Towards a new major release of the TTCN-3 Core Language

Source: Testing Task Force (TTF) T023 on TTCN-3 Maintenance

# Rationale

The last major release of TTCN-3 Core Language was published in 2009 with version 4.1.1. Since then the core language was constantly maintained. The whole TTCN-3 language has been further developed by introducing smaller extensions in the core language and by developing extension packages expressing the wishes of users and developers.

However, it is known that language maintenance through minor corrections and extensions based on issue tracking leads to a growth of the specification and is accompanied by a growth in complexity and an erosion of quality. After a while it is required to clean up the standard and to modernize and refactor the specification in order to keep the high-quality of the TTCN-3 language.

Currently, language maintenance has become complex, since even small corrections and extensions require changes in many different places in the documents. Therefore, the members of TTF T023 believe that 13 years after the last major revision a new major revision should be prepared that cleans up, modernizes and refactors the TTCN-3 language specification. Cleaning up and refactoring should address:

* technical aspects like for example the growing number of keywords and reserved words or syntax and BNF simplifications,
* structural aspects like, for example, deletion of deprecated constructs and moving descriptive language elements into annexes,
* language feature aspects which include reviews of language features which are rarely used, language features which should be moved into extension packages, language elements which should be moved from extension packages into the core language and new language features which ease the use of TTCN-3, and
* a review of the different TTCN-3 parts and extension packages.

TTF T023 understands that non-backwards compatible changes should be handled with care and should be avoided whenever possible. However, a major release offers a chance to modernize the language and get rid of legacy language features.

# Purpose of this document

This document serves for discussion purposes and collects ideas for a new major release of the TTCN-3 language. Some of the already submitted CRs are considered to be resolved in the scope of working on a new major release of TTCN-3. These CRs are mentioned in the following sections.

# Technical aspects

Technical aspects cover technical changes in the way of writing the standard. For example, the handling of reserved words and keywords has grown over the last decades. A review of the handling of these special words may help to handle them differently and thereby reduce their number.

The following already submitted CRs address technical aspects of the work on a new major release of TTCN-3:

* 8090: Deprecate “lengthof” in favor of “length”
* 8100: Inline terminal productions
* 8112: Combine boolean and bitwise operators
* 8152: Harmonize string literals
* 8192: Redefine keywords and reserved words

# Structural aspects

Structural aspects cover aspects regarding the structure of the standard like the handling of deprecated language features or moving descriptive language elements into annexes.

CRs addressing structural aspects:

* 8095: Provide a TTCN-3 specification for TCI and TRI
* 8106: Provide TTCN-3 defintions for predefined types
* 8195: Make specification updates easier to find

# Language feature aspects

Language feature aspects covers all wishes regarding new language features and the semantical and/or syntactical change of existing languages constructs. Wishes may include moving language features into extension packages and vice versa.

The following already submitted CRs cover language feature aspects:

* 7981: Support for REST APIs (HTTP)
* 8094: Provide a canonical style for source code layout
* 8111: Allow UTF-8 for charstrings
* 8113: Type traits and user defined methods
* 8153: Extend usage of break and continue statements
* 8156: Introduce user defined methods
* 8188: Support for function literals
* 8189: Implicit Apply
* 8190: Expression Bodies
* 8191: Strict Rules
* 8102: Simplify Import
* 8194: Optional Names for Formal Parameters
* 8196: Redefining Macros as Predefined Constants
* 8197: Automatic Alternative Selection for Unions

# Critical review of TTCN-3 parts and TTCN-3 extension packages

A critical review should study the usage of the different TTCN-3 parts and extension packages. On the one hand, for several parts and extension packages no CRs have been submitted for years. The need and effort for maintenance should be discussed. On the other hand, several extension packages include interesting features which are rarely used because the extension package is not supported by tool vendors. Some of these features may be moved to the core language.

# Technical Aspects

# 0008090: Deprecate `lengthof` in favor of `length`

Nokia – Matthias Simon

I often accidentally use keyword `length` instead of predefined function `lengthof`. The result is an irritating syntax error ala `unexpected length, expecting expression`.

This mistake is easy to make. Hence I propose to rename predefined function `lengthof` to `length`. And begin deprecation process for the `lengthof` keyword.

This would become valid TTCN-3 code:

for (i := 0; i < length('1100101'b); i := i + 1) { /\* ... \*/ }

COMMENT DEVOTEAM:

Keywords should not be used as function names, example  
type octetstring oct\_1 length(1); // looks like a function but is a keyword here  
for (i := 0; i < length('1100101'b); i := i + 1) { /\* ... \*/ } // use of renamed function length()  
… would imho be a bit confusing for users

The keyword **length** is used in context of type- and template-definitions. It is used to define a fixed length or range-length.

