INTERNATIONAL STANDARD

ISO/IEC 9496

Fourth edition
2003-12-15

CHILL — The ITU-T programming language

CHILL — Le langage de programmation de l'UIT-T

Reference number
ISO/IEC 9496:2003(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

CONTENTS

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the technical committee are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75 % of the member bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights.

ISO/IEC 9496 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*, in collaboration with ITU-T. The identical text is published as ITU-T Rec. Z.200.

This fourth edition cancels and replaces the third edition (ISO/IEC 9496:1998), which has been technically revised.

INTERNATIONAL STANDARD

ITU-T RECOMMENDATION

CHILL – THE ITU-T PROGRAMMING LANGUAGE

This Recommendation | International Standard defines the ITU-T programming language CHILL. When CHILL was first defined in 1980 "CHILL" stood for CCITT High Level Language.

The following subclauses of this clause introduce some of the motivations behind the language design and provide an overview of the language features.

For information concerning the variety of introductory and training material on this subject, the reader is referred to the Manuals, "Introduction to CHILL" and "CHILL user's manual".

An alternative definition of CHILL, in a strict mathematical form (based on the VDM notation), is available in the Manual entitled "Formal definition of CHILL".

1.1 General

CHILL is a strongly typed, block structured language designed primarily for the implementation of large and complex embedded systems.

CHILL was designed to:

- enhance reliability and run time efficiency by means of extensive compile-time checking;
- be sufficiently flexible and powerful to encompass the required range of applications and to exploit a variety of hardware;

- provide facilities that encourage piecewise and modular development of large systems;
- cater for real-time applications by providing built-in concurrency and time supervision primitives;
- permit the generation of highly efficient object code;
- be easy to learn and use.

The expressive power inherent in the language design allows engineers to select the appropriate constructs from a rich set of facilities such that the resulting implementation can match the original specification more precisely.

Because CHILL is careful to distinguish between static and dynamic objects, nearly all the semantic checking can be achieved at compile time. This has obvious run time benefits. Violation of CHILL dynamic rules results in run-time exceptions which can be intercepted by an appropriate exception handler (however, generation of such implicit checks is optional, unless a user defined handler is explicitly specified).

CHILL permits programs to be written in a machine independent manner. The language itself is machine independent; however, particular compilation systems may require the provision of specific implementation defined objects. It should be noted that programs containing such objects will not, in general, be portable.

1.2 Language survey

A CHILL program consists essentially of three parts:

- a description of objects;
- a description of actions which are to be performed upon the objects;
- a description of the program structure.

Objects are described by data statements (declaration and definition statements), actions are described by action statements and the program structure is described by program structuring statements.

The manipulatable objects of CHILL are values and locations where values can be stored. The actions define the operations to be performed upon the objects and the order in which values are stored into and retrieved from locations. The program structure determines the lifetime and visibility of objects.

CHILL provides for extensive static checking of the use of objects in a given context.

In the following subclauses, a summary of the various CHILL concepts is given. Each subclause is an introduction to a clause with the same title, describing the concept in detail.

1.3 Modes and classes

A location has a mode attached to it. The mode of a location defines the set of values which may reside in that location and other properties associated with it (note that not all properties of a location are determinable by its mode alone). Properties of locations are: size, internal structure, read-onliness, referability, etc. Properties of values are: internal representation, ordering, applicable operations, etc.

A value has a class attached to it. The class of a value determines the modes of the locations that may contain the value.

CHILL provides the following categories of modes:

- discrete modes: integer, character, boolean, set (enumerations) modes and ranges thereof;
- real modes: floating point modes and ranges thereof;
- powerset modes: sets of elements of some discrete mode;
- reference modes: bound references, free references and rows used as references to locations;
- composite modes: string, array and structure modes;
- procedure modes: procedures considered as manipulatable data objects;
- instance modes: identifications for processes;
- synchronization modes: event and buffer modes for process synchronization and communication;
- input-output modes: association, access and text modes for input-output operations;
- timing modes: duration and absolute time modes for time supervision;
- moreta modes: module, region and task modes for object orientation with single inheritance.

