# Extension of interface signature descriptions for automatic test generation

Chernov E.

***Abstract*—In the paper the problem of extension of the standard information taken from signatures of program interfaces, for a possibility of automatic generation of the tests calling chains of such interfaces with correct parameters and in a correct environment is considered. This problem is actual at testing systems with greater number of interfaces (more than 1000) in conditions of the limited resources on creation of tests. For the description of the extended data the so-called *specialized types* specifying semantics of types of objects, returned values and parameters are entered. With their help it is possible to add the initial description of signatures of interfaces for effective generation of tests. Various kinds and properties of specialized types are considered. The aspects accelerating input of necessary additional data for groups of interfaces are discussed, too. The example of realization of storage of the extended information on interfaces in a database of Linux Standard Base standard is shown.**

***Index Terms*—Automatic testing, Software testing, Standardization**

## I. Introduction

At testing the program systems consisting of the big number of the interfaces (API) in conditions of limited resources on creation of tests, use of the automated methods and the tools allowing quickly and in a plenty to create the tests checking at least the minimal functionality is necessary. Tests thus can not check a full correctness of corresponding interfaces, but can reveal cases of destruction of system, and at successful passage of the test - to guarantee its minimal working capacity.

To create though also the elementary, but the correct test automatically, it is necessary to have the formal description of rules of a call of tested interface. The signature of interface (name, types of parameters and returned value) can be received automatically completely in the different ways: from header or binary files or from existing databases. However in most cases types of parameters and returned values in the signature are too general. For an example we shall consider the function *arcsin(double)*. The signature allows to call it on any parameter of double type. However from mathematical features of this function follows that value of actual parameter on absolute value should not surpass 1. Otherwise there will be an exception, and it will be impossible to consider such call normal (within the limits of testing the minimal functionality). Therefore for maintenance of correct formation of value of parameter by a call of such interface it is necessary to set either formal restriction (x >= -1 && x <= 1) or simply concrete value, satisfying to this restriction (for example, x = 0,5). Rules of designing of correct value for complex types can get more complex kind, for example for designing some parameters it is necessary to call other interfaces or even the whole chains of interfaces. Notice that, besides the description of properties of parameters, for generation of tests it is necessary to describe also a condition on returned value and on object (when target interface is a method of a class).

In the given paper necessary expansions of standard information containing in signatures are considered. It allows to generate a chain of initialization of a correct environment and parameters for a call of target interface automatically completely and to check up a base correctness of its performance. The central elements of this expanded information are *specialized types* which comprise specification of semantics of standard types of objects, parameters and returned values and allow designing corresponding correct values automatically. One specialized type can be attached to various parameters and returned values of various interfaces, that is it can be reused.

The information from the signatures of some interface, added by specialized types, allows building automatically completely tests of working capacity for this interface. However describing of specialized types and its binding to particular interface is a manual work. Therefore it is very important to provide effective creation new and reusing already created specialized types for different interfaces. Because of it, the certain mechanisms for the specialized types are entered. These mechanisms allow to reduce quantity of created types and to increase speed of their binding to interfaces.

The paper consists of four sections. Terms for a designation of attributes (properties) of specialized types are entered in the first section. In the second section the basic attributes necessary for automatic generation of tests are considered. And in the third - properties and mechanisms, allowing to accelerate creation of tests for the big number of interfaces. In the fourth part concrete realization of storage of the expanded information and specialized types on an example of expansion of a database of LSB standard [1,2] is described.

