A Type System for TTCN-3

Jacob Wieland
Baltasar Trancón y Widemann
Markus Lepper
{ugh,bt,lepper}@cs.tu-berlin.de

Technische Universität Berlin
FAK IV / ISTI / ÜBB
Franklinstr. 28 / FR 5-13 / 10587 Berlin

Abstract. The introduction of TTCN-3 into the world of imperative protocol compliance testing is an important step towards the application of *formal methods* in protocol testing. Since the TTCN-3 standard[2] comes without an explicitely defined type system, this paper attempts to close this gap in the specification, permitting discussions in the community and the evaluation of compilers on a more exact basis.
The type system presented in this paper has been implemented in the TTthree (TTCN-3 to Java) compiler.

TTCN-3, compiler construction, type system

1 INTRODUCTION

The introduction of TTCN-3 into the world of imperative protocol compliance specification and testing is an important step towards the application of *formal methods* in protocol testing. Besides *formal methods* being a field of research and academic culture on its own (and maybe a source of horror for the practitioner) indeed it simply means consequent application of *mathematics* whenever it seems promising [8].

While it is possible to give exact semantics only to certain variants of graphical languages[1], in the field of textual languages the mathematical devices are well understood, mature and even frequently applied in industrial practice (e.g., parser generators, attribute grammar systems; see [1]). Therefore, the transition from the graphical / tabular format of TTCN-2 to the programming language like text based source format of TTCN-3 is a deserving step.

A *protocol specification* is a set of rules describing the set of all permitted behaviors of a system as sequences of ingoing and outgoing PDUs. In the *imperative* approach, as in TTCN-3 and its predecessors, this is achieved by giving "examples": Sequences of *stimuli* sent to the SUT are defined, together with

[1] e.g. consider that the problem of recognizing *graph isomorphism* is not yet, and may never be, solvable efficiently in general.

the allowed reactions of the SUT. These are compared to the actually received reaction by pattern matching or, more generally, constraint application.

The disadvantage of this approach is obvious: In contrast to formalisms with *denotational semantics* it is hard work to find out whether a given test suite is a complete cover of possible classes of situations, or even if it touches all important cases at least once. The advantage is *immediate executability*, which the other specification techniques do not provide in general.

So each TTCN-3 text has a twofold meaning: It is intended to be a human-readable specification, and it is a program which can be executed for automized testing of hardware and software.

For the former purpose, i.e., simplicity of understanding, TTCN-3 has been designed to resemble common and established imperative languages like C and PASCAL in multiple ways. For the same reason, modern and more complex namespace concepts, such as *overloading*, *information hiding* or *inheritance*, are not supported. On the other hand, to allow for more abstraction from the implementation details of test systems, some sorts of *active* entities (components with ports as well as timers) along with an SDL[6]-like, message-based communication model have been added to the language for the specification of parallel, synchronous and asynchronous behavior.

While for the latter purpose of automatic executability it is sufficient to implement a compiler on an intuitive basis, the claim of being a *specification language* does require to give exact semantics to the language. A first step towards this goal is a *type system*, which gives a kind of *coarse* semantics to every construct of the language's syntax. It allows *a priori* detection of some classes of erroneous run time situations and undefined expressions.

The aim of this report is to present such a type system for TTCN-3 which will allow to define the denotational semantics of that language, laying the basis for mathematical reasoning about specifications formulated in TTCN-3. Whenever automized reasoning and processing of testsuites will appear on the agenda (as done for TTCN-2 by [7]), as much static information as possible should be available.

First, we define an expression language for TTCN-3 types, values and operations[2]. Then, we introduce the different kinds of type compatibility and afterwards describe the subtyping relationship.

As means of denotating mathematical structures we use the algebraic language OPAL [9]. The most important constructs used in the following are shortly explained in Appendix B.

[2] We allow slight abstractions and comprehensions concerning some idiosyncratic distinctions found on the syntax level of TTCN-3.

2 Types of TTCN-3

The type system induced by the TTCN-3 standard is characterized by the following features:

- parametrization of types and data[3],
- usability of data types as *template* values[4],
- semantic subtypes (typing information available only at run time),
- disjunctive types,
- the special value `omit` implicitly added to any type definition.

The types in TTCN-3 can be categorized into *value types* and *behavioral types*; the former with *structural*[5], the latter with *operational* subtype compatibility.