CHILL provides denotations for a set of standard modes. Program defined modes can be introduced by means of mode definitions. Some language constructs have a so-called dynamic mode attached. A dynamic mode is a mode of which some properties can be determined only dynamically. Dynamic modes are always parameterized modes with run-time parameters. A mode that is not dynamic is called a static mode.

With moreta modes CHILL supports object oriented programming in a very versatile manner. There are three kinds of modes for objects:

- module modes: the values of these modes behave very much like modules and resemble therefore mostly the objects in classical object oriented programming (e.g. Smalltalk, C++, Eiffel, Java);
- region modes: the values of these modes behave very much like regions. Such objects are usually not found in classical object oriented programming;
- task modes: the values of these modes have essentially the same structure as regions but have their own thread of control, and communication between them and other objects is done asynchronously.

Classes have no denotation in CHILL. They are introduced in the metalanguage only to describe static and dynamic context conditions.

1.4 Locations and their accesses

Locations are places where values can be stored or from which values can be obtained. In order to store or obtain a value, a location has to be accessed.

Declaration statements define names to be used for accessing a location. There are:

1) location declarations;

2) loc-identity declarations.

The first one creates locations and establishes access names to the newly created locations. The latter one establishes new access names for locations created elsewhere.

Apart from location declarations, new locations can be created by means of a *GETSTACK* or *ALLOCATE* built-in routine call yielding reference values (see below) to the newly created location.

A location may be **referable**. This means that a corresponding reference value exists for the location. This reference value is obtained as the result of the referencing operation, applied to the **referable** location. By dereferencing a reference value, the referred location is obtained. CHILL requires certain locations to be **referable** and others to be not **referable**, but for other locations it is left to the implementation to decide whether or not they are **referable**. Referability must be a statically determinable property of locations.

A location may have a **read-only** mode, which means that it can only be accessed to obtain a value and not to store a new value into it (except when initializing).

A location may be composite, which means that it has sub-locations which can be accessed separately. A sub-location is not necessarily **referable**. A location containing at least one **read-only** sub-location is said to have the **read-only property**. The accessing methods delivering sub-locations (or sub-values) are indexing and slicing for strings and for arrays, and selection for structures.

A location has a mode attached. If this mode is dynamic, the location is called a dynamic mode location.

The following properties of a location, although statically determinable, are not part of the mode:

referability: whether or not a reference value exists for the location;

storage class: whether or not it is statically allocated;

regionality: whether or not the location is declared within a region.

1.5 Values and their operations

Values are basic objects on which specific operations are defined. A value is either a (CHILL) defined value or an **undefined** value (in the CHILL sense). The usage of an undefined value in specified contexts results in an undefined situation (in the CHILL sense) and the program is considered to be incorrect.

CHILL allows locations to be used in contexts where values are required. In this case, the location is accessed to obtain the value contained in it.

A value has a class attached. **Strong** values are values that besides their class also have a mode attached. In that case the value is always one of the values defined by the mode. The class is used for compatibility checking and the mode for describing properties of the value. Some contexts require those properties to be known and a **strong** value will then be required.

A value may be **literal**, in which case it denotes an implementation independent discrete value, known at compile time. A value may be **constant**, in which case it always delivers the same value, i.e. it need only be evaluated once. When the context requires a **literal** or **constant** value, the value is assumed to be evaluated before run-time and therefore cannot generate a run-time exception. A value may be **intra-regional**, in which case it can refer somehow to locations declared within a region. A value may be composite, i.e. contain sub-values.

Synonym definition statements establish new names to denote **constant** values.

1.6 Actions

Actions constitute the algorithmic part of a CHILL program.