The predefined function lengthof() is used to evaluate the length of a specific string- or of-type values.

I think it ‘s no improvement to use keyword **length** instead of **lengthof** for evaluation of value-length.

# 8100: Inline terminal productions

Nokia – Matthias Simon

Inlining terminal production would make the grammar easier to comprehend. For example:  
  
- 1.TTCN3Module ::= TTCN3ModuleKeyword ModuleId "{" ...  
- 2.TTCN3ModuleKeyword ::= "module"  
+ 1.TTCN3Module ::= "module" ModuleId "{" ...

COMMENT DEVOTEAM:

OK, does not touch the syntax and semantic of TTCN-3.

# 8112: Combine boolean and bitwise operators

Nokia – Matthias Simon

Currently TTCN-3 distinguishes between boolean operators (and, or, not, ...) and bitwise operators (xor4b, and4b, or4b, not4b, ...) for bit string types.  
 Combining those operators would reduce diversity towards a simpler TTCN-3 grammar.  
  
Examples:  
    0b'1100' xor 0b'1010' // 0b'0110'  
    false xor false // false  
    true xor false // true

The original bitwise operators would be deprecated.

COMMENT Tomas Urban:

Input for discussion:  
Impossible to achieve as proposed as the change would break operator precedence.  
Simplification in java/C++/C# style is possible for logical operators: &&, ||, ! (^^ could be introduced for logical xor) and most of bitwise operators: |, ^, ~  
A similar change is problematic for bitwise operators as & is used for concatenation with a similar scope. Could be resolved with a different symbol, such as #, $ or combination of symbols e.g. :&

COMMENT DEVOTEAM:

Many languages distinguish between bitwise and logical operators (C, C++, Java, Python,…). In my opinion this makes sense. Otherwise new questions must be answered: What are allowed Boolean expressions for if-, while-, for- statements, guardChar or assignments to variables of type boolean?  
E.g.: if (0b'1100' xor 0b'0011') … // what about this? Should it be allowed?

Regarding Tomas input,   
I think the syntax should not be changed as this decision has been taken long ago, nor would it be an improvement of TTCN-3. Especially the string operator & complicates such a change.

IMHO this is not really a simplification of the language.

# 8152: Harmonize string literals

Nokia – Matthias Simon

Ideas to discuss:  
\* Allow pattern string quoting in character strings  
\* Allow line breaks in character strings and patterns

COMMENT DEVOTEAM:  
Let’s discuss about the details.

# 8192: Simplify import statement (Redefine keywords and reserved words)

Nokia – Matthias Simon

|  |
| --- |
| The current specification supports a very fine-grained -- and often unnecessary -- control over importing symbols. Simplifying the specification could be beneficial: ImportDef ::= "import" ( "all" | IdentifierList ) "from" Identifier ["->" Identifier] ";" Examples:    import a,b,c from M -> OtherName;    import all from M; |

COMMENT DEVOTEAM:

Ok. But old Syntax should still be allowed!

But let’s rise up another question regarding renaming of an imported module (EXAMPLE 4 in chapter 8.2.3.1 General format of import):

**import from** VeryLongModuleNameB -> ShortBType {

**type** MyTypeA, // Where MyTypeA is of type character string

}

**import from** VeryLongModuleNameB -> ShortBFunc {

**function** func1, // Where func1 is a defined function in module VeryLongModuleNameB

}

**import from** VeryLongModuleNameB -> ShortBTestcase {

**testcase** tc1, // Where tc1 is a defined testcase in module VeryLongModuleNameB

}

Here we now have three different aliases for the SAME original module!

Could types, functions, testcases only be identified by the original module name and the alias belonging to the import-statetment?

Wouldn’t it be clearer to define a separate module-alias, e.g.  
 VeryLongModuleNameB -> ShortBType, ShortBFunc, ShortBTestcase;   
//alias should be defined only once per imported and current module  
//all these identifiers should be unique and cannot be used in another alias-statement

# Structural aspects

# 8095: Provide a TTCN-3 specification for TCI and TRI

Nokia – Matthias Simon

Input for discussion(Jens Grabowski): <https://github.com/ttcn3/specs/blob/main/control_and_runtime_interface.ttcn3>

COMMENT DEVOTEAM:

What are the advantages of a new interface-definition?