3 Expressions of TTCN-3

- The *expression* language of TTCN-3 has three categories: types, values and operations.

```
TYPE expression ==
     type(expr : type)
     value(expr : value)
     operation(expr : operation)
```

- The *type expression* language can be described by the following data type[6]

```
TYPE type ==
     all   void   type
     bool   verdict   timer   integer   float
     character(kind : string_kind)
     struct(kind : struct_kind , fields : array[declaration])
     list(kind : list_kind , element_type : type)
     ⋃(types : array[type])     -- disjunction
     ∏(types : array[type])     -- product
     io(kind : io_kind , input : type , output : type)
     modified(modifier : modifier , original_type : type)
     constraint(original_type : type , predicate : expression)
     runtime(condition : constraint , known_type : type)
```

[3] Although it would have been easily feasible, the designers of TTCN-3 restrained from applying parametrization consequently to subtypes and behavioral types.

[4] Template matching mechanisms are a field of research all of its own so this report will mostly abstract from any specifics in this area.

[5] Here, the standard is more restrictive than necessary from a mathematical viewpoint; e.g., a `set of` type is not considered compatible with corresponding `record of` type.

[6] The leaving out of the constructors for the `signature` and `enumerated` types of TTCN-3 is intentional. They can be modelled by using the types defined here, but this is outside the scope of this paper.

```
TYPE string_kind == bit   hex   octet   char   universal
TYPE struct_kind ==
     record   set   union   mapping component
TYPE list_kind == string   record   set
TYPE io_kind == generic   behavior   port
TYPE modifier == optional   template   constant   variable
TYPE declaration == _:_ (name : identifier , type : type)
```

– The *value* language describes numbers, lists, strings, structures, templates, generics and behaviors (procedures [7]), as well as undefined, void and omitted values.

```
TYPE value ==
     number(kind : type , number : number)
     list(elements : array[expression])
     string(kind : character , string : string)
     mapping(fields : array[definition])
     template(template_type : type , predicate : expression)
     generic(args : array[declaration] , result : expression)
     behavior(runs_on : type , statement : statement)
     null   nothing   omit
TYPE definition == _:=_ (name : identifier , value : expression)
```

– The *operation* language contains declared identifiers, generic application (instantiation), as well as item and field selection operations [8] (for lists and structures respectively), evaluation of templates to unique values and creation of components.

```
TYPE operation ==
     id(id : declaration)
     _ @ _ (operation : expression , argument : expression)
     _ [ _ ](list : expression , index : expression)
     _._ (operand : expression , selector : identifier)
     isvalue(operand : expression)
     valueof(operand : expression)
     create(component_type : type)
```

4 Subtyping

Typical protocol testsuites contain numerous types and even more data templates. To keep their definition manageable, TTCN-3 supports generic templates and *structured* types. Furthermore for generalization, abstraction or compatibility purposes it is often desirable or sometimes even necessary to use values of one type as values of another type. This induces a *subtyping* relation.

[7] Note: a TTCN-3 `function` is represented as a `generic` yielding a `behavior`

[8] The `isvalue` operation subsumes both the `ischosen` and `ispresent` operations of TTCN-3

4.1 Subtyping For Structured Types

Concerning the structured types of TTCN-3, one type is a subtype of another, if both types have equal[9] toplevel structure and if the types of all aggregated components of the first type are subtypes of their counterparts in the second[10].

Fig. 1 Generic Record Types and Templates

```
type record GR1(T1, T2) { T1 a, T2 b optional }
type record GR2(T1, T2) { T1 b, T2 a } // GR2(T1,T2) is also GR1(T1,T2)
type GR1(integer, integer) R1; // R1 is GR1(integer, integer)
type GR2(integer, integer) R2; // R2 is GR2(integer, integer) is also R1
type record R3 { integer c, integer d } // R3 and R2 are equivalent
template R1 r1(integer i1, template integer i2) := { a := i1, b := i2 }
template R2 r2(integer i1, integer i2) := { i1, i2 }
template R1 r3 := r2(1, 2); // correct
template R2 r4 := r1(1, 2); // warning, but correct
template R3 r5 := r1(1, *); // warning and runtime-error
template R3 r6 := r2(1, 2); // correct
```

Figure 1 defines two generic record types GR1 and GR2 and two instances of these R1 and R2. It also defines a another record type R3. R1, R2 and R3 all have the same structure and it is possible to use every item of type R2 or R3 as items of the other one or of type R1. Those values of type R1 where the second field is omitted can not be used as values of type R2 or R3, but it is possible for all other values.