E. S. Chernov, Moscow (e-mail: ches@ispras.ru).

## II. Specialized Types

For definition of some specialized type, it is necessary to list values of its all attributes. The brief description of each of them is given below:

- *Name* - reflects the basic purpose of creation of the specialized type.
- *Data type* - type that is specified by the specialized type.
- *Base type* - specialized type which is inherited by the given specialized type.
- *Kind* – there are 3 kinds of specialized types: normal, once-only, common for parameters and common for returned values.
- *Value* – the value by which the parameter using the specialized type is initialized.
- *Constraint* – it defines the condition for returned value.
- *Init/final code* - program code which is necessary for inserting accordingly before and after a call of interface.
- *Auxiliary code* - the program code which is made out in the form of a set of functions and is used for initialization of objects. Besides there is an opportunity of reusing already created specialized types in it.
- *Proxy-value* – it's useful in case of dependence of parameters from each other. In this case one specialized type specifies several parameters at once. All dependences are stored in proxy-values, each of which are used by one parameter and can have the reference to other parameters.

The *name* and *data type* attributes distinguish specialized types from each other, i.e. for one data type cannot be two different specialized types with the identical name.

The *base type* attribute defines the attitude of inheritance between specialized types. Any other attitudes between them it is not stipulated.

For the description of *value*, *constraint*, *init*, *final* and *auxiliary code* attributes should be used the programming language. In the given paper C++ language is used as an example.

For the definition of specialized type it is necessary to list values of its all attributes. Further in the paper, its nonempty attributes will be listed for definition of concrete specialized type. *Name* – is an exception, since it is not essential.

## III. Main Attributes of Specialized Type

In the given section attributes of the specialized type, allowing storing information which is necessary for automatic generation of tests, are described.

### A. Value of specialized Type

The simplest way to satisfy to constraint on entrance value of parameter is the concrete notation of value for this parameter. It is stored in *value* attribute of specialized type.

The several examples of specialized types with correct values are shown below:

1) *value* = 3,
2) *value* = "*Test string",*
3) *value* = *true*.

The following program lines are corresponding to these specialized types:

```
Par_N = 32;
Par_N = "Test string";
Par_N = true;
```

Here N – is a number of parameter, which is using the specialized type with corresponding value.

*Value* can be the function defined in an auxiliary code of specialized type, for example: *value = create_param()*. The following line in generated program corresponds to it:

```
Par_N = create_param();
```

If the name of object is necessary then it's possible to refer to it by special structure: $obj. It is replaced by the name of object in the generated program. For example, for *value = $obj.first()* the next code is generated:

```
SomeClass Obj;
…
OtherClass Par_N = Obj.first();
…
```

As a rule, there is not a unique value for parameter with which it is possible to call interface, therefore one specialized type can have few values. It will lead to generation of several tests for one interface.

### B. Constraint of Specialized Type

Besides entrance values for parameter, the specialized type can also store some constraints on parameter. This is a logic expression which is true at correct work of target interface. As well as for value of specialized type, special structures for the description of constraints are used:

- *$obj* – is replaced by object's name.
- *$0* – is replaced by the name of parameter's variable.

Several examples of such constraints: "*$0 == 1*", "*$0 != NULL*", "*$obj.isEmpty() == false*". These constraints are used by conditional operator in generated program:

```
SomeClass Obj;
…
Obj.someMethod(Par_1, Par_2, …);
If (!(Par_1 == 1)) {
/* printing error message:"Failed constraint $0 == 1" */
}
If (!(Par_2 != NULL)) {
/* printing error message:"Failed constraint $0 != NULL"*/
}
If (!(Obj.isEmpty() == false)) {
/* printing error message:"Failed constraint
  $obj.isEmpty() == false" */
}
```

In this example the specialized type with the first and the second constraints is used by the first and the second parameters of interface. Specialized type with the last constraint is used by object. If the result of one of these logic expressions is false, then the corresponding error messages will be printed.