# 8106: Provide TTCN-3 defintions for predefined types

Nokia – Matthias Simon

Predefined functions and some operations like log, match, setverdict, ... can be specified as external function using valid TTCN-3 syntax (see <https://github.com/ttcn3/specs/blob/main/predefined_functions.ttcn3#L692-L713> [[^](https://github.com/ttcn3/specs/blob/main/predefined_functions.ttcn3#L692-L713)]).  
Those functions could form some kind of standard library. This would make the core language specification slimmer and the tools easier to implement and maintain.  
  
If the object oriented extension is available, operations of ports, testcases, components and timers could possibly be specified as method of external abtract classes.

COMMENT DEVOTEAM:

I think this standard-library is still provided by tool-vendors. The Tli-interface provides interface to log customer specific log-, verdict and match- operations. Predefined functions have a fix semantic and should be provided by the tool-vendors (implementation is tool dependent), whereas external functions are implemented by customers.

# 8195: Make specification updates easier to find

Nokia – Matthias Simon

The TTCN-3 specification is a constantly evolving standard, making it sometimes difficult for users to find and understand the changes made.  
  
To solve this problem, I propose making the diffs of changes easier to find and providing release notes that summarize the changes, new features and any other important information.  
  
This will make it easier for users to familiarize themselves with the updated standard, and for vendors to implement the changes.

COMMENT DEVOTEAM:

Providing release notes that summarize the changes are nice to have, but do not show the details required to implement the changes.

IMHO the list of Mantis CRs (incl. the approved CR solution document(s)) and providing the new standard versions in Word format (\*.docx) instead of PDF helps to get a quick, meaningful diff of all changes.

# Language feature aspects

# 7981: Support for REST APIs (HTTP)

Martti Käärik

ETSI CTI and MTS have been jointly working on establishing unified methodology for specification and testing of REST APIs. The activities have been carried out by STF576 (<https://portal.etsi.org/STF/STFs/STF-HomePages/STF576> [[^](https://portal.etsi.org/STF/STFs/STF-HomePages/STF576)]) and resulted in an initial version of a guide document: EG 203 647 (<https://portal.etsi.org/webapp/WorkProgram/Report_WorkItem.asp?WKI_ID=56708> [[^](https://portal.etsi.org/webapp/WorkProgram/Report_WorkItem.asp?WKI_ID=56708)]).  
  
STF576 has integrated several existing methodologies and languages developed within MTS in the guide, to the extent that those are suitable for REST APIs. This includes TTCN-3. However, to support standardized specification and testing process, TTCN-3 should provide means for describing REST API interface specific artifacts in test cases in tool agnostic manner.  
  
Although there are several protocols that may be used for implementing REST APIs, the initial effort has implicitly focused on HTTP.

Answer from Martti:  
"Yes, we are basically talking about a standardized mapping from TTCN-3 types/templates to HTTP requests/responses (as you describe). Including a way to specify message bodies with some other (standardized) encoding (such as JSON).  
With HTTP (and JSON) mapping, pretty much all of the OpenAPI specification would be supported and no additional work would be required, in my opinion."  
  
So the task at hand is to create a new extension to the core language, similar to ES 201 873-11 (Using JSON with TTCN-3) and ES 201 873-9 (Using XML schema with TTCN-3) for HTTP requests and responses.

COMMENT DEVOTEAM:

We agree to Martti:   
REST HTTP mapping should be defined in a new Standard Extension.

# 8094: Provide a canonical style for source code layout

Nokia – Matthias Simon

A clear recommendation how TTCN-3 source code should be formatted would be beneficial:  
  
\* Tool-vendors had a solid ground to implement automatic formatter tools.  
  
\* Less time would be spent on "bike-shedding" discussions (e.g. tabs vs. spaces).  
  
\* A canonical style improves readability of TTCN-3 source code (e.g. of conformance tests, code examples, ...).

COMMENT DEVOTEAM:   
OK

# 8111: Allow UTF-8 for charstrings

Nokia – Matthias Simon

In recent years UTF-8 has become the de-facto standard for encoding strings.  
TTCN-3 files are also encoded using UTF-8.  
Maybe we should allow UTF-8 encoding for charstrings as well?

Jens Grabowski

TTF discussion: Encoding of charstring should not matter. The encoding should be handled by on encoder level and not on language level. The duality of character encoding should be removed, if not possible deprecated.

Axel Rennoch

Currently "charstring" is based on Recommendation ITU-T T.50, a document referenced multiple times in TTCN-3. Other character standards like e.g. ISO/IEC 10646 are possible using "universal charstring".  
  
TTCN-3 had been created in the telecommunition domain and is also published by ITU-T. Any changes for the current definitions may effect the ITU-T position.

