FRIS Language Service for Extended Fortran Support in Microsoft Visual Studio

I.S. Ratkevich <ratkevichis@gmail.com>,
Russian Federal Nuclear Center – All-Russian Scientific Research Institute of Experimental Physics (RFNC – VNIIEF),
607190, Mira, 37, Sarov, Nizhny Novgorod Region, Russian Federation

Abstract. This report deals with the construction of the language service for extended support of the Fortran programming language in the integrated development environment (IDE) Microsoft Visual Studio. The model and general approach for language service construction is offered.
The proposed general model of a language service consists of five key blocks: the IDE integration block; the analysis block; the recognized elements storage block; the elements serialization/deserialization block; the elements view model block.
The IDE integration block connects a language service with a basic IDE infrastructure. It's responsible for subscription of Language Service for text editing events and for providing corresponding responses.
The Analysis block is responsible for accomplishing lexical, syntactic and semantic analysis. It gathers all needed information about the elements of a programming language and puts it into the recognized elements storage block. The second task of this block is to provide information for syntax highlighting of edited text.
The Recognized elements storage block is like a database of all elements needed for the Language Service operation. In general case, it is kind of a symbol table. The storage block could be filled from two sources: from analysis block, as a result of analysis of a source files, and from elements serialization/deserialization block, as a result of deserialization from a previously existing specialized program description in the case of using model of API (Application Programming Interface) for arbitrary programming libraries.
The elements serialization/deserialization block performs two functions. Firstly, it allows saving the content of programming projects as XML files for description of API and documentation comments. Secondary, it allows restoring the content of programming projects from their XML models.
The Elements view model block is a link, a kind of adaptor for elements of storage block to their representation needed by IDE integration block. Thus, recognized elements may contain some information that is not necessary to IntelliSense technology features, or on the contrary, does not contain some needed information. The elements view model is playing this interconnection role. It contains data types that are wrappers for elements of storage block, which fulfils requirements of the IDE integration block. There is also implemented various functions of filtering and selecting of different kinds necessary information.

The proof of operability of proposed general model of a language service is given on the example of the FRIS language service developed by author. The material could be equally applied for construction language services both for other programming languages and for other development environments.

Keywords: FRIS; Fortran Intelligent Solutions; Fortran; Visual Studio Extensibility; Language Service; Visual Studio

DOI: 10.15514/ISPRAS-2015-27(3)-1

For citation: Ratkevich I.S. FRIS Language Service for Extended Fortran Support in Microsoft Visual Studio. *Trudy ISP RAN/Proc. ISP RAS*, 2015, vol. 27, issue 3, pp. 9-28. DOI: 10.15514/ISPRAS-2015-27(3)-1.

1. Introduction

Fortran [1], [2] is one of the first high-level programming languages. It was created in the 50s of XX century and it was intended for development of programs for scientific calculations. Fortran is still used by its intended purpose in the development of simulation programs. Nowadays the most widespread Fortran standard is Fortran 2003 [2] (however there is the Fortran 2008 standard, and the Fortran 2015 standard is in development stage). It cardinally differs from previous standards because it introduces the support of object-oriented programming in a Fortran language. This feature changes the language syntax, where many new statements are added in conjunction with new conceptions. Definitely, such modernizations are necessary, but at the same time they are objectively making the language more complicated.

However these difficulties may be hidden or even eliminated, if the Fortran-programmer will have appropriate assistance from the IDE in which he writes his programs code. The most widely used IDE on Windows is Microsoft Visual Studio. It is extensible and allows adding practically any feature into it. As an example, Visual Studio may be extended to support various programming languages.

The most widely used Visual Studio integrations of the Fortran language are being developed in Intel [3] and PGI [4] in conjunction with corresponding compilers. **However the supported features of those integrations significantly inferior to integrations developed by Microsoft, e.g. for C# programming language. Primarily it applies to the support of InlelliSense [5] technology, which consists of the following features: List Members, Parameter Info, Quick Info и Complete Word (table 1).**

It must be noted that in all implemented IntelliSense features, excluding those for intrinsic procedures, there is essentially absent any description of the elements except for their definitions.

This great difference between Fortran support and support for languages, developed by Microsoft, became a key factor for author in the decision to implement the FRIS (**Fo**r**t**ran **I**ntelligent **S**olutions) language service, that is intended to cover this gap and implement all IntelliSense features to support Fortran-programmer in effective development of programs.

Table 1. The IntelliSense technology features implementation in Intel and PGI

Function	Intel	PGI
List Members	**No**	**No**
Parameter Info	**Yes, excluding overloaded procedures and type bound procedures**	**Yes, only for intrinsic procedures**
Quick Info	**Yes, excluding fields and procedures of derived types**	**Yes, only for intrinsic procedures**
Complete Word	**Yes, only for modules names, functions names and subroutines names**	**Yes, only for keywords statements**

2. Making model of a language service

Language service [6] is responsible for providing language-specific support for editing source code in the Visual Studio IDE, or, generally speaking, in any IDE. Basic language service must by definition [7] to provide a program syntax highlighting, all other features, including the IntelliSense support, are extra (or extended) features. The main question that must be answered at first when starting a new language service development is what features are needed for a programmer. After that, those features must be ranked by priority (or by usability).

Next, it is needed to identify the sources of data that must be used in the implementation of the language service. The main data source for any language service, no doubt, is source files containing programs text on a target language, but in some cases additional data sources may be needed.

The next stage is to estimate implementation complexity of needed functions. This estimation may include as the IDE restrictions to different components of a language service, and the analysis complexity of the target programming language itself.

