CHILL 2000

JÜRGEN F.H. WINKLER

Jürgen F.H. Winkler (57) has since 1993 been a full professor of Computer Science at the Friedrich Schiller University in Jena, Germany. His main interests are program correctness, object-orientation, programming languages and their implementation. Before joining the university he was with the corporate research of Siemens AG in Munich. Among other projects he has been involved in the definition and implementation of Object-CHILL, a forerunner of CHILL 2000, and with the Siemens Ada compiler. He also founded the "International Workshop on Software Configuration Management". Dr. Winkler received his PhD and his Diploma, both in Computer Science, from the University of Karlsruhe, Germany.

jwinkler@acm.org

CHILL is a programming language mainly used in the area of telecom systems. This paper gives an overview of the language elements of CHILL and reports in more detail on new language elements which have been added recently, especially object-orientation and genericity.

1 Introduction

This paper gives a tutorial overview of CHILL, the ITU-T Programming Language [1]. CHILL is an acronym with the original long form "CCITT High Level Language", which reflects the fact that ITU-T was formerly called CCITT.

CHILL has been originally developed in CCITT during the period 1975 – 1983. After this, it has been continuously updated and used for the development of many telecom systems around the world [2]. This paper also contains more details about the history and application of CHILL.

Today CHILL is a modern object-oriented language, which also supports concurrency in an object-oriented manner. In the last Study Period (1997–2000) the following language elements have been included:

- Interfaces;
- Support of Unicode;
- Friend-procedures;
- Overloading of procedures;
- Final (unmodifiable) components in objects.

In the body of the paper we give an overview of the language elements of CHILL and describe in more detail those elements of CHILL that were added more recently.

In this paper we use the typical terminology of the field of programming languages, especially for basic terms. CHILL, as many other languages, has a number of specific terms. Especially for the following terms we use the traditional terminology:

type	"mode" in CHILL
variable	"location" in CHILL
statement	"action" in CHILL

2 Language Overview

CHILL is a procedural and object-oriented language, which contains a number of elements that support the development of large programs, as they are typical for the telecom field. The following tree shows the language elements of CHILL 2000.

Data Structures
- Scalar: integer, float, characters, Boolean, enumerations, pointer, procedure type, process type, event, time;
- Composite: string, record, array, set, buffer, signal.

Sequential Programming
- Variable, constant, expression, function call;
- Assignment;
- Procedure call;
- EXIT, RESULT, RETURN, GOTO;
- Statement sequence;
- Selection statements: IF, CASE (multidimensional);
- Repetition statements: DO, WHILE, FOR.

Object-oriented Programming
- Sequential, unsynchronized object;
- Sequential, synchronized object;
- Concurrent, synchronized object;
- Interface;
- Friend.

Concurrent Programming
- Process;
- Start process;
- Communication via buffer;
- Communication via signal;
- Critical region and co-ordination with events;
- Concurrent, synchronized object.

Program Structure
- Block;
- Procedure / Function / Process;
- Object-Type / Class;
- Module / Region.

Genericity
- Generic Procedure / Process;
- Generic Module / Region;
- Generic Object Type / Class;
- Generic Interface.

Program Verification
- Precondition and postcondition for methods;
- Invariant for object type / class;
- ASSERT statement.

Box 1 contains a number of small examples for most of the elements listed above, in order to give the reader some impression of CHILL 2000 as a programming language.

Box 2 contains a comparison of CHILL2000 and Java based on the tree structure of the overview on CHILL given above. If one of the languages does not contain a certain element the corresponding entry is empty (e.g. "Genericity" in Java).

3 New Elements in CHILL 2000

During the last two study periods (1993–1996, 1997–2000) new and important language elements have been added to the language. The most important of them are:

- Object-Orientation;
- Genericity.