### C. Init and Final Code

*Init code* is a program code that is inserted up before calling of target interface and *final code* – after. As for constraint, $0 and $obj are used for their description. For example, an array can be filled by 10 elements in the *init code*:

```
for (int i = 0; i < 10; i++) {
      $0.append(i);
   }
```

The following code corresponds to it in generated program:

```
…
SomeClass Obj;
// Init code
for (int i = 0; i < 10; i++) {
   Par_1.append(i);
}
// Call of target interface
Obj.someMethod(Par_1,…);
…
```

In final code it's possible to close a file, for example:

```
close($0);
```

Corresponding generated code:

```
…
Par_1 = fopen("test.cpp", "r+");
Obj.someMethod(Par_1);
// Final code
close(Par_1);
…
```

### D. Auxiliary Code

Set of additional functions, which it is possible then to use in other attributes of specialized type (including *value*), is made out in the *auxiliary code* attribute. At test generation, definition of such functions are located in the beginning of a file with initial codes of the test in which there is a call of the target interface using given specialized type. Thus, unlike an *init code*, the *auxiliary code* is not duplicated for various parameters within the limits of one test. Besides it can contain the additional structures specifying to the generator to initialize a variable of certain type or to call interface with automatically initialized correct parameters:

- *$(type)* – the instruction to the generator to create a variable of type '*type*'.
- *$[function]* – the instruction to the generator to initialize and call corresponding function.

These structures allow to lower time of creation of specialized type and quantity of mistakes essentially since the volume of a code which the developer should write decreases. Besides it is not necessary to distract on learning, how to initialize the certain data type correctly or with what parameters to call an interface. Reusing of the information brought earlier for these interfaces or data types allows to not think of it. Besides such references, but not a concrete program code, allows changing a way of initialization of interface or a class only in one place, instead of in all specialized types which use them.

It's impossible to use $0 and $obj structures here, since the *auxiliary code* is not bound to concrete parameter. However if necessary it is possible to get access to the necessary parameters, having transferred them as parameters of function defined in the *auxiliary code*.

Often it is necessary to call some set-methods of an object for its correct initialization. They can define as simple properties of object (enabled - disabled, visible - not visible, etc.), and complex, demanding initialization of other objects (a color, a font, an icon, a cursor, etc.). For example, it is possible to write the following auxiliary code in the specialized type with *value="create_SomeClass()"*:

```
SomeClass* create_SomeClass() {
   SomeClass* Obj = new SomeClass();
   Obj->setEnabled(true);
   Obj->setFont( $(QFont *) );
   Obj->setIconSet( $(QIconSet) );
   return Obj;
}
```

The special structure in the setFont() and setIconSet() methods specifies to the generator to initialize the values of QFont* and QIconSet types itself.

If the created specialized type to set for object of 'SomeClass' class we shall receive the following program:

```
static const char * const XPM[]={
"16 15 8 1",
"a c #cec6bd"
}
SomeClass* create_SomeClass(SomeClass* Obj) {
   Obj->setEnabled(true);
   Obj->setFont(new QFont("Times", 10, Bold));
   QPixmap Par_1_1(XPM);
   QIconSet Par_1(Par_1_1);
   Obj->setIconSet(Par_1);
   return Obj;
}
int main() {
   SomeClass* Obj = create_SomeClass();
   // Call of target interface
   …
}
```

*$(QFont *)* is replaced by line *"QFont ("Times", 10, Bold)"*, which is constructor of QFont class with parameters specified by specialized types with values *"Times"*, *10* и "*Bold*". $QIconSet) is replaced by few lines in which QPixmap object is created by global variable XPM, defined in auxiliary code of specialized type for QPixmap constructor. At a writing of this code manually it should to understand how to initialize objects of QFont, QIconSet classes and types which they depend on (QPixmap in this case). Instructions to the generator to create objects of these classes automatically help to reduce expenses of time and efforts to creation of tests for interfaces depending on them and to avoid mistakes arising at it.