After this the aggregate language service model is constructed, that reflects its major structural elements and interconnections between them. This report contains generalized and optimal, in author's opinion, language service model, which provides extended support for a target programming language.

When the aggregate language service model is constructed, each of its structural elements is detailed according to specific requirements to implementation of different features, and also depending on the restrictions of the target programming language.

Next in the report each of aforementioned steps in making language service will be examined in details, on example of the Fortran programming language, but the given material, without loss of generality, could be applied to any other programming language.

2.1 Analysis of requirements and the necessary features

The first thing, that definitely wants to see any programmer is a program syntax highlighting, for keywords, data type names, string literals, comments and so on. At the same time, it’s important to provide the ability to configure such highlighting, for example, for significant to user procedure names and data type names of program libraries, say, OpenMP, MPI, and others. Such syntax highlighting helps to focus attention on the most important details.

The second thing, that is important to a programmer, is the amount of provided context help, that at least must consist of the definition for a programming language element with which programmer works or wants to work (in the case of word completion lists). But in most cases the element definition is not enough to understand, how exactly the element must be used, as an example, a procedure that has more than a dozen parameters, some of which may be optional. In this case it’s necessary to accompany the element definition with some meaningful description. When the data that must be provided to user, and, respectively, that must be collected and stored, are identified, the sources, from which this could be obtained, must be analyzed.

2.2 Analysis of data sources

The most obvious way to get the definitions of programming language elements is the analysis of program source files. The form of such definitions is fixed in the programming language standard, e.g. in the Fortran standard. The meaningful description of the elements may be obtained, if to complement the program text with comments in a special form – documentation comments. The XML documentation comments are the standard for Visual Studio. So, the program text contains two languages: the base language – Fortran, and the embedded language – documentation comments language.

It should be noted, that Fortran has a distinctive feature in using of the programming libraries. There are three ways to connect the programming library to the main Fortran project:

- with source code files, that contains the library API, including procedure definitions, data types definitions and so on;
- with compiled binary files of Fortran modules, that have a closed format, which understandable just by compiler. Those files also contains the library API definitions;
- without any descriptions of library API. In such case the compiler will deduce the outer interfaces for used procedures, and will try to resolve external references by their names.

In the first case, it is possible to analyze source file that contains the library API and get all necessary information from it, but in the other two cases, it’s impossible to do so, and it's necessary to provide other mechanisms to get such information.

As a basis for implementation of this task, was taken the idea that is used in the program for automatic documentation generation for so called managed applications – Sandcastle [8]. It uses two files for generation of program documentation: one with the API description, and the other with the documentation for the API.

Fortran isn’t managed language, so it’s impossible to use the standard Sandcastle API format for description of its elements. Therefore the model for description

Fortran API was developed in FRIS for this purpose. It is the XML file in the special format, which contains a description of main Fortran elements. FRIS can save (serialization) the structure of elements, which is obtained from the analysis of program texts, into XML format and restore (deserialization) Fortran elements from their XML representation.

The XML model for Fortran documentation comments is also developed, including the features for its serialization and deserialization. This will allow to develop a special Sandcastle plug-in, and to use files of Fortran API and documentation comments description to automatically generate a developer or/and user help files.

2.3 Analysis of main operating characteristics of a language service

When developing a language service it's necessary to take into account that analysis of program texts will operate in a real time. This means that in most cases the text under analysis will be in the lexical, syntactic or semantic incorrect state, in terms of programming language specification. This peculiarity must be considered in the construction of corresponding analyzers.

The second peculiarity is in the fact that the analysis for a syntax highlighting is carried out in Visual Studio line-by-line (one line a time). The analyzer, colorizer in terms of VS, is transmitted for analysis a string of text and the analyzer state in which it was at the end of analysis of the previous line. This means that the corresponding analyzer must be constructed with the ability to save its state in any time and to restore its work from any such state. This approach makes it possible to carry out incremental analysis, which is very important for large source files (approx more than 10000 lines). Then, when some lines are changed, it's necessary to analyze just the changed lines, but not a whole file.

The third peculiarity that must be considered to create effective full-text analyzers is the need to take into account the state of source files. In terms of using program project source files in the IDE, file could be in a one of two essential states:

- opened in editor;
- doesn't opened in editor.

In the first case, it's needed to accomplish full-text analysis of a source files, but in the second one it's possible to accomplish a simplified analysis to collect information about just externally visible program elements. For example, it's not necessary to analyze whole body of procedure, because information, say, about its local variables could be needed to user just in a moment of editing a procedure body, which automatically transfers file with procedure to the sate "opened in editor", and consequently, the other analysis rules will be applied to it. Thus the requirement to analyzer to operate in two modes, for convenience "full" and "simplified" analysis, will significantly increase the analysis speed of programming project source files.

3. General model of a language service

The author proposes the following general model for building any language services, which is the result of summarizing author's experience in developing FRIS (Fig. 1).

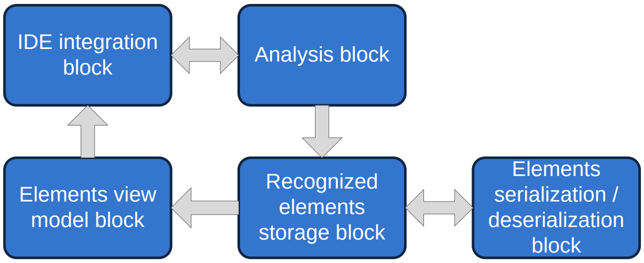

IDE integration block
Analysis block
Elements view model block
Recognized elements storage block
Elements serialization / deserialization block

Fig. 1. General language service model

As shown in Fig.1 any language service could be represented as 5 base blocks. The arrows represent the data exchange between blocks.