3.1 Object-Orientation

Object types, which are typically called classes in the area of object-orientation [3, 4], come in CHILL 2000 in four different flavors

- *Module type*
  An object (or instance) of such a type has the typical properties of a module. It has components, which can be public or internal, and it does not do any co-ordination in case of concurrent accesses to its components. With respect to concurrency module objects are passive, i.e. they do not have an own thread of control.

- *Region type*
  An object (or instance) of such a type has the typical properties of a region. It has components, which can be public or internal, and it co-ordinates concurrent accesses to its components. With respect to concurrency, region objects are passive, i.e. they do not have an own thread of control.

- *Task type*
  An object (or instance) of such a type has a similar structure as module and region objects. It has components, which can be public or internal. With respect to concurrency it has its own thread of control and it co-ordinates concurrent accesses to its components. It is therefore similar to task objects in Ada [5] and this is the reason for its name.

- *Interface type*
  An interface type defines an interface, which consists of the specification of public components. There are no objects of interface types. Interface types are typically used as base types of other object types.

Together the new object types are called moreta types, where moreta has been formed from the first letters of module, region, and task.

A common characteristic is that the definition of a non-interface moreta type (= class) consists of a specification part and a body. This separation is very useful from a software engineering point of view. The interface describes what a user (client) of the given type must know in order to use the type or its objects. The body contains the internal implementation of the components specified in the interface.

As an example we look at the definition of a stack type.

```
SYNMODE IntStackType1 = MODULE SPEC
   GRANT Push, Pop;
   Push: PROC(Elem INT IN)
         EXCEPTIONS(Overflow)  END Push;
   Pop:  PROC( ) RETURNS(INT)
         EXCEPTIONS(Underflow)  END Pop;
   SYN Length = 10_000;
   DCL StackData ARRAY (1:Length) INT,
         TopOfStack RANGE(0:Length) INIT := 0;
END IntStackType1;
```

The type IntStackType1 is defined like other types in CHILL. The keyword MODULE indicates that it is a module type and the keyword SPEC indicates that it is the specification part of this type. The procedures (methods) Push and Pop are exported and are therefore public components of IntStackType1. Length, StackData, and TopOfStack are internal components. This is an example of encapsulation and is necessary to guarantee the stack protocol.

The corresponding body contains in this case the bodies of the two procedures.

```
SYNMODE IntStackType1 = MODULE BODY
   Push: PROC(Elem INT IN) EXCEPTIONS(Overflow)
      IF TopOfStack = Length
      THEN CAUSE Overflow;
      ELSE  TopOfStack +:= 1;
            StackData(TopOfStack) := Elem;
      FI;
   END Push;
   Pop: PROC( ) RETURNS(INT) EXCEPTIONS(Underflow)
      IF TopOfStack = 0
      THEN CAUSE Underflow;
      ELSE  RESULT StackData(TopOfStack);
            TopOfStack -:= 1;
      FI;
   END Pop;
END IntStackType1;
```

Objects of the type IntStackType1 are declared in the same way as variables for the traditional types. The manipulation of these variables is done in the typical style of object-orientation.

```
DCL Stack1, Stack2 IntStackType1;
Stack1.Push(10);
Stack2.Push(100);
. . . Stack1.Pop(). . .
```

Stack1 and Stack2 are adequate for sequential programming. It is now quite easy to define a stack type CIntStackType1 whose objects co-ordinate concurrent calls of their methods. In CHILL there are two ways to accomplish this:

a) change the keyword MODULE into REGION

```
SYNMODE CIntStackType1 = REGION SPEC
     /* same as before */
END CIntStackType1;
```

And analogously for the body.

b) derive the type CIntStackType1 from the existing type IntStack-Type1.

```
SYNMODE CIntStackType1 = REGION SPEC
    BASED_ON IntStackType1
END CIntStackType1;
```

Since there are different kinds of object types there exist several possibilities for the derivation of types from base types.

- A class can be directly derived from one base class (single inheritance between classes);
- A class can be directly derived by combining an arbitrary number of base interface types (multiple inheritance between interfaces and classes);
- An interface type can be derived from an arbitrary number of base interface types (multiple inheritance between interfaces).