Tomas Urban

TCI constitutes a serious problem. Unfortunately there's no way to have a Java class that implements both CharstringValue and UniversalCharstringValue interfaces. The interfaces contain the same method getChar with different return types. All other language mappings work fine though.  
  
There are two possible solutions, but both have drawbacks:  
1. Introduce a new interface that could be implemented along the existing ones where the getChar method is replaced with getAt (and setChar replaced with setAt for consistency reasons):  
  
package org.etsi.ttcn.tci;  
public interface CStringValue {  
 String getString ();  
 void setString (String value);  
 int **getAt** (int position);  
 void **setAt** (int position, int value);  
 int getLength ();  
 void setLength (int len);  
}  
  
Newly written code can use the new interface and legacy code would still work. The old interfaces would be changed to deprecated, but continue to work in order to provide backwards compatible solution for legacy code. However, it means that the TE would still have to track whether the underlying type is charstring or universal charstring in order to provide a correct legacy interface to Java-based TCI implementations.  
  
2. We could modify the getChar in the Java CharstringValue interface to return an integer value. It is a backwards incompatible change, so this might be a red flag for us. However, the fix in legacy software would be an easy one: explicitly casting the return value to char. All places requiring the fix would be detected by the compiler. A similar change would be made in the setChar method, but this one is backwards compatible. The biggest advantage comparing to the first approach is that charstring and universal charstring types would become true synonyms as the TE wouldn't have to consider the underlying type when passing the data to the TCI.

COMMENT DEVOTEAM:

Where is getAt() or setAt() used by the TCI-interface?

C-Interface provides defined function-definitions:   
  
void **tciGetUCStringCharValue**

(Value inst,

unsigned long int **position**,

TciUCValue result)

void **tciSetUCStringCharValue**

(Value inst,

unsigned long int **position**,

TciUCValue value)

void **tciSetCStringCharValue**

(TciValue inst, long int **position**, char value)

char **tciGetCStringCharValue**   
(TciValue inst, long int **position**)

So, doing the same for Java (proposal 1) would be sufficient.

# 8113: Type traits and user defined methods

Nokia – Matthias Simon

Type traits allow to compose behavior in a lightweight, but powerful way.  
  
**## Methods**  
This extensions allows to specify methods for any user defined types. The  
receiver type is specified using the "for" keyword. Inside the behavior the  
receiver value is accessible via "this" symbol:  
  
module Example {  
 type integer Timestamp  
 function year() for Timestamp return string {  
 return int2str(1970+this/SECONDS\_PER\_YEAR);  
 }  
   
 control {  
 const Timestamp t := 1660681400;  
 log(t.year()) // logs "2022"  
 }  
}  
  
  
**## Traits**  
A trait is a set of methods and can be defined using the "trait" keyword:  
  
trait Stringer {  
 function String() charstring;  
}  
  
A variable of a trait type can hold any value that implements the trait:  
  
module Example {  
  
 type record Point2D { integer x, integer y }  
 function string() for Point3D return charstring {  
 return sprintf("(%d|%d)", this.x, this.y)  
 }  
   
 type record Point3D { integer x, integer y, integer z }  
 function string() for Point2D return charstring {  
 return sprintf("(%d|%d|%d)", this.x, this.y, this.z)  
 }  
   
 trait Stringer {  
 function string() charstring;  
 }  
   
 function logPoints(Stringer s) {  
 log(s.string())  
 }  
   
 control {  
 var Point2D p1 := {1,2};  
 var Point3D p2 := {1,2,3}  
   
 logPoints(p1); // okay because Point2D implements Stringer trait  
 logPoints(p2); // okay because Point3D also implements Stringer trait  
 }  
}  
  
  
**## Embedding**  
When the field name is omitted, the field is called an embedded field:  
  
type integer Timestamp;  
external function year() for Timestamp return charstring;  
  
type record Date {  
 Timestamp, // embedded field  
 charstring zone // regular field  
}  
  
  
An embedded field is accessible by its type name:  
  
var Data d := { Timestamp := 1660681400, zone := "GMT+2" };  
d.Timestamp := d.Timestamp + 3600;  
  
  
Embedded fields must be unique:  
  
type record Date {  
 Timestamp,  
 Timestamp // not allowed.  
}  
  
  
Methods of an embedded field are promoted and become methods of the embedding type:  
  
var Data d := { Timestamp := 1660681400, zone := "GMT+2" };  
log(d.year()); // year is a promoted method implemented by the Timestamp type  
  
  
Conflicting promoted methods have to be resolved explicitly:  
  
type integer Duration;  
external function year() for Duration return charstring;  
  
type record Event {  
 Timestamp,  
 Duration  
}  
  
// Timestamp and Duration both provide a "year"-method. Event type need  
// to resolve this conflict explicitly:  
function year() for Event return charstring {  
 return sprintf("start=%s, duration=%s", this.Timestamp, this.Duration)  
}  
  
  
**## Notes and Open Questions**  
\* Should we call it "trait" or rather "interface" like in Java, C# and Go?  
\* Should we support default implementations for traits (requires "implements" keyword)?