The IDE integration block contains interfaces implementation, which are required for interaction with IDE. It's responsible for subscription of a language service on the text editing editor events, and for corresponding responses, for example, for syntax highlighting and information providing for work of IntelliSense features.

The analysis block is responsible for lexical, syntactic and semantic analysis. When it receives events from the IDE integration block, it performs appropriate actions. For example, in response to file open event or text changed event, it will provide the information for syntax highlighting. It's also responsible for providing source files analysis depending on their states.

The recognized elements storage block is central data storage about all elements, necessary for language service. In general case, it is kind of a symbol table. The storage block could be filled from two sources: from analysis block, as a result of analysis of a source files, and from serialization/deserialization block, in the case of using model of Fortran API for any program libraries.

The elements serialization/deserialization block performs two functions. Firstly, it allows saving the content of programming projects as XML files for description of Fortran API and documentation comments. Secondary, it allows restoring the content of programming projects from their XML models. This approach reflects the dual nature of programming projects. Thus, for author of programming project, for example, program library, it is accessible in source files and it is perceived as "internal", but for a user of this library, it is perceived as "external", and its source files may be inaccessible to user.

The elements view model block is a link, a kind of adaptor for elements of storage block to their representation needed by IDE integration block. Thus, recognized elements may contain some information that is not necessary to IntelliSense technology features, or on the contrary, does not contain some needed information. The elements view model is playing this interconnection role. It contains data types that are wrappers for elements of storage block, which fulfils requirements of the IDE integration block. There is also implemented various functions of filtering and selecting of different kinds necessary information. It could be said, that the storage block is like a database, and the view model block is like a data selection procedures.

3.1 IDE integration block

The IDE integration block connects a language service with a basic IDE infrastructure. In the case of Visual Studio, the base language service must implement the IVsLanguageInfo [9] interface. This interface is responsible for providing information about target language including its name, associated file extensions, and component for a syntax highlighting (colorizer). Colorizer must to implement the IVsColorizer [10] interface, which is responsible for providing character-by-character information about colors of buffered program text representation in memory. In order to provide the IntelliSense technology support it is needed to implement 5 additional interfaces [11]: IVsCodeWindowManager, IVsMethodData, IVsCompletionSet, IVsTextViewFilter and IOleCommandTarget.

To simplify for developers the task of creating new language services, and the other tasks of Visual Studio extension, Microsoft created MPF (Managed Package Framework) [12] library, which supplies a set of base classes that implements many needed interfaces, and thus provides to developers the ability to implement only the features that is needed to them. Let’s take a brief look at the key classes that are necessary for the implementation of the language service and its various features.

The LanugageService abstract class provides basic implementation of a language service. It contains a number of abstract methods responsible for different features of a language service, such as syntax highlighting, and initialization of full-text source files analysis in order to provide information for various IntelliSense features, and so on.

The Source class is a source file abstraction in terms of a language service. It is used to store all information about edited file, as well as for interoperability with other language service model classes, which require information about current source file. In particular, it contains an instance of the Colorizer class, which is responsible for syntax highlighting.

The Colorizer class implements IVsColorizer interface. This class is used by the core editor of IDE for providing of syntax highlighting in current source file. For even more flexibility and abstraction MPF Colorizer from concrete programming language, the scanner abstraction is used.

The scanner must to implement IScanner interface. Each scanner is essentially a specialized lexical analyzer, which must to be able to save its current state and to restore its state for continuation of analysis as if it is doing a simple linear analysis of character stream.

The AuthoringScope class contains all information about a source file which is the result of parsing of this file. It is the central place for providing information for basic

IntelliSense technology features. In particular, method GetDataTipText – returns a string that contains description of programming language element, under the mouse cursor. It provides data for Quick Info IntelliSense feature. Method GetDeclarations – returns a list of programming language element definitions. It provides data for List Members and Complete Word IntelliSense features. Method GetMethods – returns a list of method signatures with a given name, including their overloaded versions. It provides data for Parameter Info IntelliSense feature.

In FRIS implementation is used modified version of MPF library, since a number of methods needed by FRIS were inaccessible for overriding in Microsoft’s MPF classes.

3.2 Analysis block

The FRIS analysis block consists of two sub blocks: analysis for syntax highlighting and full-text analysis (in “full” and “simplified” mode) for a collection of information about elements in a source file.

The FRIS analyzers are built with the ability to support sublanguages. In this case, the base language is Fortran, and sublanguages are any other languages, other than Fortran, that are used in the program text, for example, the XML documentation comments language and the OpenMP directives language.

Fig. 2 shows the general scheme of working of the analyzers stack, on the example of analysis of a part of XML documentation comment. The base language analyzer (Fortran) generates tokens, which are then passed through a tokens filter. If token matches with one of registered sublanguages, the appropriate analyzer is called. The output is a set of fully recognized tokens for all supported languages.

The peculiarity of work of a syntax highlighting analyzing block is that it is essentially some kind of extended version of a lexical analyzer, since there are strict requirements on the speed of operation of a syntax highlighting. Support for arbitrary program library in FRIS is, in particular, in the ability of a visual highlighting of their elements such as procedures, modules, data types, etc. Such highlighting is performed in a syntax highlighting block based on the current context. For any identifier under analysis the check depending on current scope is performed, whether it belongs to arbitrary library, which elements necessary to highlight. Then, if necessary, the identifier is highlighted with a defined earlier color.