These conditions can be summed up to the rule that CHILL uses single inheritance for classes and multiple inheritance for interfaces.

Since module, region and task differ in their properties, the following derivation constraints have to be observed:

Base type:	Permissible derived type:
module	module, region, task
region	region
task	task

The derivation mechanism of object-orientation is a mechanism for the realization of structural polymorphism. A derived type DT and its objects contain the components inherited and possibly additional components defined in DT. As an example, we define a stack type IntStackType2, which is derived from IntStackType1 but contains the additional function Top() (INT) which returns the value of the topmost element, but does not change the contents of the stack.

```
SYNMODE IntStackType2 = MODULE SPEC
                        BASED_ON IntStackType1
   GRANT Top;
   Top: PROC( ) RETURNS(INT) EXCEPTIONS(Underflow)
   END Top;
END IntStackType2;

SYNMODE IntStackType2 = MODULE BODY
                        BASED_ON IntStackType1
   Top: PROC( ) RETURNS(INT) EXCEPTIONS(Underflow)
      IF TopOfStack = 0
      THEN CAUSE Underflow;
      ELSE RETURN StackData(TopOfStack);
      FI;
   END Top;
END IntStackType2;
```

3.2 Genericity

The stack is a good example to demonstrate the concept of genericity. In section 3.1 the element type of the stack is INT. If we need stacks with other element types, we have to duplicate or in general replicate the code for each new element type. From a software engineering point of view, this code replication is very unwelcome. There are two ways to try to avoid this problem.

a) Use “REF UltimateBaseType” as the element type of the stack type. If the language does not have an ultimate base type, an appropriate base type has to be used.

```
SYNMODE IntStackType3 = MODULE SPEC
    GRANT Push, Pop, ElemType;
    SYNMODE ElemType = REF UltimateBaseType;
    Push: PROC(Elem ElemType IN)
          EXCEPTIONS(Overflow) END Push;
    Pop:  PROC( ) RETURNS(ElemType)
          EXCEPTIONS(Underflow) END Pop;
    SYN Length = 10_000;
    DCL StackData ARRAY (1:Length) ElemType,
           TopOfStack RANGE(0:Length) INIT := 0;
END IntStackType3;
```

The body of IntStackType3 is essentially the same as that of IntStacktype1. The difference is in the identifiers IntStackType3 and ElemType.

The objects of IntStackType3 are now heterogeneous stacks, i.e. due to polymorphism, they may contain objects of different types.

```
DCL Stack3 IntStackType3;
Stack3.Push(new IntStackType1);
Stack3.Push(new IntStackType2);
Stack3.Push(new IntStackType3);
```

b) If we want to have homogeneous stack objects, as those of the types IntStackType1, IntStackType2, or CIntStackType1 are, genericity (or parametric polymorphism) is the right mechanism to use. A generic entity is an entity which is parameterized in a more general way than traditional procedures. In CHILL the following entities may be used as parameters of a generic entity:

values of arbitrary types;
types;
procedures and functions.

It is especially the possibility to use types as parameters which provides new possibilities for the formulation of programs.

The use of genericity is typically done in two steps:

i) define a generic entity, i.e. an entity which has formal generic parameters. Such a generic entity is a template for more specific entities.

ii) define an instantiation of the generic template by providing actual generic parameters for the formal ones.

A generic stack type may now look as follows:

```
GenericStackTemplate1:
    GENERIC MODE ElemType = ANY_ASSIGN;
MODULE SPEC
    GRANT Push, Pop;
    Push: PROC(Elem ElemType IN)
          EXCEPTIONS(Overflow) END Push;
    Pop:  PROC( ) RETURNS(ElemType)
          EXCEPTIONS(Underflow) END Pop;
    SYN Length = 10_000;
    DCL StackData ARRAY (1:Length) INT,
    TopOfStack RANGE(0:Length) INIT := 0;
END GenericStackTemplate1;
```

As for IntStackType3, the body of GenericStackTemplate1 is essentially the same as that for IntStackType1.