### E. Proxy-value for Complex Specialized type

Usually specialized type includes the additional description for only one parameter. Thus the test using only such

specialized types can be divided into independent blocks of a code, each of which initializes own parameter. However in practice there are dependences between descriptions of parameters from each other, i.e. it is necessary to use value of one parameter in lines of a code for initialization of other parameter. Typical example of such dependence is the situation when one parameter is a string, and another is equal to length of this string. In this case the description of the first and second parameter cannot be divided into independent blocks of a code:

Par_1 = “Some String”;

Par_2 = strlen( Par_1 );

Such cases are described by *complex specialized types*. Actually, one complex specialized type contains description for several parameters. Dependent parameters unite into one parameter in extended signature of interface. All dependences between initial parameters remain only inside of specialized type, and dependences between parameters in the extended description of interface are not present. Thus the final program represents association of independent descriptions of groups of parameters.

*Proxy-value* attribute of specialized type allows to unite parameters by one description. The number of these values is equal to the number of united parameters. Every proxy-value stores an initializing string for corresponding parameter.

For the example above complex specialized type has two proxy-value: “*$1 = “Some string*” - for the first parameter and “*strlen($1)*” – for the second. Special structure *$1* is used for this description. Parameter with “*$1 = “Some string”* proxy-value has “*Some string*” value in generated program. *$1* is replaced by the name of this parameter. So the second parameter with *strlen($1)*” proxy-value has “*strlen(Par_1)*“ value in generated program.

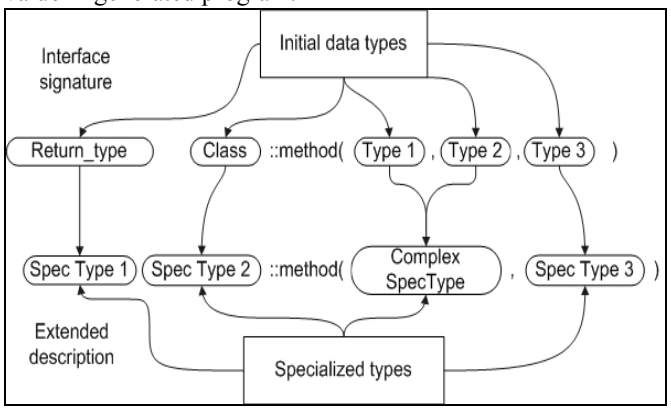


Fig. 1. Extension of interface signature by specialized and complex specialized types.

It’s possible to use some other structures instead of $1, but in any case such mechanism of references to other parameters is necessary. $1 is a reference to the first parameter here. Besides, it’s possible to use $0 and $obj structures here that replaced by a name of parameter and a name of object accordingly.

## IV. Accelerating of Tests Creation

The main advantage of automatic creation of the elementary tests is small expenses of work for one test. Manual actions are the longest part of process of tests creation. It is creation of specialized types and extension of interface descriptions (a binding of specialized types to parameters of interface). Developer does these actions by the special GUI tools that allow editing of the information about specialized types. To lower efforts to processing of each interface, some mechanisms have been entered. These mechanisms allow to decrease number of created specialized types and to facilitate interaction of developer with the GUI tools.

### A. Inheritance of Specialized Types

At creation of specialized type for some initial data type there can be a need for reuse some attributes of other, already created specialized type for similar initial data type. *Value*, *constraint*, *init* and *final code*, *auxiliary code* are the attributes that can be reused. For such reusing the mechanism of inheritance has been entered. The specialized type that inherits other type must have reference to other one in *base type* attribute. At test generation the empty attributes from the child-type are taken from parent-type. Thus, the specialized type can be created quickly – it’s enough to specify its parent, not specifying any other attributes. Such inheritance is useful at creation of specialized types for intrinsic data types (*int*, *char*, *long*, *void* *, etc.) for check of returned value. For example, to have specialized type for check on *NULL*, it is necessary to create the specialized type based on *void* * with *constraint = "$0 != NULL"*, and other specialized types of nonzero pointers can be inherited from it.

In addition to this simple inheritance with capturing of all parent’s attributes, it is possible to use more complex kind of inheritance - with redefinition of some attributes in child-type. Such inheritance is typical for initialization of objects, when *value* and *constraint* of specialized parent-type needs to be kept, but *auxiliary code* need to be changed a little.