The peculiarity of full-text analysis is in the used analysis strategy. Since the analysis is need to be performed in the real time, while the user modifies the text of program, all analyzers must to work in the error suppression mode. It must be noted that Fortran is very complicated language for analysis, because of its lexical and syntactical peculiarities. The most striking examples are:

- the ability to use multiline tokens, for example, identifiers. Next is given the sample of a multiline identifier “my_id”. The special attention must be given the fact that in between a start and end lines of any multiline lexeme, it is allowed to use comments and blank lines.

```
1    my_&
2    !comment
3
4    !another comment after blank line
5    &id
```

- the absence of reserved keywords. The decision whether identifier is a keyword depends on a context of its usage in a statement. Therefore, it is not statements that are identified by keywords, as in languages with reserved keywords, but the keywords are identified by statements. Taking into account that analysis is performed in a real time, it is impossible to determine the identity of incomplete statement. For example, it is unclear, whether "if" is a keyword that belongs to conditional statement, or it is a name of an array, in the following part of statement: "if(".

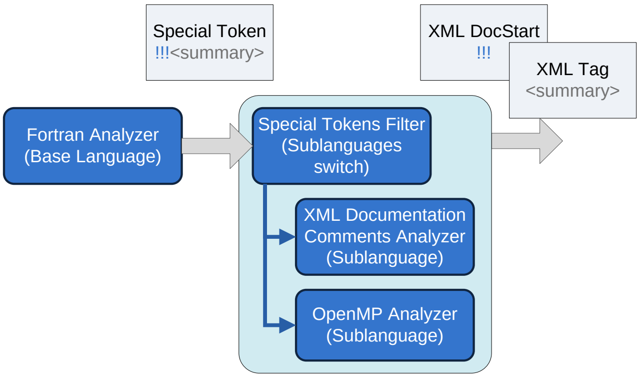

Special Token
!!!<summary>
XML DocStart
!!!
XML Tag
<summary>
Fortran Analyzer (Base Language)
Special Tokens Filter (Sublanguages switch)
XML Documentation Comments Analyzer (Sublanguage)
OpenMP Analyzer (Sublanguage)

Fig. 2. The general analyzers operation scheme

The emphasized peculiarities greatly complicate the development of analyzers for Fortran. But all of them are taken into account in FRIS. In particular, the optimistic parsing strategy is used. The parser processes a source file statement-by-statement. For every statement the abstract syntax tree (AST) is built. If the statement could not be matched, e.g. as a result of that the user just not has time to completely type it; the special AST is generated for it, which includes all mismatched tokens.

In conjunction with a parser the full AST builder is operating (Fig.3). It builds the full AST from the individual statement ASTs. It also stores the AST that is already built. The builder task is to track operations of opening and closing of syntactical contexts, in particular their optimistic completion.

For example, if now the operator "if(…)then" is analyzed, then according to standard, it could be completed only by "endif" statement. However, the user could not have enough time to fully type this statement, then the builder will interpret the "end" statement as a completion of a "if(…)then" operator. Similarly to it, if in the end of parsing of source file the stack of open contexts of the builder is not empty, then they are completing in a special mode – completion by the end of the file. It is also have ability of priority processing of high level element statements. For example, if the subroutine element is processed now, and as a result of a parsing the function element definition statement is discovered, then the current subroutine element is being completed with a special flag, and the function element processing is being started.

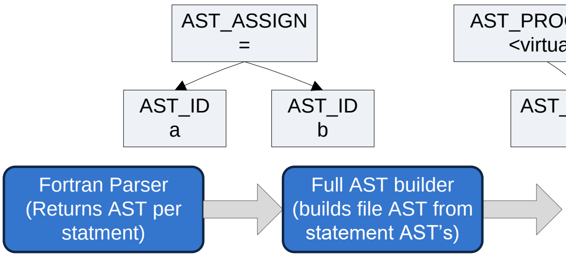

AST_ASSIGN
=
AST_ID
a
AST_ID
b
AST_PROGRAM
<virtual>
AST_ASSIGN
=
Fortran Parser (Returns AST per statment)
Full AST builder (builds file AST from statement AST's)

Fig.3. The FRIS parser operation scheme

Thus, the parser is always outputs the correct AST, which has no error nodes. This allows simplifying the semantic analysis algorithm. The semantic analyzer walks the AST and collects information about all needed Fortran elements, which then stores in the recognized elements storage block.

3.3 The recognized elements storage block

The recognized elements storage block is a central storage for all known in the current programming project elements (modules, data types, variables, etc.). It is filled from two sources: as a result of a source files parsing, and as a result of deserializing information about arbitrary libraries.

This block is essentially a kind of a symbol table. Its design must take into account that information in it will be continuously updating as a result of the user editing of source files.

Consider the proposed generic model of the storage block (Fig. 4).

It consists of following parts:

- the class for a symbol table description;
- the class for an interface description for a typical element of the programming language;
- the class for an interface description for a typical scope of the programming language;
- the classes describing specific elements of the programming language, that implement interfaces of a typical element and of a typical scope, for elements, which are scopes.

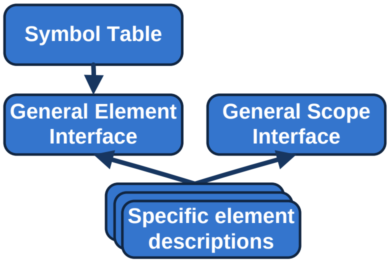


Fig. 4. The model of the recognized elements storage block

The class for a symbol table description must be built as indexed data storage, in order to effectively processing operations of update and elements search. For maximum flexibility it must store the references on the interface for a typical element, instead of references to specific elements. The specific element could be obtained from an abstract interface as a result of type casting. The following scheme of a symbol table is proposed (Table 2).