GenericStackTemplate1 has one formal generic parameter, ElemType, which is of the kind ANY_ASSIGN. This means that variables of type ElemType can be assigned inside the definition of GenericStackTemplate1. This property is needed in the bodies of Push and Pop. On the other hand, any type which is used as a corresponding actual generic parameter must at least support the operation of assignment. This guarantees that any legal instantiation will produce a legal type.

Using GenericStackTemplate1, we obtain non-generic stack types by instantiating the template with an actual generic parameter. If we use INT as actual generic parameter, we obtain an object type which is essentially equivalent to IntStackType1.

```
SYNMODE IntStackType4 = NEW GenericStackTemplate1
    SYNMODE ElemType = INT;
END IntStackType4;
```

If we use FLOAT as actual generic parameter we obtain a type FloatStackType whose objects can only take float values as elements.

```
SYNMODE FloatStackType = NEW GenericStackTemplate1
    SYNMODE ElemType = FLOAT;
END FloatStackType;
```

After having created two generic instantiations of the template GenericStackTemplate1 we see that with genericity the code duplication is avoided.

We see that both structural polymorphism (through inheritance) and parametric polymorphism (through genericity) are very useful mechanisms for the formulation of programs.

4 Use of CHILL in Telecom Systems

Since its birth, CHILL has been used quite widely in the world of telecommunications. Rekdal mentions about 13 companies [2], and if we account for the fact that several companies in Korea have built systems using CHILL, we can say that about 15 significant companies in the telecom field have built systems using CHILL. Since large companies as e.g. Alcatel and Siemens sell their systems all over he world, CHILL is passively used by hundreds of millions of people. In Germany for example, the conventional telephone network is essentially based on systems written in CHILL. There are mainly two systems used: EWSD from Siemens and System12 from Alcatel.

A lot more details about these aspects of CHILL are given in [2].

References

1 ITU-T. *CHILL – The ITU-T programming language).* ITU, Geneva, 1999. (Recommendation Z.200 (11/99.) (http://www.itu.int/itudoc/itu-t/approved/z/z200.html)

See also: ISO/IEC 9496:1998 CCITT high level language (CHILL). http://www.iso.ch/cate/d30537.html

2 Rekdal, K. CHILL – the international standard language for telecommunications programming. *Telektronikk*, 89 (2/3), 5–10, 1993.

3 Dahl, O, Myhrhaug, B, Nygaard, K. *Common Base Language.* Oslo, Norwegian Computing Center, 1970.

4 Goldberg, A, Robson, D. *Smalltalk-80 – The Language.* Reading, Mass., Addison Wesley, 1989. (ISBN 0-201-13688-0)

5 ISO/IEC. *Information Technology – Programming Languages – Ada.* Geneva, ISO/IEC, 1995. (ISO/IEC 8652:1995(E).)

Box 1 CHILL in Examples

This box gives a tutorial overview on the language elements of CHILL in three pieces:

Sequential programming
(types and statements)
Object-oriented programming and Genericity
Concurrent programming

Sequential Programming: Types

From a structural point of view we may distinguish between scalar types and composite types. In this overview we follow roughly this pattern.

Scalar Types

The values of scalar types are indivisible entities. Important scalar types are numbers, enumerations and references.

As is usual in computing, we distinguish integer numbers and types, and floating point numbers and types.

Integer numbers are written as usual:

```
1, 123, -450
```

Large numbers may be structured for better readability using the underscore character: `1_721_119`

We may write numbers using different bases:

```
Binary numbers:       b'1010
Octal numbers:        o'12367
Hexadecimal numbers:  -h'12ABC
```

There are predefined integer types (e.g. INT) and the user may also define his own types, especially types with specific value ranges:

```
NEWMODE line = RANGE(1:8);
        /* e.g. the lines of a chess board */
```

A variable of a given type is defined in a declaration statement:

```
DCL CurrentLine line INIT := 1;
```

Such a variable can be initialized with a specific value.