Nokia – Matthias Simon

What syntax for defining methods do your prefer?  
  
# Free floating (like in Go, Perl, ...)  
  
Example:  
function F() for T runs on C return boolean {  
 return this > 5;  
}  
  
Variations:  
    function F() with T ...  
    function F() extends T ...  
    function F() on T ...  
    function F() to T ...  
    function F() at T ...  
    function F() in T ...  
    function F() -> T ...  
    function F() => T ...  
    on T function F() ...  
    function for T F() ...  
  
   
# Qualified names (like in C++)  
  
Example:  
function T::F() runs on C return boolean {  
 return this > 5;  
}  
  
Variations:  
    function T.F() ...  
    function T:F() ...  
  
  
# Nested declarations (like in Java, Python, ...)  
  
Example:  
type record T {  
 integer x,  
 function F() runs on C return boolean { return true },  
 integer y  
}  
  
type integer I8 (-127..128) {  
 function F() runs on C return boolean { return true };  
}  
  
  
# Dedicated method block  
  
Example:  
type record T {  
 integer x,  
 integer y  
} with {  
 function F() runs on C return boolean { return true };  
}  
  
type integer I8 (-127..128) with {  
 function F() runs T return boolean { return true };  
}  
  
Variations:  
    type integer I8 extends { ... }  
    type integer I8 implements { ... }  
    type integer I8 group { ... }  
    type integer I8 bind { ... }  
    type integer I8 connect { ... }

Tomas Urban:

|  |
| --- |
| This is a topic for a discussion.   Free floating syntax resembles what we already have in TTCN-3. It can extend built-in types as well. However, there are two major problems that are not easy to resolve (valid for qualified names too): 1. Import of traits, especially when they are defined in a different module than the type (could be resolved by dedicated import rules) 2. Violation of scoping principles of TTCN-3: So far all lower scopes are declared inside a definition. Using free floating could lead to traits with the same name but different functionality being present in more than one module and we would need a whole set of rule for handling that.  For these reasons, I would prefer an encapsulation approach. Out of the proposed options I like nested declarations more, because the syntax doesn't use excess symbols. |
|  | |
| Jens Grabowski:   |  | | --- | | TTF discussion: Open questions are related to syntax and semantics. Unclear where to put this CR. May become part of core language or may become part of the OO extension. Resolution of issues should be discussed in the scope of language renovation. | |  | | | Matthias Simon:  Split of embedded fields part: <http://oldforge.etsi.org/mantis/view.php?id=8154> [[^](http://oldforge.etsi.org/mantis/view.php?id=8154)]  Split of methods part: <http://oldforge.etsi.org/mantis/view.php?id=8156> [[^](http://oldforge.etsi.org/mantis/view.php?id=8156)]  **import from** Math **all**;  **type** **record** Pixel {  Math.Point, // embedded field with implicit name Point  **charstring** color,  **float** z  }  **var** Pixel p := { Point := {4,8}, color := "red" };  **log**(p.**Point**.x); // explicit access to field x  **log**(p.x); // access to promoted field x  **log**(p.z); // is NOT allowed: z is not a promoted field.   // Explicit access is required.  // **Comment Devoteam:**  // log(p.z) is not promoted but is allowed here  // as float z is explicitly defined in the record.  // If p.z is not assigned it would lead to a runtime error  **module** Math {  **type** **record** Point { **integer** x, **integer** y, **integer** z }  } |  | |  |

Additional comment to following text in 6.2.1.4 of CR1541: A field of an *embedded field* is called *promoted* if its name has no conflicts with other field names. *Promoted* fields act like ordinary fields of a record, except that they shall not be used as field names for [a](https://go.dev/ref/spec#Composite_literals)ssignment notations.  
WHAT DOES IT MEAN? Following Value-notation is allowed?:  
  
var Pixel P p := { {4,8},color := “red” }; // might be allowed?  
but  
var Pixel P p := { 4,8,“red” }; // assign as promoted field is not allowed?

COMMENT DEVOTEAM:

WHAT DOES IT MEAN? Following Value-notation is allowed?:  
var Pixel P p := { {4,8}, “red” }; // might be allowed?