Table 2. The model of a symbol table

Field	Data type	Description
Names	map<long, string>	Map unique identifier to string
Elements	map<long, object>	Map element unique identifier to element object
Projects	map<string, map<string, list<long>>>	Map program project name to map of project file names to list of file elements unique identifiers
ProjectDependencies	map<string, list<string>>	Map program project to program projects it depends from

In this approach, firstly there is an access to all elements (Elements field). Secondly, for any project there is a list of its dependencies from other projects, which allows simplify a search procedure of needed elements, and to exclude from the search result the elements that is not visible in target project. Thirdly, every project contains a dictionary of its source files, and elements, which contained in every file that allows to effectively performing the update operations. The update operation is a result of a source file parsing operation, due to a text changes made by user. Thus, since all elements that are connected with file is known, so their deletion from other dictionaries and insertion a newly recognized elements, is a relatively simple task.

Next consider the proposed interface for a typical element of a programming language (table 3).

Every element must have at least a name, a scope, where it’s defined, a description, for example, that is obtained from documentation comments, and a location. An element location consists from a declaration location and a definition location. Each of which is in turn consists from a file name, and an element region in it.

Consider the proposed interface for a typical scope of the programming language (table 4). The scope, in a general case, is a container of elements.

Table 3. The model of interface for a typical element of a programming language

Field	Data type	Description
Name	string	Name of element
Scope	Scope	Outer scope of element
Description	string	Description of element. For instance from documentation comments
Location	Location	Element location: definition location, declaration location. Location consists of file name and region. Region consists of 4 integer indexes: start line, start line character index, end line, end line character index.

Table4. The model of a typical scope of the programming language

Field	Data type	Description
Scope	Scope	Outer scope of this scope
Elements	list<Element>	List of elements of the scope

Every scope contains a reference to a parent scope and a list of elements that make up this scope.

Every specific element of a programming language must be derived from an interface for a typical element, and if it is a scope, from an interface of a typical scope.

3.4 The elements serialization/deserialization block

The elements serialization/deserialization block is a key element for the implementation of a mechanism to support arbitrary user libraries. The serialization mechanism performs a saving of a given programming project in a form of two special XML files: description of Fortran API and description of documentation comments. The optional level of refinement could be additionally specified. In the case, when the serialization is performed for creation a developer documentation of a programming project, then all elements are saved, but in the case of creation a user documentation or interface for a programming project as an external library, then just externally visible elements are saved. It should be recalled that for each element in the Fortran module, could be specified the access mode: public or private. The public elements are externally accessible when the module is used, but the private elements could be used just inside the module and inaccessible outside of it.

The deserialization mechanism operation is slightly different, because in deserialization there is just one operation mode – reading all information describing an arbitrary library. In this case, even if there will be provided XML files, that contains full description of arbitrary library, only externally visible elements will be read. This allows reducing the amount of memory needed to store a library description, and also eliminates the need to store elements, which will not be accessed to user under no circumstances, for example, private module elements, or internal elements of procedures.

For serialization and deserialization are used the models for description of Fortran API and XML documentation comments, that is developed by author and are expressed in the form of appropriate XML Schema Definitions (XSD) [13], [14]. Let’s consider each of these models.

The model of Fortran API (Fig.5) allows describing external interfaces of any library as a Fortran interfaces. The meaning and purpose some of the model elements are given in table 5.

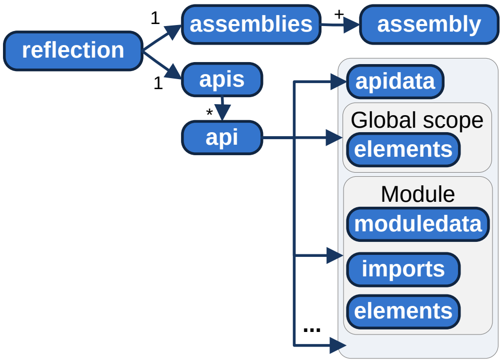


Fig. 5. The part of Fortran API XSD

Table 5. The description of some elements of the Fortran API model

Element (tag)	Description
reflection	Root tag
assemblies	Describes set of projects that API contained in this file
assembly	Describes individual project
apis	Root for all API description
api	Element description
apidata	Describes group and subgroup of element. I.e. for function: group – method, subgroup - function
moduledata	Module description switch
referencedata	Reference element switch
typedata	Derived type description switch
variabledata	Variable description switch
proceduredata	Procedure description switch
interfacedata	Interface description switch
methoddata	Method description switch
namelistdata	Name list description switch
commonblockdata	Common block description switch
imports	Module imports description
elements	List of inner elements

As can be seen from the above figure, tag “apis” contains a description of all project elements. The tag “api” is used for a direct element description. In order to uniquely identify the type of element: a module, a function, a subroutine, a data type and so on, the special switches, like a “moduledata” tag, are used.

One more remark should be made regarding the tag “elements”, which is used to describe the internal elements of current element. It’s allowed to specify here references – fully qualified element names, and their description place next in a main “apis” tag, and also it’s allowed to provide the description of child elements directly in this tag.

It should be noted that description of Fortran API may be used for a creation of Fortran procedure interfaces for their calls from other programming languages, that is solves the inverse problem.

Consider the model of documentation comments. It conceptually consists of two interconnected parts: a description of documentation tags for documenting program elements (Fig.6), and a description of documentation comments XML file format (Fig. 7). The meaning and purpose of the model elements are given in table 6.