For rational numbers CHILL uses floating point types. FLOAT is a predefined type, but it is also possible to define problem specific floating point types, e.g. a type for temperature in a given range.

```
NEWMODE Temp = FLOAT(-273.15:1000.0);
```

For numbers the usual arithmetic operations are defined:

```
DCL I INT INIT := 25*25 + 17;
DCL J INT INIT := I/2;
```

The type BOOL contains the two truth values FALSE and TRUE and can be used for conditions and computations in propositional logic:

```
DCL CallFinished BOOL INIT := FALSE;
. . .
IF NOT CallFinished THEN . . .
```

Very useful are also the enumeration types, e.g.

```
NEWMODE ActionType = SET(A1, A2, A3);
NEWMODE ColorTy = SET(red, green, blue);
SYNMODE month = SET
        (jan,feb,mar,apr,may,jun,
        jul,aug,sep,oct,nov,dec);
```

Composite Types

The values of composite types consist of several components which may themselves be scalar or composite values. The composite types in CHILL are structures (records), arrays and strings, buffer and signal, sets, and objects.

Structures are heterogeneous tuples:

```
NEWMODE DateType =
        STRUCT ( day INT(1:31),
                 mo       month,
                 year     INT(1:3000) );

NEWMODE TimedActionType =
        STRUCT ( action   ActionType,
                 date     DateType );
```

The values of structures can be denoted by unlabelled or by labelled tuples:

```
DCL Today DateType INIT := [24,aug,2000];
DCL Today DateType INIT :=
      [day: 24, mo: aug, year: 2000];
```

If we want to implement a linked list of timed actions, we can use a reference type (pointer type). The values of reference types point to other values.

```
NEWMODE RefToTimedActionListType =
        REF TimedActionListType;

NEWMODE TimedActionListType =
        STRUCT(action TimedActionType,
               next   RefToTimedActionListType );

DCL TimedActionList TimedActionListType;
```

The following two assignment statements now create a linked list containing two timed actions.

Box 1 CHILL in Examples, continued

```
TimedActionList :=
       ALLOCATE(TimedActionListType,
                [[A1, [16,sep,2000]],NULL]);
TimedActionList :=
       ALLOCATE(TimedActionListType,
                [[A3, [28,aug,2000]],
                TimedActionList]);
```

For homogeneous tuples, as e.g. vectors or matrices, array types can be used. They can have an arbitrary number of dimensions.

```
NEWMODE VectorType = ARRAY(1:3)FLOAT;
NEWMODE SqMatrixType =
       ARRAY(1:3, 1:3) FLOAT;
DCL Vect1 VectorType
          INIT := [1.0, 2.5, 5.0];
DCL Matrix1 SqMatrixType
       INIT := [ [1.0, 2.0, 3.0],
                 [4.0, 5.0, 6.0],
                 [7.0, 8.0, 9.0] ] ;
```

String types are similar to one-dimensional arrays with a special element type, which is either CHARS (= Latin-1), WCHARS (= Unicode) or BOOL.

```
NEWMODE NameType = CHARS(20) VARYING;
DCL MyName NameType INIT := "Winkler";
DCL FirstLetter CHAR INIT := 'W';
```

Sequential Programming: Statements

The section on types already contains several assignment statements. It is therefore not necessary to give further examples.

