used in case a component is omitted in the definition.

Records are used in a wide range of applications. They are useful in all applications where multiple informations of different types are associated to one element. In OSI specifications, the records are used for service primitives. An other example of records are files in a data base.

Operations on records are the selection of record components and the setting of the values of record components. It is possible to test whether a component of a record is set. This test is useful before the selection of a component, because the operation is only defined if the record component is set. Another available predicate gives information, about the completeness of a record, i.e. tests whether all mandatory components are set. The conceptual framework provides equality and order relations on records.

Record Scheme

```
recordtype  typename
is          import_typen, import_type2, …, import_type1
fields      selector1:   sort1,
            selector2:   sort2 optional,
            …,
            selectorn:   sortn default = typename |{  selectorg(expressiong), …,
                                                      selectorh(expressionh)}|
values                                                       (* OPTIONAL          *)
            constant1 = typename |{selectori(expressioni), …, selectorj(expressionj)}|;
            constant2 = typename |{selectork(expressionk), …, selectorl(expressionl)}|;
            …
            constantn = typename |{selectorm(expressionm), …, selectorn(expressionn)}|;
endtype
```

Types

typename

Auxiliary Types (see next page for their definitions):
typename_sel, typename_component, typename000, typename00, typename0, typename_set_interface

Library Types

Boolean, NaturalNumber, Set

Sorts

typename

Operations

```
init_typename:                        -> typename    (* initializes a record with the   *)
                                                     (* default setting                 *)
complete:        typename             -> Bool        (* tests whether all non optional  *)
                                                     (* fields are set                  *)
selector1:       typename             -> sort1       (* selects the value of a record   *)
selector2:       typename             -> sort2       (* field                           *)
…
selectorn:       typename             -> sortn

set_selector1:   typename, sort1      -> typename    (* sets the value of a field       *)
set_selector2:   typename, sort2      -> typename
…
set_selectorn:   typename, sortn      -> typename

_eq_:            typename, typename   -> Bool        (* equality*)
_ne_:            typename, typename   -> Bool
_lt_:            typename, typename   -> Bool        (* lexicographical order*)

constant1:                            -> typename    (* constants forrecord             *)
constant2:                            -> typename    (* values                          *)
…
constantn:                            -> typename
```

Dependencies of Generated Types

Auxiliary Types, Sorts, and Operations:

```
enumtype    typename_sel
is
            |{ selector1, selector2, …, selectorn }|
endtype

uniontype   typename_component
is          Boolean, typename_sel, import_typen, import_type2, import_type1
            |{  selector1:      sort1,
                selector2:      sort2,
                …,
                selectorn:      sortn }|
endtype

settype     typename00
is          import_type
elements    sort
endtype

type        typename0
is          typename00
renamedby
sortnames   typename for typename 00
endtype

type        typename_set_interface
is          typename0
opns        sel:        typename_component            ->  typename_sel
            add:        typename_component, typename  ->  typename
            remove:     typename_sel, typename        ->  typename
            _overlay_:  typename, typename            ->  typename
            is_set:     typename_sel, typename        ->  Bool
            get:        typename_sel, typename        ->  typename_component
endtype
```

Formal Syntax Definition of Structured Data Types

In this chapter the syntax of structurd data type specifications is defined. The syntax definition shall be included into the syntax definition of the LOTOS standard [ISO:8807]. The non-terminals ‘data-type-definition’ and ‘term-expression’ are redefined.

DATA TYPE

A.1 data-type-definition = type-symbol type-identifier is-symbol
p-expression end-type-symbol
| structured-data-type-definition
| library-declaration.

STRUCTURED DATA TYPE

A.2 structured-data-type-definition = enumeration-type-definition
| union-type-definition
| sequence-type-definition
| set-type-definition
| array-type-definition
| record-type-definition.

A.3 enumeration-type-definition = enumeration-type-symbol
sort-and-type-identifier is-symbol
open-set-symbol
[operation-identifier-list] close-set-symbol
[subclass-expression] end-type-symbol.

A.4 union-type-definition = union-type-symbol sort-and-type-identifier
is-symbol type-union open-set-symbol
union-projection-list close-set-symbol
[subclass-expression] end-type-symbol.