Such mechanisms as plural (from several specialized types) and multilevel (from the descendant) inheritance also can be realized, however the need for them arises extremely seldom, and their existence can lead to the confused dependence between specialized types.

### B. Implicit Inheritance of Specialized Types

By extension of interface description type of parameter is replaced by specialized type. In other interface parameter of the same type can be replace be this specialized type. *Data type* attribute specifies data type that can be replaced by the specialized type. But *int*, *int* *, *const int &* types are different formally. Therefore for initialization of parameters of these types, for example, by one, it would be necessary to create specialized types for each parameter. It requires some efforts, but created types will have identical semantic: everyone initializes the parameter by one.

One of decisions of this problem is use of the mechanism of inheritance. For example, it is possible to create specialized

type for *int*, and create its child for *const int*. In this case creation of new type will take less time (since the basic attributes are inherited from the parent), but it also requires creation of new specialized type. Besides for *int* * type it is impossible to inherit value from specialized type for *int*, because of the first has the *value = "new int(1)"*, but the second has the *value = "1"*.

To avoid the described problem, it is necessary to have an opportunity to use one specialized type for parameters of types that can be cast to each other. Then for parameter of type *int* * it will be possible to specify the specialized type based on *int*. The type of the variable generated at initialization of parameter coincides with *data type* of its specialized type. At use of this variable as parameter of interface, it will be cast to the necessary type. For example, if to specify specialized type with *value="1"* and based on *int* for parameter of type *int* *, the following code will be generated:

```
…
int Par_1 = 1;
someFunction(&Par_1);
…
```

Types which can be cast to each other form a class of equivalence. The specialized type can be used instead of any type from a class of equivalence of its *data type*. It can be interpreted as a kind of inheritance: at discrepancy of type of parameter and *data type* of specialized type, there is a creation of the new specialized type based on type of parameter and inheriting this specialized type. Actually it is not necessary to create new specialized type - it is generated automatically, but because of similarity with process of inheritance the given mechanism is called as implicit inheritance.

### C. Once-only Specialized Types

At extension of the description of interface the developer should choose the necessary specialized type from the list of created for the parameter type. Often this list is too big, so it is difficult to find necessary type. But there can be types, each of which is used only by one interface, because of its strong connection with feature of interface. With a high probability they will never be used more. Such specialized types are called as *once-only*. This property is contained in *kind* attribute of these types.

As it is clear from its name, once-only specialized type can be used only once. Until it's used, it appears in specialized types list.

As an example of once-only specialized type consider the special type for returned value of a method *className()* for any class from Qt3 library [3], for example for *QPushButton*. This method returns the name of the class, that is *"QPushButton"*. The corresponding specialized type has the *constraint = "QString($0) == "QPushButton""*. *Data type=char**. This specialized type and similar should be once-only. Otherwise the list of specialized types for *char* type will contain big number of types checking class name.

### D. Default Specialized Types

If no specialized type bound with parameter, then generator have to initialize this parameter by oneself. Depending on a situation the generator can initialize parameter concrete value (for intrinsic types), the constructor or a call of other interface with suitable returned value. But sometimes such initialization without use of specialized types leads to undesirable result. In this case it is necessary to create (or to choose from already created) suitable specialized type. As a rule, for classes there is an allocated specialized type which creates object of this class and sets its some general properties. This type is used by developer as *default* specialized type. I.e. always, when it is not necessary to initialize object in any special way, this specialized type is used for it. It is convenient in that case to specify to the generator to use this specialized type when no type id specified for parameter by developer. For this purpose, the necessary specialized type should be set as a *default for parameters* specialized type. This property is contained in *kind* attribute of this specialized type.