There are two kinds of selection statements: IF and CASE.

```
IF a>b THEN max := a; ELSE max := b; FI
```

The CASE-statement selects among more alternatives. The CASE-statement of CHILL can also select an alternative using a tuple of n selection values.

```
CASE A, B OF Bool, Bool;
       (false),(false) : Res := false;
       (false),(true)  : Res := false;
       (true), (false) : Res := false;
       (true), (true)  : Res := true;
ESAC
```

There are FOR-loops and WHILE-loops to express repetitive computations.

```
DO WHILE sieve/=empty;
       primes OR:= [MIN(sieve)];
       DO FOR j := MIN(sieve)
                   BY MIN(sieve) TO max;
          sieve -:= [j];
       OD;
OD;
```

Object-Oriented Programming and Genericity

CHILL supports object-oriented programming in a very versatile way in that it combines object-orientation, concurrency and genericity. We show the popular example of the stack data type.

First the specification / interface:

```
SYNMODE IntStackType1 = MODULE SPEC
       GRANT Push, Pop;
       Push: PROC(Elem INT IN)
             EXCEPTIONS(Overflow) END Push;
       Pop:  PROC( ) RETURNS(INT)
             EXCEPTIONS(Underflow) END Pop;
       SYN Length = 10_000;
       DCL StackData ARRAY (1:Length) INT,
             TopOfStack RANGE(0:Length)
                        INIT := 0;
END IntStackType1;
```

The corresponding implementation/body looks like this:

```
SYNMODE IntStackType1 = MODULE BODY
       Push: PROC(Elem INT IN)
             EXCEPTIONS(Overflow)
          IF TopOfStack = Length THEN
             CAUSE Overflow;
          ELSE
             TopOfStack +:= 1;
             StackData(TopOfStack) := Elem;
          FI;
       END Push;
       /* body of Pop */
END IntStackType1;
```

Stack objects are declared in the same manner as variables of other types.

```
DCL Stack1, Stack2  IntStackType1;
Stack1.Push(10);
Stack1.Push(20);
IF Stack1.Pop() > 10 ...
```

Since Stack1 and Stack2 have a finite capacity, it would be better to check whether the operations have been executed normally, i.e. check whether an exception has occurred.

```
Stack1.Push(30)
       ON(Overflow): TempValStack1 := 30;
                     PushStack1 := True;
       END;
```

IntStackType1 is a sequential stack without coordination of concurrent calls, i.e. Stack1 behaves very much like a module. It is easy to define a stack type whose objects behave like regions:

Box 1 CHILL in Examples, continued

```
SYNMODE IntStackType2 = REGION SPEC
       /* as in IntStackType1 */
END IntStackType2;
SYNMODE IntStackType2 = REGION BODY
       /* bodies of Push and Pop */
END IntStackType2;
```

If we use inheritance such a stack type with coordination can be obtained even simpler:

```
SYNMODE IntStackType2 = REGION SPEC
       BASED_ON IntStackType1
END IntStackType2;
```

Both IntStackType1 and IntStackType2 have a fixed element type. If we need stack types for other element types, we have to duplicate the code.

It is simpler first to define a generic stack template StackTemplate1 and then define IntStackType1 and DateStackType1 as generic instantiations of StackTemplate1.

```
GenericStackTemplate1: GENERIC
       MODE ElemType = ANY_ASSIGN;
MODULE SPEC
       GRANT Push, Pop;
       Push: PROC(Elem ElemType IN)
             EXCEPTIONS(Overflow) END Push;
       Pop:  PROC( ) RETURNS(ElemType)
             EXCEPTIONS(Underflow) END Pop;
       SYN Length = 10_000;
       DCL StackData ARRAY (1:Length) INT,
           TopOfStack RANGE(0:Length)
             INIT := 0;
END GenericStackTemplate1;
```

The corresponding implementation/body looks like this:

```
GenericStackTemplate1:
       GENERIC MODE ElemType = ANY_ASSIGN;
MODULE BODY
       /* bodies of Push and Pop */
END GenericStackTemplate1;
```

This template can be used to define object types as instantiations of the template. We do not have to duplicate the code, but only have to provide an actual generic parameter.

```
SYNMODE IntStackType4 =
       NEW GenericStackTemplate1
           SYNMODE ElemType = INT;
END IntStackType4;
SYNMODE DateStackType1 =
       NEW GenericStackTemplate1
           SYNMODE ElemType = DateType;
END DateStackType1;
```

IntStackType4 is essentially equivalent to IntStackType1.