What’s the sense of embedded fields (CR 8154)? Why log(p.z), as z is also a field of type Pixel (EXAMPLE in chapter 6.2.1.4 of CR8154.doc, it’s a field of type Pixel)? This should lead to a runtime error as the field is uninitialized.

type record EmbedType {  
 EmbedType3,  
 EmbedType2,  
 integer embedType  
};

type record EmbedType2 {  
 Math.Point,  
 integer embedType2  
};  
  
type record EmbedType3 {  
 Math.Point,  
 integer embedType3  
};  
  
var EmbedType Et := {  
 EmbedType3:={Point:={x:=1,y:=2,z:=3},embedType3:=3},  
 EmbedType2:={Point:={x:=1,y:=2,z:=3},embedType2:=2},  
 embedType=1  
}

log(eT.x); log(eT.y); log (eT.z);   
// allowed or not? As x,y,z are defined in EmbedType2 and EmbedType3?

log(eT.EmbedType2.x); log(eT.EmbedType2.y); log(eT.EmbedType2.z);  
// explicit usage should be allowed

log(eT.Point.x); log(eT.Point.y); log(eT.Point.z);  
// allowed or not? As Point is embedded field in EmbedType2 and EmbedType3?

What is the advantage? Complex analysis necessary to examine, if a field is promoted!   
This makes the language more complex and not easier to understand.

# 8153: Extend usage of break and continue statements

Nokia – Matthias Simon

\* Allow break statements in select statements.  
\* Allow optional label to break/continue from nested loops

**label** L;

**for** (**var** i := 0; i<n; i++) {

**for** (**var** j := 0; j<m; j++) {

**if** (a[i][j] > 0) {

state := FOUND;

**break** L;

}

}

}

**label** L;

**var** **integer** i;

**for** (i := 0; i<**lengthof**(a); i++) {

**if** (a[i]) {

**break** L; *// Is NOT allowed: label does not belong to enclosing loop.*

}

}

// Possible solution can be "named cycles"  
  
while (...) {  
 while (...){  
 break w1; //or continue w1;  
 }  
} : w1  
  
// break w1; will terminate both cycles,   
// execution continues with the next instruction after w1.  
  
// continue w1; will terminate the innermost cycle and   
// takes the next iteration of the "w1" cycle.  
  
// Similar construct can be applied for nested alt instructions   
// with break/repeat "named alt").  
  
alt {  
  [] ...{  
          alt {  
            [] ... {break a1;}  
            ...  
          }  
        ...  
  [] ....  
        }  
}: a1;  
  
// "altlabels" and "cyclelabels" can be mixed, until they do not "jump" out of the scope.  
  
alt {  
 [] ... { while (...) { break a2;}}  
 ...  
}: a2

COMMENT DEVOTEAM:

“intuitive semantic” of break is:   
stop (nested) loop(s) and continue after the loop(s), i.e. a jump forward  
“intuitive semantic” of goto is: jump to the defined destination (forward or backwards)  
A backward jump using break is IMHO not intuitive.  
Suggestion: continue to use **goto** for backward jumps as then  
  
label L;  
var integer i;  
for (…) { … }

would be fine.

# 8156: Introduce user defined methods

Nokia – Matthias Simon

Uploaded initial proposal for methods, please review.  
  
Please note, after a good while of consideration I finally went with the free-floating syntax. Because it's the least intrusive. It doesn't fit perfectly with the OOP extension, but with the rest of the standard, though.  
  
What I also value high is that free-floating syntax makes it easy to retro-fit existing operations and demote them to some kind of a standard library, without breaking compatibility. For example:  
  
    external function start(in float duration) extends timer;  
    external function stop() extends timer;

Tomas Urban:

I made some changes in the document, adding an explanatory rule for imports and a couple of rules for type synonyms.  
  
The document also references a new clause 6.2.1.4 (promoted methods). Could you please add a reference to a CR where this clause was introduced?  
  
If you are happy with the changes I made or make just minor corrections, please assign the document to Jens for final reading.

Matthias Simon:

Good catch on the rule for imports, thanks.  
  
I'd like to discuss about the additional rule for type synonym, though.

Embedded fields are proposed in this CR: <http://oldforge.etsi.org/mantis/view.php?id=8154> [[^](http://oldforge.etsi.org/mantis/view.php?id=8154)]

Jens Grabowski:

TTF discussion: Open questions: (1) Is the same thing as methods in OO extension package? (2) In case a component type extends another component: How do methods behave? TTF requires answers before the feature can be introduced.