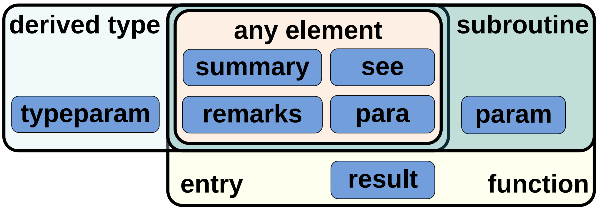


Fig. 6. The usage of documentation tags for different Fortran elements

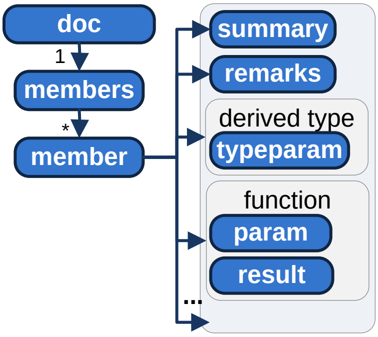


Fig.7. The part of Fortran Documentation XSD

For description of any element may be used 4 tags, two of which are high-level: “summary” and “remarks”, and other two are nested, it means that they could be used just inside of other tags: “see” and “para”. In addition to them, for description of:

- derived type parameters is used “typeparam” tag;
- arguments of subroutines, functions and entry points is used “param” tag;
- result of function is used “result” tag.

Table 6. The elements description of the Fortran documentation model

Element (tag)	Description
doc	Root element
members	Container for all documentation elements
member	Contains documentation for single element
summary	Element summary
remarks	Additional information for element
see	Internal tag, makes reference to given element
para	Internal tag, creates paragraph in parent tag
typeparam	Describes derived type parameter
param	Describes argument of subroutine or function
result	Describes function result

Thus, files for description of the model of Fortran API and documentation comments form the basis not only for work with arbitrary libraries in Fortran, but also form the basis for the generation of the reference documentation, for example with a Sandcastle tool. It should be noted that Fortran API model can be used for solving the inverse problem – description of API for a Fortran procedures for their using from other programming languages.

3.5 The elements view model block

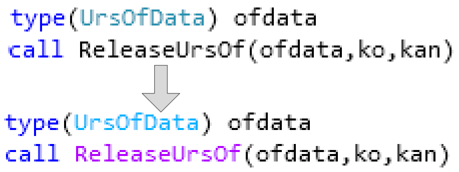
The elements view model block is a link between the IDE integration block and the data storage block. It performs two basic functions: converts a data from a storage block to a form required by the IDE, and performs various search operations in a storage block.

The convert operation of stored data to the form required by the IDE produces elements that are complemented by the properties of visual representation. For example, such properties as text color and element icon, which used in various completion lists, are set. In other words, the elements view model block contains various aspects of data presentation to user. Thus the structure of the view model block is analogue to the structure of the storage block. It also defines interfaces for typical presentation elements and scopes, and a set of their specific implementations for each element of the storage block.

The second function of this block is the search function. Here are performed various operations of elements resolution in a scope, a search for elements with the specified name and type, etc. That is, it performs the selection of needed elements from the storage block that taking into account a different aspects of a programming language. Then, selected data converted to the form required for user representation.

4. Proof of concept

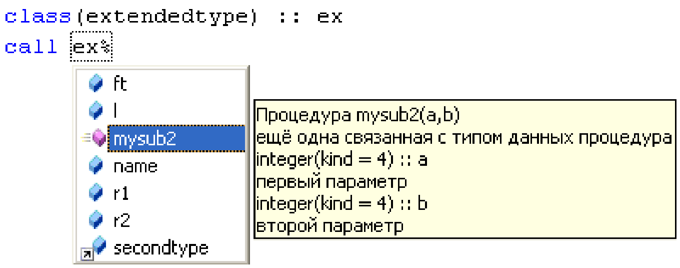
The FRIS language service is built on the basis of the general model of a language service, and implements all described blocks. Figures 8-13 are examples of work of its various functions, proving the presented conception of a generalized language service model, including providing extended support for user libraries.



```
type(UrsOfData) ofdata
call ReleaseUrsOf(ofdata,ko,kan)

type(UrsOfData) ofdata
call ReleaseUrsOf(ofdata,ko,kan)
```

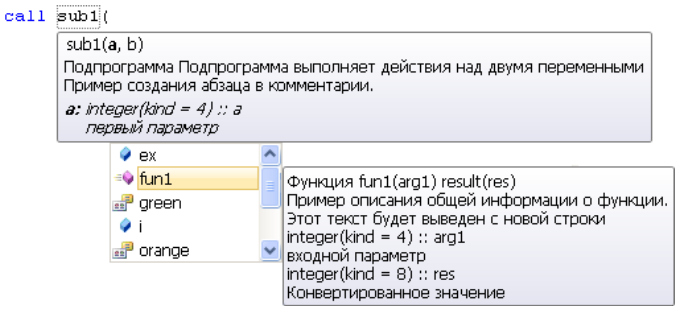
Fig. 8. The extended support of user libraries (before and after)



```
class(extendedtype) :: ex
call ex%
```

Процедура mysub2(a,b)
ещё одна связанная с типом данных процедура
integer(kind = 4) :: a
первый параметр
integer(kind = 4) :: b
второй параметр