Concurrent Programming

One essential difference between sequential and concurrent programming is the presence of active entities, i.e. entities which have their own thread of control. Such entities are called active entities in contrast to passive entities, as e.g. procedures.

CHILL contains two kinds of active entities: the process and the task object.

Processes typically communicate via buffers, signals or regions. A traditional example is the producer-consumer problem, where a number of processes produce data items and a number of processes consume these data items.

```
ProducerConsumer: MODULE
       DCL PCBuffer BUFFER(100)ItemType;
       ProducerType: PROCESS()
          DCL Item ItemType;
          DO WHILE NotFinished
             /* produce new data item */
             Item := NewValue;
             SEND PCBuffer(Item);
          OD;
       END ProducerType;
       ConsumerType: PROCESS()
          DCL Item ItemType;
          DO WHILE NotFinished
             RECEIVE (PCBuffer IN Item);
             /* consume the data item */
             OD;
       END ConsumerType;
       /* Two producers and one consumer */
       START ProducerType();
       START ProducerType();
       START ConsumerType();
END ProducerConsumer;
```

If there are several kinds of consuming or processing the items produced by the producers, we can define a task type with corresponding methods.

```
ProducerConsumer2: MODULE
   ProducerType: PROCESS()
      DCL Item ItemType;
      DO WHILE NotFinished
         /* produce new data item */
         Item := NewValue;
         CASE KindOfProcessing OF
            (Kind1): Consumer.Consume1(Item);
            (Kind2): Consumer.Consume2(Item);
         ESAC;
   END ProducerType;
   SYNMODE ConsumerType = TASK SPEC
      GRANT Consume1, Consume2;
      Consume1: PROC(Item ItemType IN);
      Consume2: PROC(Item ItemType IN);
   END ConsumerType;
```

CHILL in Examples, continued

```
SYNMODE ConsumerType = TASK BODY
   Consume1: PROC(Item ItemType IN)
      /* consume the data item */
   END Consume1;
   Consume2: PROC(Item ItemType IN)
      /* consume the data item */
   END Consume2;
END ConsumerType;

   /* Two producers and one consumer */
   DCL Consumer ConsumerType;
      /* automatic start */
   START ProducerType();
   START ProducerType();
END ProducerConsumer2;
```

Box 2 CHILL vs. Java

CHILL	**Java**
Data Structures	Data Structures
Scalar: integer, float, characters, boolean, enumerations, pointer, procedure type, process type, event, time range types	Scalar: integer, float, characters, boolean no range types
Composite: string, record, array, set, buffer, signal	Composite: string, array, set, and many others (in the predefined APIs)
Sequential Programming	Sequential Programming
Variable, constant, expression, function call	Variable, constant, expression, function call
Assignment	Assignment
Procedure call	Procedure call
EXIT, RESULT, RETURN, GOTO	BREAK, RETURN
Statement sequence	Statement sequence
Selection statements: IF, CASE (multidimensional)	Selection statements: IF, SWITCH (onedimensional)
Repetition statements: DO, WHILE, FOR	Repetition statements: WHILE-DO, DO-WHILE, FOR
Object-oriented Programming	Object-oriented Programming
Sequential, unsynchronized object	Sequential, unsynchronized object
Sequential, synchronized object	
Concurrent, synchronized object	Concurrent object
Interface	Interface
Friend	
Concurrent Programming	Concurrent Programming
Process	
Start process	
Communication via buffer	
Communication via signal	
Critical region and coordination with events	Synchronized method and synchronized statement
Concurrent, synchronized object	Concurrent object
Program Structure	Program Structure
Block	Block
Procedure / Function / Process	Procedure / Function
Object-Type / Class	Object-Type / Class
Module / Region	Package
Genericity	Genericity
Generic Procedure / Process	
Generic Module / Region	
Generic Object Type / Class	
Generic Interface	
Program Verification	Program Verification
Precondition and postcondition for methods	
Invariant for object type / class	
ASSERT statement	
	Additional Elements