4.2 Operational Subtyping

If two entities allow the same operation to be applied to them yielding a result of the same type, they are operationally compatible in regard to that operation. If all entities of one type are operationally compatible with all entities of another type in regard to an operation then these types are operationally compatible in regard to that operation. Finally, if one type is operationally compatible with another type on *all* that type's operations, then the first type is a subtype of the other type.

4.2.1 Port types In figure 2, the port type P2 is operationally compatible with the port type P1 in regard to all operations (i.e., connect, map, send and receive). Thus, it is a subtype of P2. It additionally allows reception of all values of type integer or those of type R1 that are not of type R2.

[9] Note: TTCN-3 uses diverting notions of structure equality for different type constructs.

[10] Note: this does not generally imply memory layout compatibility.

Fig. 2 Port Types

```
type port P1 { in R2; out R3 }
type port P2 { in R1, integer; out R2 } // P2 is also P1
```

4.2.2 Component Types The component types C2 and C3 in figure 3 allow all operations (i.e., selection of port p with a result of a port of a subtype of P1) that are allowed on the component type C1. Again, they allow additional operations (by having additional or more specific fields). The component type C4 is the most general type that is operationally compatible to C2 and C3.

Fig. 3 Component Types

```
type component C1 { port P1 p }
type component C2 { port P1 p; timer t } // C2 is also C1
type component C3 { port P2 p } // C3 is also C1
type component C4 { port P2 p; timer t } // C4 is also C3 and C2
```

4.2.3 Generic and Behavior Types Generics and behaviors are also examples for operationally compatible entities. Behaviors are activated as defaults or invoked by other behaviors and must be able to run on the component they are executed on. In figure 4, the instantiated functions f1 and f2 can be run on instances of all 4 component types because they are all specializations of C1.

To conform to operational compatibility, parameters which are used to instantiate a generic must be of a subtype of the generic's domains.

Fig. 4 Functions, Testcases and Operations

```
function f1(integer a) runs on C1 {
    var R2 r;
    p.receive(r1(a, *)) -> value r; // compiler warning
    p.send(r2(r.a,r.b)); // possible runtime error
    p.receive(float:*); // compiler warning/error
}
function f2(integer a) runs on C1 {
    p.send(r2(a, - )); // compiler error
}
testcase t1(integer a) runs on C2 system C1 {
    var C3 c := C4.create; // correct
    map(c:p, system:p); // correct
    connect(mtc:p, c:p); // correct
    c.start(f2(a)); // compiler warning
    f1(a); // correct
}
```

In figure 1, the generic template `r1` takes an `integer` value and an `integer` template as parameters and yields a template of type `R1` as result. `f1` in figure

4 on the other hand is a generic template which takes an `integer` value and yields a `behavior` which can be run on any component of type `C1` and returns `nothing`.

5 Types and Their Relation

5.1 The Subtyping Relation

It is possible to define a function $\sqsubseteq$ which computes a constraint that must be solvable for the types to be in a subtype relation. The subtyping relation is an ordering relation[11].

FUN $\_ \sqsubseteq \_ : \texttt{type} \times \texttt{type} \to \texttt{constraint}$
THEORY `Order[` $\sqsubseteq$ `]` -- *reflexive, transitive, antisymmetric*

5.1.1 Subtyping Constraints A constraint in this context shall be a logical formula of boolean expressions, i.e. predicates.

For description purpose, we introduce logical quantors on arrays, lifting of expressions and equality to constraints.

FUN $\forall \exists$: $\texttt{array}[\alpha] \times (\alpha \to \texttt{constraint}) \to \texttt{constraint}$
FUN $\ll \_ \gg : \texttt{expression} \to \texttt{constraint}$
FUN $\_ = \_ : \alpha \times \alpha \to \texttt{constraint}$

5.2 Special Types

The special types `all`, `none`, `void` and `type` are not really types that appear in code of the TTCN-3 language, but they exist implicitely. The `void` type is used as the `output` type of `behavior` types of behaviors that return `nothing`. The type `type` is the type of type expressions.

All types are subtypes of themselves, and are subtypes of another type, if their carrier sets are subsets. The largest type `all` is supertype to all types (i.e. only operations which are allowed on all types — like equality — are allowed on that type). The smallest type `none` represents the type that is a subtype to all types.[12] Since there can be no values of type `none`, all operations of all types are allowed on that type. It is used for `input` and `output` types of ports.