Fig. 9. List Members

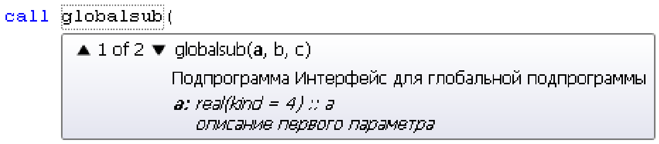


```
call sub1(
```

sub1(**a**, b)
Подпрограмма Подпрограмма выполняет действия над двумя переменными
Пример создания абзаца в комментарии.
a: *integer(kind = 4) :: a*
первый параметр

Функция fun1(arg1) result(res)
Пример описания общей информации о функции.
Этот текст будет выведен с новой строки
integer(kind = 4) :: arg1
входной параметр
integer(kind = 8) :: res
Конвертированное значение

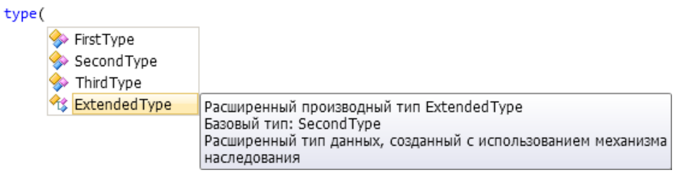
Fig. 10. Parameter Info and Complete Word



```
call globalsub(
```

▲ 1 of 2 ▼ globalsub(**a**, b, c)
Подпрограмма Интерфейс для глобальной подпрограммы
a: *real(kind = 4) :: a*
описание первого параметра

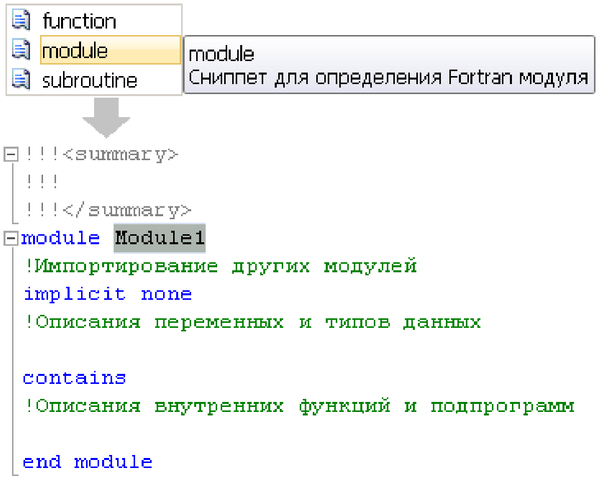
Fig. 11. Parameter Info for overloaded subroutine



```
type(
```

Расширенный производный тип ExtendedType
Базовый тип: SecondType
Расширенный тип данных, созданный с использованием механизма наследования

Fig. 12. Complete word for a derived type name



```
!!!<summary>
!!!
!!!</summary>
module Module1
!Импортирование других модулей
implicit none
!Описания переменных и типов данных

contains
!Описания внутренних функций и подпрограмм

end module
```

Fig. 13. Code Snippet Sample

Consider the pivot table of the language services from Intel, PGI and FRIS (table 7).

Table 7. The Intel, PGI and FRIS language services comparison

Function	Intel	PGI	FRIS
List Members	No	No	Yes
Parameter Info	Yes, excluding overloaded procedures and type bound procedures	Yes, only for intrinsic procedures	Yes
Quick Info	Yes, excluding fields and procedures of derived types	Yes, only for intrinsic procedures	Yes
Complete Word	Yes, only for modules names, functions names and subroutines names	Yes, only for keywords statements	Yes
Code Snippet [15] Support	Yes, but only as menu command or shortcut	No	Yes. Snippets included in Completion Lists
Documentation comments support	No	No	Yes. Documentation included in all tooltips
Support of user libraries	No	No	Yes

Thus, due to use of the developed general language service model, FRIS provides extended support of a Fortran in Microsoft Visual Studio.

5. Conclusion

The report presents the general model of a language service for extended support of a Fortran programming language developed by author. This model can be easily applied not only to create new language services for other languages, but also to create a language services in other IDEs.

All aspects that must be taken into account in development of a language service are given in details, including the analysis of user requirements, the analysis of a data sources for a language service, and the analysis of operation peculiarities of a language service in a specific IDE.

As a result of executing described analysis kinds, in every particular case, the plan of a language service development must be created. For a language service development simplification, the general model of a language service is given and each its block is described in details on example of its implementation in FRIS.

At last, the proof of proposed concept of constructing language services is given, on example of comparison FRIS with existing language services from Intel and PGI. The model that is used in FRIS provides its significant advantage over other language services.

It especially should be noted that FRIS implements a model for supporting user libraries. It includes a model of Fortran API and a model of documentation comments, developed by author. The Fortran API model allows not only to describe the interfaces of any library in terms of Fortran, but also allows solving the inverse problem, by known Fortran interfaces obtain API for target language. The documentation comments model allows user to document different Fortran elements straight in the program text, and then obtain documentation in various types of context help. The model of Fortran API in conjunction with the model of documentation comments can be used to create a developer and/or user documentation, for example with a Sandcastle tool.