Similar situation and for the specialized types checking returned values. Such type can be defined as *default for returned values* specialized type. When no specialized type is specified for returned value of any interface, but there is a default for returned values type, the generator will take this default type for check of correctness of the returned result. Usually such specialized types check the general properties, for example: *"$0.isValid() == true"*, *"$0.isEmpty == false"*, *"$0.isNull() == false"*, etc. Such checks compel to select parameters that satisfied for these conditions, or to specify the usual specialized types that checking reverse conditions (*"$0.isValid() == false"*, *"$0.isEmpty == true"*, *"$0.isNull() == true"*, etc).

In whole, *default* specialized types allow to lower efforts to specification of the description of interface, and to make test more right by correct ways of initialization of objects by default and an automatic insert of checks of returned values.

## V. Extension of LSB Database

The mechanism of specialized types described in given paper has been realized within the limits of LSB Infrastructure project in ISP of the Russian Academy of Science. LSB database has been taken as a basis. It contains the necessary signature description of interfaces and data types. It has been extended by the tables containing specialized types and some auxiliary information, necessary for automatic test generation. The scheme of this expansion is shown in figure 2.

Expansion of LSB database includes 8 tables. 4 tables from them store the information about specialized types: *TGSpecType*, *TGSpecTypeValue*, *TGSpecTypeConstraint*, *TGParameterProxy* (these tables are separeted by a bold dotted line in the scheme). *TGParameter* and *TGInterface* tables bind the tables of specialized types with the *Parameter* and *Interface* tables from LSB. *TGType* stores information about initial base types for every data types.

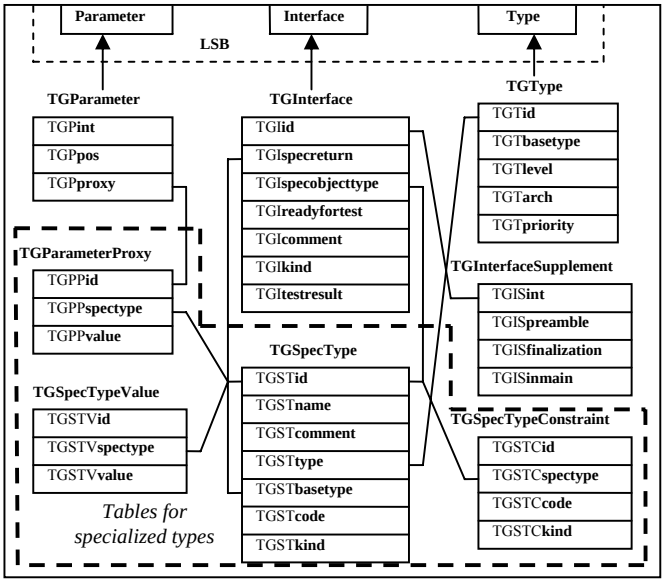


Fig. 2. Extension of LSB database.

*TGInterfaceSupplemen* stores code that create correct environment for calling corresponding interface.

## VI. Conclusion

The given paper discusses the problem of definition and representation of the information necessary in addition to signatures of interfaces for automatic generation of simple tests that call these interfaces with correct parameters and in a correct environment and check for blunders. This information is represented in *specialized types*, which specify semantic of parameters and returned value of tested interfaces, allowing to create initialization of parameters and environment automatically. Besides there are the various mechanisms that allow to reduce quantity of created specialized types within the limits of work with group of the connected interfaces, that essentially reduces laboriousness of the extension information for such groups.

Realization of storage of the extended information about specialized types is shown in this paper. This realization is based on LSB database and created within the limits of LSB Infrastructure project [4]. Based on this realization the extended information for almost 10000 interfaces from Qt3 library [3] has been set. Thus it has been created about 1600 specialized types, and productivity of creation of tests (with debugging) is 70 interfaces in day per person on average.

## References


[1] http://www.linuxbase.org.
[2] http://www.linux-foundation.org/navigator/commons/welcome.php.
[3] http://doc.trolltech.com/3.3/index.html.
[4] http://ispras.linux-foundation.org/.