<table>
<tr><td></td><td>A.5</td><td>sequence-type-definition =</td><td>sequence-type-symbol sort-and-type-identifier
is-symbol type-identifier
elements-symbol sort-identifier
[initial-setting-expression] end-type-symbol.</td></tr>
<tr><td></td><td>A.6</td><td>set-type-definition =</td><td>set-type-symbol sort-and-type-identifier
is-symbol type-identifier
elements-symbol sort-identifier
[initial-setting-expression] end-type-symbol.</td></tr>
<tr><td></td><td>A.7</td><td>array-type-definition =</td><td>array-type-symbol sort-and-type-identifier
is-symbol type-identifier
elements-symbol sort-identifier
open-bracket-symbol index-list
close-bracket-symbol
[initial-settings-expression] end-type-symbol.</td></tr>
<tr><td></td><td>A.8</td><td>record-type-definition =</td><td>record-type-symbol sort-and-type-identifier
is-symbol type-union
fields-symbol [record-projection-list]
[initial-settings-expression] end-type-symbol.</td></tr>
<tr><td>SUBCLASS</td><td>A.9</td><td>subclass-expression =</td><td>subclass-symbol subclass-list.</td></tr>
<tr><td></td><td>A.10</td><td>subclass-list =</td><td>subclass [subclass-list].</td></tr>
<tr><td></td><td>A.11</td><td>subclass =</td><td>operation-identifier open-set-symbol
[operation-identifier-list] close-set-symbol.</td></tr>
<tr><td>ARRAY INDEX</td><td>A.12</td><td>index-list =</td><td>number range-symbol number
[comma-symbol index-list].</td></tr>
<tr><td></td><td>A.13</td><td>number =</td><td>"1" | "2" | "3" | " 4" | "4" | "5" | "6" | "7" | "8" | "9" { digit }.</td></tr>
<tr><td>INITIAL SETTING</td><td>A.14</td><td>initial-settings-expression =</td><td>values-symbol value-definition-list.</td></tr>
<tr><td></td><td>A.15</td><td>value-definition-list =</td><td>value-definition [value-definition-list].</td></tr>
<tr><td></td><td>A.16</td><td>value-definition =</td><td>operation-identifier equal-symbol value-expression.</td></tr>
<tr><td>RECORD PROJECTION</td><td>A.17</td><td>record-projection-list =</td><td>projection [feature-expression]
[comma-symbol record-projection-list].</td></tr>
<tr><td>UNION PROJECTION</td><td>A.18</td><td>union-projection-list =</td><td>projection [comma-symbol union-projection-list].</td></tr>
<tr><td></td><td>A.19</td><td>projection =</td><td>operation-identifier colon-symbol
sort-identifier.</td></tr>
<tr><td></td><td>A.20</td><td>feature-expression =</td><td>default-symbol equal-symbol value-expression
| optional-symbol.</td></tr>
<tr><td>AUXILIARIES</td><td>A.21</td><td>sort-and-type-identifier =</td><td>identifier.</td></tr>
<tr><td></td><td>A.22</td><td>operation-identifier-list =</td><td>operation-identifier
[comma-symbol operation-identifier-list].</td></tr>
</table>