The assignment action stores a (computed) value into one or more locations. The procedure call invokes a procedure, a built-in routine call invokes a built-in routine (a built-in routine is a procedure whose definition need not be written in CHILL and whose parameter and result mechanism may be more general). To return from and/or establish the result of a procedure call, the return and result actions are used.

To control the sequential action flow, CHILL provides the following flow of control actions:

– if action: for a two-way branch;

– case action: for a multiple branch. The selection of the branch may be based upon several values, similarly to a decision table;

– do action: for iteration or bracketing;

– exit action: for leaving a bracketed action or a module in a structured manner;

– cause action: to cause a specific exception;

– goto action: for unconditional transfer to a labelled program point.

Action and data statements can be grouped together to form a module or begin-end block, which form a (compound) action.

To control the concurrent action flow, CHILL provides the start, stop, delay, continue, send, delay case, and receive case actions, and receive and start expressions.

1.7 Input and output

The input and output facilities of CHILL provide the means to communicate with a variety of devices in the outside world.

The input-output reference model knows three states. In the free state there is no interaction with the outside world.

Through an *ASSOCIATE* operation, the file handling state is entered. In the file handling state there are locations of association mode, which denote outside world objects. It is possible via built-in routines to read and modify the language defined attributes of associations, i.e. **existing**, **readable**, **writeable**, **indexable**, **sequencible** and **variable**. File creation and deletion are also done in the file handling state.

Through the *CONNECT* operation, a location of access mode is connected to a location of an association mode, and the data transfer state is entered. The *CONNECT* operation allows positioning of a **base** index in a file. In the data transfer state various attributes of locations of access mode can be inspected and the data transfer operations *READRECORD* and *WRITERECORD* can be applied.

Through the text transfer operations, CHILL values can be represented in a human-readable form which can be transferred to or from a file or a CHILL location.

1.8 Exception handling

The dynamic semantic conditions of CHILL are those (non context-free) conditions that, in general, cannot be statically determined. (It is left to the implementation to decide whether or not to generate code to test the dynamic conditions at run time, unless an appropriate handler is explicitly specified.) The violation of a dynamic semantic rule causes a run-time exception; however, if an implementation can determine statically that a dynamic condition will be violated, it may reject the program.

Exceptions can also be caused by the execution of a cause action or, conditionally, by the execution of an assert action. When, at a given program point, an exception occurs, control is transferred to the associated handler for that exception, if one is specified. Whether or not a handler is specified for an exception at a given point can be statically determined. If no explicit handler is specified, control may be transferred to an implementation defined exception handler.

Exceptions have a name, which is either a CHILL defined exception name, an implementation defined exception name, or a program defined exception name. Note that when a handler is specified for an exception name, the associated dynamic condition must be checked.

1.9 Time supervision

Time supervision facilities of CHILL provide the means to react to the elapsing of time in the external world. A process becomes **timeoutable** when it reaches a well-defined point in the execution of certain actions. At this point it may be interrupted. When this happens, control is transferred to an appropriate handler.

Programs may detect the elapsing of a period of time or may synchronize to an absolute point of time or at precise intervals without cumulated drifts. Built-in routines for time are provided to convert absolute time values and duration values into integer values, to suspend a process until a time supervision expires.

1.10 Program structure

The program structuring statements are the begin-end block, module, procedure, process, region and moreta mode. The program structuring statements provide the means of controlling the lifetime of locations and the visibility of names.

The lifetime of a location is the time during which a location exists within the program. Locations can be explicitly declared (in a location declaration) or generated (*GETSTACK* or *ALLOCATE* built-in routine call), or they can be implicitly declared or generated as the result of the use of language constructs.

A name is said to be **visible** at a certain point in the program if it may be used at that point. The scope of a name encompasses all the points where it is **visible**, i.e. where the denoted object is identified by that name.

Begin-end blocks determine both visibility of names and lifetime of locations.