Java – Additional Elements

Element	Package
Applet	java.applet
Reflection	java.lang.reflect
GUI definition	javax.swing, java.awt
SW components	java.beans, org.omg.CORBA
Remote Procedure Call	java.rmi
Internet access	java.net
Data security	java.security
Data base access	java.sql
Data compression	java.util.zip
Painting	java.awt
Music	java.sound.midi

4/2000

Languages for Telecommunication applications

Telektronikk

Volume 96 No. 4 – 2000
ISSN 0085-7130

Editor:
Ola Espvik
Tel: (+47) 63 84 88 83
email: ola.espvik@telenor.com

Status section editor:
Per Hjalmar Lehne
Tel: (+47) 63 84 88 26
email: per-hjalmar.lehne@telenor.com

Editorial assistant:
Gunhild Luke
Tel: (+47) 63 84 86 52
email: gunhild.luke@telenor.com

Editorial office:
Telenor Communication AS
Telenor R&D
PO Box 83
N-2027 Kjeller
Norway
Tel: (+47) 63 84 84 00
Fax: (+47) 63 81 00 76
email: telektronikk@telenor.com

Editorial board:
Ole P. Håkonsen,
Senior Executive Vice President.
Oddvar Hesjedal,
Vice President, R&D.
Bjørn Løken,
Director.

Graphic design:
Design Consult AS, Oslo

Layout and illustrations:
Gunhild Luke, Britt Kjus (Telenor R&D)

Prepress and printing:
Optimal as, Oslo

Circulation:
4,000

Telenor

Contents

Feature:
Languages for Telecommunications Applications

1 Guest Editorial; *Rolv Bræk*

4 The ITU-T Languages in a Nutshell; *Arve Meisingset and Rolv Bræk*

20 SDL-2000 for New Millennium Systems; *Rick Reed*

36 SDL Combined with UML; *Birger Møller-Pedersen*

54 MSC-2000: Interacting with the Future; *Øystein Haugen*

62 A Tutorial Introduction to ASN.1 97; *Colin Willcock*

70 CHILL 2000; *Jürgen F H Winkler*

78 Object Definition Language; *Marc Born and Joachim Fischer*

85 Conformance Testing with TTCN; *Ina Schieferdecker and Jens Grabowski*

96 On Methodology Using the ITU-T Languages and UML; *Rolv Bræk*

107 Descriptive SDL; *Steve Randall*

113 Combined Use of SDL, ASN.1, MSC and TTCN; *Anthony Wiles and Milan Zoric*

120 Implementing from SDL; *Richard Sanders*

130 Validation and Testing; *Dieter Hogrefe, Beat Koch and Helmut Neukirchen*

137 Distributed Platform for Telecommunications Applications; *Anastasius Gavras*

146 Formal Semantics of Specification Languages; *Andreas Prinz*

156 Telelogic SDL and MSC Tool Families; *Philippe Leblanc, Anders Ek and Thomas Hjelm*

164 Cinderella SDL – A Case Tool for Analysis and Design; *Anders Olsen and Finn Kristoffersen*

172 The Evolution of SDL-2000; *Rick Reed*

181 Perspective on Language and Software Standardisation; *Amardeo Sarma*

Special

191 Quality of Service in the ETSI TIPHON Project; *Magnus Krampell*

196 QoS and SLA Structure in a VoIP Service Case; *Irena Grgic, Ola Espvik, Terje Jensen and Magnus Krampell*

220 Some Physical Considerations Concerning Radiation of Electromagnetic Waves; *Knut N Stokke*

229 Telektronikk Index 2000