<table>
<tr><td>SHORT VALUE EXPRESSIONS</td><td>A.23</td><td>term-expression =</td><td>value-identifier
| operation-identifier [value-expression-list]
| open-parenthesis-symbol value-expression close-parenthesis-symbol
| set-expression
| sequence-expression
| array-expression
| record-expression.</td></tr>
<tr><td></td><td>A.24</td><td>set-expression =</td><td>open-set-symbol [value-expression-list] close-set-symbol.</td></tr>
<tr><td></td><td>A.25</td><td>sequence-expression =</td><td>open-sequence-symbol [value-expression-list] close-sequence-symbol.</td></tr>
<tr><td></td><td>A.26</td><td>array-expression =</td><td>open-array-expression value-expression-list close-array-symbol.</td></tr>
<tr><td></td><td>A.27</td><td>record-expression =</td><td>operation-identifier open-set-symbol record-assignment-list close-set-symbol.</td></tr>
<tr><td></td><td>A.28</td><td>record-assignment-list =</td><td>operation-identifier open-parenthesis-symbol term-expression close-parenthesis-symbol [comma-symbol record-assignment-list].</td></tr>
<tr><td>WORD SYMBOLS</td><td>A.29</td><td>enumeration-type-symbol =</td><td>"enumtype".</td></tr>
<tr><td></td><td>A.30</td><td>union-type-symbol =</td><td>"uniontype".</td></tr>
<tr><td></td><td>A.31</td><td>sequence-type-symbol =</td><td>"seqtype".</td></tr>
<tr><td></td><td>A.32</td><td>set-type-definition =</td><td>"settype".</td></tr>
<tr><td></td><td>A.33</td><td>array-type-symbol =</td><td>"arraytype".</td></tr>
<tr><td></td><td>A.34</td><td>record-type-symbol =</td><td>"recordtype".</td></tr>
<tr><td></td><td>A.35</td><td>subclass-symbol =</td><td>"subclass".</td></tr>
<tr><td></td><td>A.36</td><td>fields-symbol =</td><td>“fields”.</td></tr>
<tr><td></td><td>A.37</td><td>optional-symbol =</td><td>"optional".</td></tr>
<tr><td></td><td>A.38</td><td>default-symbol =</td><td>"default".</td></tr>
<tr><td>SPECIAL SYMBOLS</td><td>A.39</td><td>open-set-symbol =</td><td>“|{“.</td></tr>
<tr><td></td><td>A.40</td><td>close-set-symbol =</td><td>“|}”.</td></tr>
<tr><td></td><td>A.41</td><td>open-sequence-symbol =</td><td>“|(“.</td></tr>
<tr><td></td><td>A.42</td><td>close-sequence-symbol =</td><td>“)|”.</td></tr>
<tr><td></td><td>A.43</td><td>open-array-symbol =</td><td>“|<“.</td></tr>
<tr><td></td><td>A.44</td><td>close-array-symbol =</td><td>“>|”.</td></tr>
<tr><td></td><td>A.45</td><td>range-symbol =</td><td>"..".</td></tr>
<tr><td></td><td>A.46</td><td>reverse-arrow-symbol =</td><td>"<-".</td></tr>
</table>

Installation of the Structured Data Type Translation Tool 'pao2ao'

FILES

'pao2ao' stands for 'powerfull ACT ONE to ACT ONE'. The following files contain the source code for the pao2ao tool:

makefile	makefile to create translator
trans_main.c	main function for translator
trans_main.h	type declarations of variables and functions
trans_l.l	lex file (token definitionen)
trans_y.y	yacc file (syntax definition)
trans_decl.k	kimwitu tree declarations
trans_array.k	rewrite functions for array types and expressions
trans_lib.k	functions to create the library types
trans_record.k	rewrite functions for record types and expressions
trans_rw.k	general rewrite function pool
trans_unp.k	unparse functions to create text file of the kimwitu tree
trans_graph.k	generation of graphical annnotations for visualization with DEMON

MAKEFILE

The makefile shows which variables have to be defined and how the executable will be build:

```
IT =            trans_
KFILES  =       ${IT}decl.k ${IT}rw.k ${IT}lib.k ${IT}unp.k ${IT}record.k\
                ${IT}array.k ${IT}graph.k
CC =            cc
YOURFILES =     ${KFILES} ${IT}y.y ${IT}l.l ${IT}main.c
ALLOBJS =       k.o rk.o csgiok.o  unpk.o\
                ${KFILES:k=o} ${IT}y.o  ${IT}l.o ${IT}main.o
GENERATED_C =   k.c rk.c csgiok.c  unpk.c ${KFILES:k=c}
GENERATED_H =   k.h  rk.h  csgiok.h  unpk.h ${KFILES:k=h}
```

```
GENERATED_BY_KC = ${GENERATED_C} ${GENERATED_H}
YACC =                bison
YFLAGS =              -dyv
KC =                  /net/jupiter/usr85/SEM/kimwitu/kc-distr.V3_8/src/Gen/kc
GENERATED_LN =        ${IT}decl.ln ${IT}rw.ln ${IT}lib.ln ${IT}unp.ln\
                      ${IT}record.ln ${IT}array.ln ${IT}graph.ln\
                      k.ln rk.ln csgiok.ln unpk.ln ${IT}main.ln ${IT}y.ln ${IT}l.ln

new_${IT}:            ${ALLOBJS}
                      ${CC} ${CFLAGS} ${ALLOBJS}  -ll -o  $@
${GENERATED_BY_KC}:     kctimestamp
kctimestamp:          ${KFILES}
                      ${KC} ${KFILES}; touch kctimestamp
${ALLOBJS}:           k.h
${IT}main.o ${IT}l.o: x.tab.h
${IT}main.o ${KFILES:k=0}: ${KFILES:k=h}
${IT}main.o rk.o:     rk.h
${IT}main.o csgiok.o: csgiok.h

${IT}main.o unpk.o:   unpk.h

x.tab.h:              y.tab.h
                      -cmp -s x.tab.h  y.tab.h || cp y.tab.h x.tab.h

lint:                 ${GENERATED_LN}
                      -@ lint -u -n -q -v ${CFLAGS} ${GENERATED_LN} 2>&1 |\
                      sed -e '/warning:/d' -e '/malloc[,:]/d' -e '/printf[,:]/d'\
                          -e '/scanf[,:]/d' -e '/^free[,:]/d'

.c.ln:
                      lint -u -n -q -v -i $< 2>&1 | sed '/warning:/d'
```