Modules are provided to restrict the visibility of names to protect against unauthorized usage. By means of visibility statements, it is possible to exercise control over the visibility of names in various program parts.

A procedure is a (possibly parameterized) sub-program that may be invoked (called) at different places within a program. It may return a value (value procedure) or a location (location procedure), or deliver no result. In the latter case the procedure can only be called in a procedure call action.

Processes, task locations, regions and region locations provide the means by which a structure of concurrent executions can be achieved.

Generic templates provide the means by which generic modules, regions, procedures, processes and moreta modes can be constructed. These templates can be parameterized by SYN constants, modes and procedures. Generic instantiation statements are used to obtain (nongeneric) modules, regions, procedures, processes and moreta modes which are called generic instances. A generic instance is obtained from a generic template T by replacing in T the formal generic parameters with the corresponding actual generic parameters.

A complete CHILL program is a list of program units that is considered to be surrounded by an (imaginary) process definition. This outermost process is started by the system under whose control the program is executed. A program unit can be a module, a region, a moreta synmode definition statement, a moreta newmode definition statement or a generic template.

Constructs are provided to facilitate various ways of piecewise development of programs. A spec module and spec region are used to define the static properties of a program piece, a context is used to define the static properties of seized names. In addition it is possible to specify that the text of a program piece is to be found somewhere else through the remote facility.

1.11 Concurrent execution

CHILL allows for the concurrent execution of program units. A thread (process or task) is the unit of concurrent execution. The evaluation of a start expression causes the creation of a new process of the indicated process definition. The process is then considered to be executed concurrently with the starting thread. CHILL allows for one or more processes with the same or different definition to be active at one time. The stop action, executed by a process or a task, causes its termination.

A thread is always in one of two states; it can be active or delayed. The transition from active to delayed is called the delaying of the thread; the transition from delayed to active is called the re-activation of the thread. The execution of delaying actions on events, or receiving actions on buffers or signals, or sending actions on buffers, or call action to a component procedure of a region location, or call action to a component procedure of a task location in case there is not enough storage to perform can cause the executing thread to become delayed. The execution of a continue action on events, or sending actions on buffers or signals, or receiving actions on buffers, or release of a region location, or at the beginning of the execution of an externally called component procedure of a task location can cause a delayed thread to become active again.

Buffers and events are locations with restricted use. The operations send, receive and receive case are defined on buffers; the operations delay, delay case and continue are defined on events. Buffers are a means of synchronizing and transmitting information between processes. Events are used only for synchronization. Signals are defined in signal definition statements. They denote functions for composing and decomposing lists of values transmitted between processes. Send actions and receive case actions provide for communication of a list of values and for synchronization.

A region or region location is a special kind of module. Its use is to provide for mutually exclusive access to data structures that are shared by several threads.

1.12 General semantic properties

The semantic (non context-free) conditions of CHILL are the mode and class compatibility conditions (mode checking) and the visibility conditions (scope checking). The mode rules determine how names may be used; the scope rules determine where names may be used.

The mode rules are formulated in terms of compatibility requirements between modes, between classes and between modes and classes. The compatibility requirements between modes and classes and between classes themselves are defined in terms of equivalence relations between modes. If dynamic modes are involved, mode checking is partly dynamic.

The scope rules determine the visibility of names through the program structure and explicit visibility statements. The explicit visibility statements influence the scope of the mentioned names. Names introduced in a program have a place where they are defined or declared. This place is called the defining occurrence of the name. The places where the name is used are called applied occurrences of the name. The name binding rules associate a unique defining occurrence with each applied occurrence of the name.

1.13 Implementation options

CHILL allows for implementation defined integer modes, implementation defined built-in routines, implementation defined **process** names, implementation defined exception handlers and implementation defined exception names.

An implementation defined integer mode must be denoted by an implementation defined **mode** name. This name is considered to be defined in a newmode definition statement that is not specified in CHILL. Extending the existing CHILL-defined arithmetic operations to the implementation defined integer modes is allowed within the framework of the CHILL syntactic and semantic rules. Examples of implementation defined integer modes are long integers, and short integers.