Matthias Simon:

|  |
| --- |
| For documentation purposes:  The method proposal has some benefits: \* methods for all TTCN-3 types (not only objects) \* simple syntax and semantics (no new rules for visibility, importing, ...) \* it's possible bind behaviour to a type without changing its representation.  This proposal has to be harmonized with the OOP-extension, because introducing a second, slightly different OOP-style is counter to our efforts in unifying and simplifying TTCN-3. |
|  | |

COMMENT DEVOTEAM:

Why do we need multiple kinds of object oriented methods?

# 8188: Support for function literals

Nokia – Matthias Simon

A function literal, also known as lambda function or anonymous function, is a function definition without name. Example:  
  
    var fn := function (integer x) return boolean { return x mod 2 == 0 }  
    apply(fn(23));

Nokia – Matthias Simon elaborated example:

Function literals are evaluated lazily and capture their context. This makes implementation of deferred functions feasible. For example:  
  
type function CancelFunc();  
  
testcase Example() runs on MTC {  
 // start fixtures  
 var cancel := SetupDatabase();  
  
 // do some testing  
  
 // handle graceful shutdown  
 cancel();  
}  
  
function SetupDatabase() return CancelFunc {  
 var db := DatabaseComponent.create;  
 db.start(listen());  
 return function() {  
 db.stop; // Captures db and allows to call methods, locally  
 }  
}  
  
Another use-case is parametrizing callbacks:  
  
module Service {  
 type component Component { /\* ... \*/ }  
  
 function Start(in ExitFunc ef := withSuccess) return Component {  
 var c := Component.create;  
 c.start(mainLoop(ef));  
 return c;  
 }  
  
 private function mainLoop(ExitFunc ef) runs on Component {  
 p.receive(ProcessState:{started := ?});  
 log("Service is up and running");  
   
 var ExitCode ec;  
 p.receive(ProcessState:{exited := ?}) -> value ec;  
 f(ec);  
 }  
  
 type function ExitFunc(ExitCode ec) runs on Component;  
  
 type enumerated ExitCode {  
 SUCCESS(0),  
 ERROR(1..127),  
 SIGTERM(-15),  
 SIGKILL(-9),  
 }  
  
 group helpers {  
 function withSuccess(in ExitCode ec) runs on Component {  
 if (ec != SUCCESS) {  
 setverdict(fail, "want=0, got=" & ec);  
 stop;  
 }  
 }  
  
 function withError(in ExitCode ec) runs on Component {  
 if (ec != ERROR) {  
 setverdict(fail, "want!=0, got=" & ec);  
 stop;  
 }  
 }  
  
 function withSignal(in ExitCode sig) return ExitFunc {  
 return function(in ExitCode ec) runs on Component {  
 if (ec != sig) {  
 setverdict(fail, "want=" & sig, got=" & ec);  
 stop;  
 }  
 }  
 }  
  
 function withTimeout(in float duration) return ExitFunc {  
 return function(in ExitCode ec) runs on Component {  
 // ...  
 }  
 }  
 }  
  
module Example {  
 testcase SunnyDay() {  
 var service1 := Service.Start();  
 var service2 := Service.Start(Service.withError);  
  
 // do some API testing...  
  
 all components.done;  
 }  
  
  
 testcase ResilianceTest\_ServiceCrashes() {  
 var service1 := Service.Start(Service.withSignal(Service.SIGTERM));  
 var service2 := Service.Start(Service.withTimeout(60.0));  
  
 // do some API testing...  
  
 all components.done;  
 }  
  
}

Gusztáv Adamis:

The problems in examples can be solved with the existing TTCN-3 instructions.

COMMENT DEVOTEAM:

var <missing var type???> fn := function (integer x) return …

var function fn := (integer x) return … // we suggest instead

The function SetupDatabase() returns a function-ptr, but this one is defined in the function and so in my opinion could no longer be used in the calling testcase Example()! The function that is retured uses a local variable of SetupDatabase(), that is also not accessible in the calling testcase!  
The same problem occurs with example-functions withSignal: After execution the input-parameter sig is no longer valid to execute the returned function-ptr! withTimeout:After execution the input-parameter duration (I guess it is used in return-statement) is no longer valid to execute the returned function-ptr! Missing usage of service1, service2 variables.

# 8189: Implicit Apply

Nokia – Matthias Simon

Make `apply` optional, when invoking behavior type values:  
  
    type function Handler();  
    var Handler fn := someFunction;  
    fn(); // <-- shorthand for: apply(fn());

COMMENT DEVOTEAM:

Because of backward compatibility it should be as it is, different notations for the same is not necessary! Assignment of a function-“pointer” and execution of a function-“pointer” is clearer, if we use apply!