DEF $\texttt{none} == \bigcup(\{\})$
DEF $\texttt{t}_1 \sqsubseteq \texttt{t}_2 == \texttt{t}_1 = \texttt{t}_2 \vee \ll \texttt{t}_1 \subseteq \texttt{t}_2 \gg$
DEF $\texttt{t} \sqsubseteq \texttt{all} == \texttt{true}$
LAW $\texttt{none} \sqsubseteq \texttt{t} == \texttt{true}$

[11] Therefore, special definitions would need to be added to those mentioned here to assure transitivity.
[12] which is implied by the subtype relation for disjunction types

5.3 Modified Types

The different type modifiers add roles or functionality to items of the modified types. Every type can be modified to be `optional`. Additionally, a type can be modified to be either `variable`, `constant` or `template`. `template` types are implicitely also `constant`. To ensure this, the constructor functions `template`, `constant`, `variable` and `optional` are defined in such a way that they obey these restrictions.[13].

The function `basetype` strips all modifiers, constraints and implications from a type, while the `modifiers` function computes from a given type a function which adds its modifiers to another type.

FUN `template constant variable optional basetype : type → type`
FUN `modifiers : type → (type → type)`
LAW $\forall$ `t : type . modifiers(t)(basetype(t)) = t`

An item of type `template(t)` can be used as a template of type t. It can be used for pattern matching. They cannot be used as values[14]. Every item of a non-template type can also be used as a template of that type.

An item of type `constant(t)` is a constant value of type `t`. Assignments are not allowed to it.[15] In contrast, an item of type `variable(t)` is always a reference to an item of type `t` to which assignments of values of type `t` are allowed. Every variable of a type can be used as a constant of that type. Constants and variables of a type can be used as expressions of that type. In every context (except in field declarations of value `struct` types), every expression is either a `constant` or `variable` value or a `template`.

Finally, an item of type `optional(t)` may have a value of type `t` or it may have the value `omit`.[16]

DEF $\texttt{modified(template}, \texttt{t}_1\texttt{)} \sqsubseteq \texttt{modified(template}, \texttt{t}_2\texttt{)} == \texttt{t}_1 \sqsubseteq \texttt{t}_2$
DEF $\texttt{modified(constant}, \texttt{t}_1\texttt{)} \sqsubseteq \texttt{modified(constant}, \texttt{t}_2\texttt{)} == \texttt{t}_1 \sqsubseteq \texttt{t}_2$
DEF $\texttt{modified(variable}, \texttt{t}_1\texttt{)} \sqsubseteq \texttt{modified(constant}, \texttt{t}_2\texttt{)} == \texttt{t}_1 \sqsubseteq \texttt{t}_2$
DEF $\texttt{modified(variable}, \texttt{t}_1\texttt{)} \sqsubseteq \texttt{modified(variable}, \texttt{t}_2\texttt{)} ==$
$\texttt{t}_1 \sqsubseteq \texttt{t}_2 \wedge \texttt{t}_2 \sqsubseteq \texttt{t}_1$
DEF $\texttt{t}_1 \sqsubseteq \texttt{modified(template}, \texttt{t}_2\texttt{)} == \texttt{t}_1 \sqsubseteq \texttt{t}_2$
DEF $\texttt{modified(constant}, \texttt{t}_1\texttt{)} \sqsubseteq \texttt{t}_2 == \texttt{t}_1 \sqsubseteq \texttt{t}_2$
DEF $\texttt{modified(optional}, \texttt{t}_1\texttt{)} \sqsubseteq \texttt{modified(optional}, \texttt{t}_2\texttt{)} == \texttt{t}_1 \sqsubseteq \texttt{t}_2$
DEF $\texttt{t}_1 \sqsubseteq \texttt{modified(optional}, \texttt{t}_2\texttt{)} == \texttt{t}_1 \sqsubseteq \texttt{t}_2$

[13] Their definition is trivial.
[14] e.g. as value parameters
[15] Constant values need not to be copied at runtime when passing them as a parameter that is implicitely declared constant — like generic template parameters.
[16] In TTCN-3 this is only used for optional fields in structure types, but it could be equally well used for defining optional parameters.

5.4 Number Types

Every integer can be used as a float value.

DEF $\texttt{integer} \sqsubseteq \texttt{float} == \texttt{true}$

5.5 Structured Types

There are a multitude of structure types in TTCN-3 — mostly derived from its relationship with ASN.1[5]: records, sets and unions.