A built-in routine is a procedure whose definition need not be written in CHILL and that may have a more general parameter passing and result transmission scheme than CHILL procedures.

A built-in **process** name is a process name whose definition need not be written in CHILL and that may have a more general parameter passing scheme than CHILL processes. A CHILL process may cooperate with built-in processes or start such processes.

An implementation defined exception handler is a handler appended to a process definition. If this handler receives control after the occurrence of an exception, the implementation decides which actions are to be taken. An implementation defined exception is caused if an implementation defined dynamic condition is violated.

2 Preliminaries

2.1 The metalanguage

The CHILL description consists of two parts:

- the description of the context-free syntax;
- the description of the semantic conditions.

2.1.1 The context-free syntax description

The context-free syntax is described using an extension of the Backus-Naur Form. Syntactic categories are indicated by one or more English words, written in slanted characters, enclosed between angular brackets (< and >). This indicator is called a non-terminal symbol. For each non-terminal symbol, a production rule is given in an appropriate syntax section. A production rule for a non-terminal symbol consists of the non-terminal symbol at the left-hand side of the symbol *::=*, and one or more constructs, consisting of non-terminal and/or terminal symbols at the right-hand side. These constructs are separated by a vertical bar (|) to denote alternative productions for the non-terminal symbol.

Sometimes the non-terminal symbol includes an underlined part. This underlined part does not form part of the context-free description but defines a semantic category (see 2.1.2).

Syntactic elements may be grouped together by using curly brackets ({ and }). Repetition of curly bracketed groups is indicated by an asterisk (*) or plus ($^+$). An asterisk indicates that the group is optional and can be further repeated any number of times; a plus indicates that the group must be present and can be further repeated any number of times. For example, { *A* }* stands for any sequence of *A*'s, including zero, while { *A* }$^+$ stands for any sequence of at least one *A*. If syntactic elements are grouped using square brackets ([and]), then the group is optional. A curly or square bracketed group may contain one or more vertical bars, indicating alternative syntactic elements.

A distinction is made between strict syntax, for which the semantic conditions are given directly, and derived syntax. The derived syntax is considered to be an extension of the strict syntax and the semantics for the derived syntax is indirectly explained in terms of the associated strict syntax.

It is to be noted that the context-free syntax description is chosen to suit the semantic description in this Recommendation | International Standard and is not made to suit any particular parsing algorithm (e.g. there are some context-free ambiguities introduced in the interest of clarity). The ambiguities are resolved using the semantic category of the syntactic elements.

2.1.2 The semantic description

Each syntactic category (non-terminal symbol) is described in sub-sections **semantics**, **static properties**, **dynamic properties**, **static conditions** and **dynamic conditions**.

The section **semantics** describes the concepts denoted by the syntactic categories (i.e. their meaning and behaviour).

The section **static properties** defines statically determinable semantic properties of the syntactic category. These properties are used in the formulation of static and/or dynamic conditions in the sections where the syntactic category is used.

The section **dynamic properties** defines the properties of the syntactic category, which are known only dynamically.

The section **static conditions** describes the context-dependent, statically checkable conditions which must be fulfilled when the syntactic category is used. Some static conditions are expressed in the syntax by means of an underlined part in the non-terminal symbol (see 2.1.1). This use requires the non-terminal to be of a specific semantic category. For example, *boolean expression* is identical to <*expression*> in the context-free sense, but semantically it requires the *expression* to be of a boolean class.

The section **dynamic conditions** describes the context-dependent conditions that must be fulfilled during execution. In some cases, conditions are static if no dynamic modes are involved. In those cases, the condition is mentioned under **static conditions** and referred to under **dynamic conditions**. In other cases, dynamic conditions can be checked statically; an implementation may treat this as a violation of a static condition.