# 8190: Expression Bodies

Nokia – Matthias Simon

Expression bodies are a shorthand for function literals. Example:  
  
var even := (integer x) => x mod 2 == 0;  
  
is a shorthand for:  
  
var even := function (integer x) return boolean { return x mod 2 == 0 }

COMMENT DEVOTEAM:

var function even := (integer x) => x mod 2 == 0;

var function even := (integer x) return boolean { return x mod 2 == 0 }

How could this kind of variable be used as actual parameter? Define especially rules for their usage? Why should variable “even” be defined without a type? The variable “even” could be assigned expression-bodies using a different formal-parameter-list.   
What about:

even := (integer x, integer y) return boolean { return (x+y) mod 2 == 0 }  
var boolean b := apply(even(5)); // causes runtime error?

# 8191: Strict Rules

Nokia – Matthias Simon

Stricter TTCN-3 language rules are beneficial for avoiding code smells. For example:

\* 8094: Provide a canonical style for source code layout

\* 8098: Mandatory module prefix for imported module definitions

\* 8099: Disallow circular imports

\* xxxx: Private as default visibility for module definitions

\* xxxx: Disallow references in pattern strings

\* xxxx: Explicit imports

\* ...

Individual rules should be optional to assure backwards compatibility. Those rule could be configured by some kind of project manifest, or file-local by pragma directives. Examples from other languages:

\* Perl: use strict;

\* Python: from \_\_future\_\_ import nested\_scopes

\* Visual Basic: Option Strict On

\* C#: #pragma warning disable 414, CS3021

COMMENT DEVOTEAM:

Canonical style does not change semantic, so why should this be enforced?  
Mandatory module prefix should only be necessary, if Identifier is not unique.  
Default should be public, as e.g. current TTCN-3 test suites do not use public in the module definition.

# 8102: Optional semicolon

Nokia – Matthias Simon

Current semicolon rules are context sensitive and are impossible to be defined properly in grammars without not-predicates/look-ahead (yacc, Bison, ...).  
  
I propose to make the semicolon optional.  
  
Open question: How could we handle conflicts such as:  
  
    alt {  
        var integer a[x]  
        [y] p.receive  
    }

COMMENT DEVOTEAM:

alt {  
 var integer a[x][y] p.receive  
 // is same as above, but is it now a two-dimensional array???  
 // see our comment below:  
}

No, semicolon should at least separate statements or terminate a statement. The optional use in TTCN-3 means a function-definition (and other definitions) must not terminate with semicolon. Statements but the least of a StatementBody must be terminated by a semicolon. In the example above either after a[x] (one dimensional variable) or after [y] (2 dimensional variable ???)

By the way,  
there is a nice suggestion how to simplify the German language in 5 steps:  
<http://www.wunderland-deutsch.com/post/Vereinfachung-der-deutschen-Sprache-in-nur-5-Schritten>

# 8194: Optional Names for Formal Parameters

Nokia – Matthias Simon

The current TTCN-3 specification requires formal parameters to have names, even when they are not used in the code, which can create confusion when using assignment notation and make the code harder to read.  
  
To solve this problem, I propose that we allow developers to omit the names of formal parameters.  
  
This change would reduce noise in the code, as unnecessary parameter names would no longer be required. It would be especially useful when specifying built-in functions and interfaces, as it would make the code more readable and easier to understand.

COMMENT DEVOTEAM:

Formal parameters without names should not be allowed.   
Reduce noise of the code: it would help to remove not used parameters.

# 8196: Redefining Macros as Predefined Constants

Nokia – Matthias Simon

A problem with the current TTCN-3 specification its diversity of the language.   
To address this, I propose that we define macros such as   
\_FILE\_\_, \_\_SCOPE\_\_, and others, as predefined constants.   
This will make the language model a little smaller.

COMMENT DEVOTEAM:  
Macros \_\_LINE\_\_, \_\_SCOPE\_\_, … are NO constants!

# 8197: Automatic Alternative Selection for Unions

Nokia – Matthias Simon

Examples:

type union Payload {  
 charstring name,  
 integer number,  
}  
  
var Payload p := "Joseph Malik"; // implicit field name is chosen automatically  
  
// Question: automatic selection also for right hand side contexts?  
if (p != 23) {  
 setverdict(fail, "unexpected payload");  
}

COMMENT DEVOTEAM:  
This would lead to a maintenance nightmare for huge test suites   
e.g.: if   
 var Payload xyz := “…”   
is used multiple times throughout the suite and   
then type Payload must be extended by a second charstring field!