5.5.1 Emptiness of Structured Types `union` types are empty if all their component types are empty. Other `struct` types are empty if at least one of their field types is empty. Thus, if a `union` type has no fields, it has no values, while other `struct` types[17] with no fields have exactly one value.

DEF $\texttt{struct(union , f)} \sqsubseteq \texttt{none} == \forall\,\texttt{(type} * \texttt{f} , \lambda \texttt{t} .\ \texttt{t} \sqsubseteq \texttt{none)}$
DEF $\texttt{struct(k , f)} \sqsubseteq \texttt{none} ==$
$\exists\,\texttt{(name} * \texttt{f} , \lambda \texttt{n} .\ \texttt{fieldtype(k , f , n)} \sqsubseteq \texttt{none)}$

5.5.2 Union, Record and Set Types The `record`, `set` and `union` structure kinds are value types. They are can only be subtypes of other `struct` types of the same kind. A `record` type is a subtype of another one if it declares the same amount of fields and for every field its type is a subtype to the field at the same position in the other type.[18] For `set` types the position of the fields is not important, but only their names. Therefore, for a `set` type to be a subtype of another one, it must declare the same fields with the same names and with subtypes of the corresponding field type in the supertype.[19]

DEF $\texttt{struct(union , f}_1\texttt{)} \sqsubseteq \texttt{struct(union , f}_2\texttt{)} ==$
$\forall\,\texttt{(name} * \texttt{f}_1 ,$
$\lambda\,\texttt{n} .\ \texttt{fieldtype(union , f}_1\texttt{ , n)} \sqsubseteq \texttt{fieldtype(union , f}_2\texttt{ , n))}$
DEF $\texttt{struct(record , f}_1\texttt{)} \sqsubseteq \texttt{struct(record , f}_2\texttt{)} ==$
$|\texttt{f}_1| = |\texttt{f}_2| \wedge \forall\,(0 \mathinner{..} |\texttt{f}_1| - 1 , \lambda \texttt{i} .\ \texttt{type(f}_1\texttt{[i])} \sqsubseteq \texttt{type(f}_2\texttt{[i]))}$
DEF $\texttt{struct(set , f}_1\texttt{)} \sqsubseteq \texttt{struct(set , f}_2\texttt{)} ==$
$|\texttt{f}_1| = |\texttt{f}_2| \wedge$
$\forall\,\texttt{(name} * \texttt{f}_1 ,$
$\lambda\,\texttt{n} .\ \texttt{fieldtype(set , f}_1\texttt{ , n)} \sqsubseteq \texttt{fieldtype(set , f}_2\texttt{ , n))}$

[17] excepting `component` types
[18] This condition could be relaxed so that the supertype can have less fields than the subtype and the subtype relation must only apply for these fields.
[19] As for records, it would be possible to relax this condition by allowing additional fields for the subtype. The positions of these additional fields are irrelevant, though.

5.5.3 Mapping Types All `record`, `set` and `union` types can be denotated the same way: as a mapping of names to values. For these denotation expressions another `struct` kind `mapping` is introduced. This meta-`struct`-kind is a subtype to some of the other `struct` types under different circumstances.

```
DEF struct(mapping, f1) ⊑ struct(union, f2) ==
    |f1| = 1 ∧ type(f1[0])) ⊑ fieldtype(union, f2, name(f1[0]))
DEF struct(mapping, f1) ⊑ struct(k, f2) ==
    ≪ record?(k) ∨ set?(k) ≫ ∧
    ∀ (name * f1 + name * f1,
      λ n . fieldtype(mapping, f1, n) ⊑ fieldtype(k, f2, n))
```

5.5.4 Component Types The `component` structure kind represents the type of components on which testcases and functions are executed. A `component` type is more special than another `component` type if all operations that are defined on the more general type are also defined on the more special type and yield results of the same type.[20] Thus, the `component` type containing *all* components is that which has no fields.

```
DEF struct(component, f1) ⊑ struct(component, f2) ==
    ∀ (name * f2, λn . fieldtype(component, f1, n)
                    ⊑
                    fieldtype(component, f2, n))
```

5.5.5 Optional Fields The function `fieldtype` reflects the different semantics of "missing" fields. For `component` types a missing field is "possibly present, but unspecified". For `mapping` types it is "implicitely omitted". For the other `struct` types it is undefined.

```
FUN fieldtype: struct × array[declaration] × identifier → type
DEF fieldtype(kind, f, n) ==
    LET i == find(n, name * f)
    IN IF i < 0 THEN undeffieldtype(kind) ELSE type(f[i]) FI
FUN undeffieldtype: struct → type
DEF undeffieldtype(component) == all
DEF undeffieldtype(mapping) == optional(none)
DEF undeffieldtype(other) == none
```

5.6 Product Types

The type product $\prod$ is used for all kinds of inhomogenous lists, like parameter lists or initializers of inhomogenous records and sets.