Annex
Signatures of Library Types

The following signatures are provided for those that do not use the lite and TOPO tool set. It provides the sorts, operations and equations of the three library types that are used.

```
(*  This library is the IS 8807 Annex A standard libray, modified by LotosPhere WP2, and
    recommended for use with tools. the changes are in types Set and SetElement.
    Feb 1991
*)
```

NaturalNumber

```
type    NaturalNumber
is      Boolean
sorts   Nat
opns    0                                       :             -> Nat
        Succ                                    : Nat         -> Nat
        _+_, _*_, _**_                          : Nat, Nat    -> Nat
        _eq_, _ne_, _lt_, _le_, _ge_, _gt_      : Nat, Nat    -> Bool
eqns    forall m, n : Nat
ofsort  Nat
        m + 0 = m;
        m + Succ(n) = Succ(m) + n;
        m * 0 = 0;
        m * Succ(n) = m + (m * n);
        m ** 0 = Succ(0);
        m ** Succ(n) = m * (m ** n);
ofsort  Bool
        0 eq 0 = true;                  0 eq Succ(m) = false;
        Succ(m) eq 0 = false;           Succ(m) eq Succ(n) = m eq n;
        m ne n = not(m eq n);
        0 lt 0 = false;                 0 lt Succ(n) = true;
        Succ(n) lt 0 = false;           Succ(m) lt Succ(n) = m lt n;
        m le n = m lt n or (m eq n);
        m ge n = not(m lt n);
        m gt n = not(m le n);
endtype
```

String

```
type     String
is       Boolean, Element, NaturalNumber
sorts    String
opns     String                                  : Element           -> String
         _+_                                     : Element, String   -> String
         _+_                                     : String, Element   -> String
         _++_                                    : String, String    -> String
         Reverse                                 : String            -> String
         Length                                  : String            -> Nat
         <>                                      :                   -> String
         _eq_, _ne_                              : String, String    -> Bool
eqns     forall s, t : String, x, y : Element
ofsort   String
         String(x) + y = x + String(y);
         x + s + y = x + (s + y);
         String(x) ++ s = x + s;
         x + s ++ t = x + (s ++ t);
         Reverse(String(x)) = String(x);
         Reverse(x + s) = Reverse(s) + x;
         String(x) = x + <>;
         <> + x = x + <>;
         <> ++ s = s;
         (* new equation *)
         s ++ <> = s;
         (* end new equation *)
         Reverse(<>) = <>;
ofsort   Nat
         Length(String(x)) = Succ(0);
         Length(x + s) = Succ(Length(s));
         Length(<>) = 0;
ofsort   Bool
         <> eq <> = true;
         <> eq (x + s) = false;
         x + s eq <> = false;
         x eq y => x + s eq (y + t) = s eq t;
         x ne y => x + s eq (y + t) = false;
         s ne t = not(s eq t);
endtype
```