[20] The only operations that are not defined on *all* `component` types are *member selection* and *execution* of functions and testcases on them.

If one of the component types of a product is `none` then the whole product is a subtype of `none`. `list` values are of product type and can be used to initialize items of `record` or `set` type. Omitting an item at the end of the list has the same semantics as setting it to `omit`. Therefore, the type of an implicitely omitted element is `optional(none)`.

DEF $\prod(\mathtt{t}_1) \sqsubseteq \mathtt{none} == \exists(\mathtt{t}_1, \lambda \mathtt{t} . \ \mathtt{t} \sqsubseteq \mathtt{none})$
DEF $\prod(\mathtt{t}_1) \sqsubseteq \prod(\mathtt{t}_2) ==$
$|\mathtt{t}_1| = |\mathtt{t}_2| \wedge \forall(0 .. |\mathtt{t}_2| - 1, \lambda \mathtt{i} . \ \mathtt{t}_1[\mathtt{i}] \sqsubseteq \mathtt{t}_2[\mathtt{i}])$
DEF $\prod(\mathtt{t}) \sqsubseteq \mathtt{struct(record, f)} ==$
$\ll |\mathtt{t}| \leq |\mathtt{f}| \gg \wedge$
$\forall(0 .. |\mathtt{f}| - 1, \lambda \mathtt{i} . \ \mathtt{componenttype(t, i)} \sqsubseteq \mathtt{type(f[i])})$
DEF $\prod(\mathtt{t}_1) \sqsubseteq \mathtt{list(k, t_2)} == \forall(\mathtt{t}_1, \lambda \mathtt{t} . \ \mathtt{t} \sqsubseteq \mathtt{t}_2)$
FUN `componenttype: array[type]` $\times$ `nat` $\rightarrow$ `type`
DEF `componenttype(a, i) ==`
IF $\mathtt{i} < |\mathtt{a}|$ THEN `a[i]` ELSE `optional(none)` FI

5.7 List Types

`list` types represent homogenous lists (i.e. lists of elements of the same type), like strings or arrays.[21] `list` types of every kind are compatible with `list` types of the same kind if their element types are compatible. Also, `record` types are compatible with `list` types of `record` kind if all the `record` fields are subtype to the element type of the `list` type. The same applies for `set` types and those `list` types of `set` kind.

DEF $\mathtt{list(k_1, t_1)} \sqsubseteq \mathtt{list(k_2, t_2)} == \mathtt{k_1} = \mathtt{k_2} \wedge \mathtt{t_1} \sqsubseteq \mathtt{t_2}$
DEF $\mathtt{struct(record, f_1)} \sqsubseteq \mathtt{list(record, t_2)} ==$
$\forall(\mathtt{f}_1, \lambda \mathtt{f} . \ \mathtt{type(f)} \sqsubseteq \mathtt{t}_2)$
DEF $\mathtt{struct(set, f_1)} \sqsubseteq \mathtt{list(set, t_2)} ==$
$\forall(\mathtt{f}_1, \lambda \mathtt{f} . \ \mathtt{type(f)} \sqsubseteq \mathtt{t}_2)$

5.8 Disjunction Types

For the `input` and `output` types of `port` types[22] a type disjunction $\bigcup$ is needed. Such a type encompassing all items of the combinded types. If a disjunction of types shall be a subtype of another type, then all the types of the disjunction have to be a subtype of that type. In the case that a type should be the subtype of a disjunction type (of constraint types), the matter is more complicated. [23]

DEF $\bigcup(\mathtt{l}) \sqsubseteq \mathtt{t}_1 == \forall(\mathtt{l}, \lambda \mathtt{t}_2 . \ \mathtt{t}_2 \sqsubseteq \mathtt{t}_1)$
DEF $\mathtt{t}_1 \sqsubseteq \bigcup(\mathtt{l}) == \exists(\mathtt{l}, \lambda \mathtt{t}_2 . \ \mathtt{t}_1 \sqsubseteq \mathtt{t}_2) \vee \ll \mathtt{t}_1 \subseteq \bigcup \mathtt{l} \gg$

[21] `record of` and `set of` types
[22] as well as for the `exceptions` type of procedural messages which are not explicitely covered by this paper
[23] This is an unsolved problem in general, but doesn't appear very often in practice.