Set

```
type    Set
is      SetElement, Boolean, NaturalNumber
sorts   Set
opns    {}                                          :               -> Set
        Insert, Remove, Insert_1                    : Element, Set  -> Set
        _IsIn_, _NotIn_                             : Element, Set  -> Bool
        _Union_, _Ints_, _Minus_                    : Set, Set      -> Set
        _eq_, _ne_, _lt_, _Includes_, _IsSubsetOf_  : Set, Set      -> Bool
        Card                                        : Set           -> Nat
eqns    forall x, y : Element, s, t : Set
ofsort  Set
        Insert(x, {}) = Insert_1(x, {});
        x lt y => Insert(x, Insert_1(y, s)) = Insert_1(x, Insert_1(y, s));
        Insert(x, Insert_1(x, s)) = Insert_1(x, s);
        y lt x => Insert(x, Insert_1(y, s)) = Insert_1(y, Insert(x, s));
        Remove(x, {}) = {};
        Remove(x, Insert_1(x, s)) = s;
        x lt y => Remove(x, Insert_1(y, s)) = Insert_1(y, s);
        y lt x => Remove(x, Insert_1(y, s)) = Insert_1(y, Remove(x, s));
        {} Union s = s;
        Insert_1(x, s) Union {} = Insert_1(x, s);
        x lt y => Insert_1(x, s) Union Insert_1(y, t) = Insert_1(x, s Union Insert_1(y, t));
        y lt x => Insert_1(x, s) Union Insert_1(y, t) = Insert_1(y, Insert_1(x, s) Uni on t);
        Insert_1(x, s) Union Insert_1(x, t) = Insert_1(x, s Union t);
        {} Ints s = {};
        Insert_1(x, s) Ints {} = {};
        Insert_1(x, s) Ints Insert_1(x, t) = Insert_1(x, s Ints t);
        x lt y => Insert_1(x, s) Ints Insert_1(y, t) = s Ints Insert_1(y, t);
        y lt x => Insert_1(x, s) Ints Insert_1(y, t) = Insert_1(x, s) Ints t;
        s Minus {} = s;
        s Minus Insert_1(x, t) = Remove(x, s) Minus t;
ofsort  Bool
        x IsIn {} = false;
        x IsIn Insert_1(x, s) = true;
        y lt x => x IsIn Insert_1(y, s) = x IsIn s;
        x lt y => x IsIn Insert_1(y, s) = false;
        x NotIn s = not(x IsIn s);
        s Includes {} = true;
        s Includes Insert_1(x, t) = x IsIn s and (s Includes t);
        s IsSubsetOf t = t Includes s;
        {} eq {} = true;
        {} eq Insert_1(x, s) = false;
        Insert_1(x, s) eq {} = false;
        x eq y => Insert_1(x, s) eq Insert_1(y, t) = s eq t;
        x ne y => Insert_1(x, s) eq Insert_1(y, t) = false;
        {} lt {} = false;
        {} lt Insert_1(x, s) = true;
        Insert_1(x, s) lt {} = false;
        x lt y => Insert_1(x, s) lt Insert_1(y, t) = true;
        Insert_1(x, s) lt Insert_1(x, t) = s lt t;
        y lt x => Insert_1(x, s) lt Insert_1(y, t) = false;
        s ne t = not(s eq t);
ofsort  Nat
        Card({}) = 0;
        Card(Insert_1(x, s)) = Succ(Card(s));
endtype
```

Auxiliary Types

```
type        Boolean
is
sorts       Bool
opns        true, false                                              :                    -> Bool
            not                                                      : Bool               -> Bool
            _and_, _or_, _xor_, _implies_, _iff_, _eq_, _ne_         : Bool, Bool         -> Bool
eqns        forall x, y : Bool
ofsort      Bool
            not(true) = false;
            not(false) = true;
            x and true = x;
            x and false = false;
            x or true = true;
            x or false = x;
            x xor y = x and not(y) or (y and not(x));
            x implies y = y or not(x);
            x iff y = x implies y and (y implies x);
            x eq y = x iff y;
            x ne y = x xor y;
endtype

type        FBoolean
is
formalsorts     FBool
formalopns      true                                                 :                    -> FBool
                not                                                  : FBool              -> FBool
formaleqns      forall x : FBool
ofsort          FBool
                not(not(x)) = x;
endtype

type        Element
is          FBoolean
formalsorts     Element
formalopns      _eq_, _ne_                                           : Element, Element -> FBool
formaleqns      forall x, y : Element
ofsort          Element
                x eq y => x = y;
ofsort          FBool
                x = y => x eq y = true;
                x ne y = not(x eq y);
endtype

type        SetElement
is          Element
formalopns      _lt_                                                 : Element, Element -> FBool
endtype
```