5.9 I/O Types

The io-types `generic`, `behavior` and `port` are contravariant in regard to the subtype relation on their `input` types and covariant on their `output` types.

DEF $\texttt{io(generic}\,,\,\texttt{i}_1\,,\,\texttt{o}_1\texttt{)} \sqsubseteq \texttt{io(generic}\,,\,\texttt{i}_2\,,\,\texttt{o}_2\texttt{)} \;==$
$\texttt{i}_2 \sqsubseteq \texttt{i}_1 \wedge \texttt{o}_1 \sqsubseteq \texttt{o}_2$

DEF $\texttt{io(behavior}\,,\,\texttt{i}_1\,,\,\texttt{o}_1\texttt{)} \sqsubseteq \texttt{io(behavior}\,,\,\texttt{i}_2\,,\,\texttt{o}_2\texttt{)} \;==$
$\texttt{i}_2 \sqsubseteq \texttt{i}_1 \wedge \texttt{o}_1 \sqsubseteq \texttt{o}_2$

DEF $\texttt{io(port}\,,\,\texttt{i}_1\,,\,\texttt{o}_1\texttt{)} \sqsubseteq \texttt{io(port}\,,\,\texttt{i}_2\,,\,\texttt{o}_2\texttt{)} \;==$
$\texttt{i}_2 \sqsubseteq \texttt{i}_1 \wedge \texttt{o}_1 \sqsubseteq \texttt{o}_2 \wedge \texttt{o}_2 \sqsubseteq \texttt{o}_1$

This means that a `generic` that has a result of type `integer` for every `float` value can also be seen as a `generic` that results in a value of type `float` for every `integer` value.

For `behavior` types, the `input` type is the `component` type the behavior runs on, so every behavior that runs on a certain type of component is also a behavior which runs on all components which are more special (i.e., have specializations of all port and constant types in corresponding fields) of that `component` type.

For port types, the situation is a bit more complex. For the `input` the same applies as for all `io` types: every `port` which is able to receive all values of type R1 or integer is also able to receive all values of type R2 (if R2 is a subtype of R1).

But, if a specific port is used as a more general port, it still is not allowed to send more data than in its specific context. Thus, the `output` also behaves contravariant for `port` types. As the `output` of a port must be a subtype to the `input` of all ports it is connected to, `port` types behave also *covariant* in regard to the connect operation. Together, these conditions imply that the `output` types of `port` types must be equivalent for them to be in a subtyping relation.

5.10 Constraint and Runtime Types

`constraint` types are only easy to handle if they appear as a subtype in a subtype relation. For the computation of other instances of this subtyping system, one is forced to either limit the computation at this point (i.e. stick to the computation for some manageable constraints), use heuristics or more general subtyping constraint solvers. [24]

DEF $\texttt{constraint(t}_1\,,\,\texttt{p)} \sqsubseteq\ \texttt{t}_2 \;==$
$\texttt{t}_1 \sqsubseteq \texttt{t}_2 \vee \ll \texttt{constraint(t}_1\,,\,\texttt{p)} \subseteq\ \texttt{t}_2 \gg$

`runtime` types are types that are inferred under certain constraints which are not necessarily true and can be checked only at runtime. The constraining conditions are propagated to the resulting constraint of the subtyping relation.

[24] Although it is outside the scope of this report, it has been successfully implemented for the template pattern language of TTCN-3 with the exception of the `complement` pattern for which computation of a proper type, and thus subtyping, is very difficult.

```
DEF runtime(p , t1) ⊑ t2 == p ⟹ t1 ⊑ t2
DEF t1 ⊑ runtime(p , t2) == t1 ⊑ t2 ∧ p
```

6 CONCLUSION

In this paper, we have presented a type system for TTCN-3. By extrapolating from the statements found in the standard document concerning type compatibility, we have discovered the two notions of structural and operational subtyping that are complementary and interact in a complex way, e.g., when applied to record types. We have defined a unified mathematical model covering both. Thus, type correctness for data and operations can be inferred with a single algorithm (see Appendix A). Our model also outlines various possibilities for generalization of the TTCN-3 typing rules, which contain some semantically unnecessary restrictions.

ACKNOWLEDGEMENTS

The work presented herein contains the consequences drawn from a common project with Ina Schieferdecker, Theofanis Vassiliou-Gioles and others from TestingTechnologies, Berlin. Thanks also to all in the TTCN-3 mailing list.

Appendix A Type Analysis Algorithm

```
FUN conditional : constraint × type → type
DEF conditional(true , t) == t
DEF conditional(p , t) == runtime(p , t)
DEF type(type(t)) == type
DEF type(operation(op)) = type(op)
FUN type : operation → type
DEF type(id(i : t)) == t
DEF type(op @ arg) ==
    IF  io?(t) and generic?(kind(t))
      THEN  conditional( type(arg) ⊑ from(t) ∨
                         ≪ arg ∈ from(t) ≫, to(t))
    FI WHERE t == basetype(type(op))
DEF type(e . s) ==
    IF  modified?(type(e))
      THEN  modifiers(type(e))(type(valueof(e) . s))
    IF  struct?(t)
      THEN  fieldtype(kind(t) , fields(t) , s)
    FI WHERE t == basetype(type(e))
```

```
DEF type(e[i]) ==
    IF modified?(type(e))
       THEN modifiers(type(e))(type(valueof(e)[i]))
    IF list?(t)
       THEN conditional( type(i) ⊑ integer ∨
                         ≪ i ∈ integer ≫, element_type(t))
    FI WHERE t == basetype(type(e))
DEF type(valueof(e)) == constant * basetype(type(e))
DEF type(isvalue(e . s)) ==
    IF struct?(basetype(type(e))) THEN constant(bool) FI
```

Although there are a lot more statements in TTCN-3, we mention here only some where non-trivial subtyping restrictions apply[25]. The constraints of the TTCN-3 standard are easily translated into a typechecking function `check`.

```
TYPE statement ==
      connect(from : expression , to : expression)
      invoke(on : expression , behavior : expression)
     ...
FUN check : statement → constraint
DEF check(connect(p1 , p2)) ==
    IF io?(t1) ∧ port?(kind(t1)) ∧ io?(t2) ∧ port?(kind(t2))
       THEN output(t1) ⊑ input(t2) ∧ output(t2) ⊑ input(t1)
    FI WHERE (t1 , t2) == (basetype(type(p1)) , basetype(type(p2)))
DEF check(invoke(c , b)) ==
    IF struct?(t1) ∧ component?(kind(t1)) ∧
       io?(t2) ∧ behavior?(kind(t2))
       THEN t1 ⊑ input(t2)
    FI WHERE (t1 , t2) == (basetype(type(c)) , basetype(type(b)))
```

Appendix B Opal Notation Guide

Fig. 5 Keywords

FUN	declaration of an item's functionality
DEF	definition of an item
TYPE	declaration of a data type (constructors, selectors, discriminators)
LAW	proposition about defined items
THEORY	complex generic proposition (set of laws)

[25] The *invoke* statement executes a behavior on a component.

Fig. 6 Functions

_ * _	application of a function to all elements of a structure
_ ∘ _	function composition
{_}	denotation of arrays

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers - Principles, Techniques and Tools*. Addison-Wesley, 1986.
2. ETSI. *[DES/MTS-00063-1] TTCN-3: Core Language*, 2000.
3. J. Grabowski and D. Hogrefe. Towards the Third Edition of TTCN. In *Testing of Communicating Systems (Proceedings of the TestCom 1999)*. Kluwer Academic Publishers, 1999.
4. J. Grabowski, A. Wiles, C. Willcock, and D. Hogrefe. On the Design of the New Testing Language TTCN-3. In *Testing of Communicating Systems (Proceedings of the TestCom 2000)*. Kluwer Academic Publishers, 2000.
5. ISO/IEC. *[8824] Abstract Syntax Notation One*, 2000.
6. ITU-T. *[Recommendation Z.100] Specification and Description Language*, 2000.
7. C. Jard, T. Jéron, and P. Morel. Verification of Test Suites. In *Testing of Communicating Systems (Proceedings of the TestCom 2000)*. Kluwer Academic Publishers, 2000.
8. D. L. Parnas. Verbal statement in invited talk (not in proceedings). In *Third International Conference on Formal Engineering Methods*. York, 2000.
9. P. Pepper. The Programming Language OPAL (5th corrected edition). Technical Report 91–10, TU Berlin, June 1991.