References

[1]. The Fortran automatic coding system for the IBM 704 EDPM. Programmers reference manual. IBM, 1956

[2]. ISO. ISO/IEC 1539-1:2004 Information technology - Programming languages - Fortran - Part 1: Base Language, pp. 569

[3]. Intel Fortran Composer (Visual Fortran) URL: http://software.intel.com/en-us/articles/intel-fortran-composer-xe-2013-sp1-release-notes

[4]. PGI Visual Fortran URL: https://www.pgroup.com/products/pvf.htm

[5]. Using IntelliSense URL: http://msdn.microsoft.com/en-us/library/hcw1s69b(v=vs.80).aspx

[6]. Language Services URL: http://msdn.microsoft.com/en-us/library/bb165099.aspx

[7]. Model of a Language Service URL: http://msdn.microsoft.com/en-us/library/bb166518(v=vs.100).aspx

[8]. Eric Woodruff's Sandcastle Help File Builder Documentation URL: http://ewsoftware.github.io/SHFB/html/bd1ddb51-1c4f-434f-bb1a-ce2135d3a909.htm

[9]. IVsLanguageInfo Interface URL: https://msdn.microsoft.com/en-us/library/microsoft.visualstudio.textmanager.interop.ivslanguageinfo(v=vs.80).aspx

[10]. IVsColorizer Interface URL: https://msdn.microsoft.com/en-us/library/microsoft.visualstudio.textmanager.interop.ivscolorizer(v=vs.80).aspx
[11]. Language Service Interfaces URL: http://msdn.microsoft.com/en-us/library/bb164598(v=vs.80).aspx
[12]. Managed Package Framework Classes URL: http://msdn.microsoft.com/en-us/library/bb164709(v=vs.80).aspx
[13]. W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures URL: http://www.w3.org/TR/xmlschema11-1/
[14]. W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes URL: http://www.w3.org/TR/xmlschema11-2/
[15]. Creating and Using IntelliSense Code Snippets URL: https://msdn.microsoft.com/en-us/library/ms165392(v=vs.80).aspx

Языковой сервис FRIS для расширенной поддержки Fortran в Microsoft Visual Studio

И.С. Раткевич <ratkevichis@gmail.com>,
Российский Федеральный Ядерный Центр – Всероссийский Научно Исследовательский Институт Экспериментальной Физики,
607190, Россия, Нижегородская обл., г. Саров, пр-т Мира, 37

Аннотация. В данной статье рассматриваются вопросы построения языкового сервиса для расширенной поддержки языка программирования Fortran в интегрированной среде разработки Microsoft Visual Studio. Предлагается модель и общий подход к построению языковых сервисов.

Предлагаемая общая модель языкового сервиса состоит из пяти блоков: блока интеграции со средой разработки; блока анализа; блока хранения распознанных элементов; блока сериализации/десериализации элементов; блока модели представления элементов.

Блок интеграции с IDE соединяет языковой сервис с базовой инфраструктурой IDE. Он отвечает за подписку языкового сервиса на события редактирования текста пользователем в редакторе и за соответствующие отклики.

Блок анализа отвечает за проведение лексического, синтаксического и семантического анализа. Он собирает всю необходимую информацию об элементах языка программирования и помещает их в блок хранения распознанных элементов. Второй задачей данного блока является предоставление информации для подсветки синтаксиса редактируемого текста программы.

Блок хранения распознанных элементов является своеобразной базой данных всех элементов, необходимых для работы языкового сервиса. В общем случае он является разновидностью таблицы символов. Наполнение блока хранения может вестись из двух источников: из блока анализа, как результат разбора файлов с текстами программ, и из блока сериализации/десериализации элементов, как результат десериализации из ранее существующего специализированного описания программы, в случае использования модели API (Application Programming Interface) для произвольных библиотек.

Блок сериализации/десериализации элементов выполняет две функции. Во-первых, он позволяет сохранять содержимое программных проектов в виде XML файлов описания API и комментариев документирования к ним. Во-вторых, он позволяет восстанавливать содержимое программных проектов из их XML моделей.

Блок модели представления элементов является связующим звеном, своеобразным адаптером, элементов блока хранения, к тому виду, который необходим для использования в блоке интеграции с IDE. Так распознанные элементы могут содержать некоторую информацию, которая не требуется функциям технологии IntelliSense, или наоборот, не содержать нужной информации. В модели представления элементов организуются типы данных – обёртки для элементов блока хранения, соответствующие требованиям блока интеграции с IDE. Также здесь реализуются всевозможные функции выборки и поиска необходимой информации.

Доказательство работоспособности предложенной обобщённой модели приводится на примере разработанного автором языкового сервиса FRIS. Изложенный материал может быть в равной мере использован для построения языковых сервисов, как для других языков программирования, так и для других средств разработки.

Ключевые слова: FRIS; Fortran Intelligent Solutions; Fortran; Visual Studio Extensibility; Language Service; Visual Studio

DOI: 10.15514/ISPRAS-2015-27(3)-1

Для цитирования: Раткевич И.С. Языковой сервис FRIS для расширенной поддержки Fortran в Microsoft Visual Studio. Труды ИСП РАН, том 27, вып. 3, 2015 г., стр. 9-28 (на английском языке). DOI: 10.15514/ISPRAS-2015-27(3)-1.

Список литературы

[1]. The Fortran automatic coding system for the IBM 704 EDPM. Programmers reference manual. IBM, 1956

[2]. ISO. ISO/IEC 1539-1:2004 Information technology - Programming languages - Fortran - Part 1: Base Language, pp. 569

[3]. Intel Fortran Composer (Visual Fortran) URL: http://software.intel.com/en-us/articles/intel-fortran-composer-xe-2013-sp1-release-notes

[4]. PGI Visual Fortran URL: https://www.pgroup.com/products/pvf.htm

[5]. Using IntelliSense URL: http://msdn.microsoft.com/en-us/library/hcw1s69b(v=vs.80).aspx

[6]. Language Services URL: http://msdn.microsoft.com/en-us/library/bb165099.aspx

[7]. Model of a Language Service URL: http://msdn.microsoft.com/en-us/library/bb166518(v=vs.100).aspx

[8]. Eric Woodruff's Sandcastle Help File Builder Documentation URL: http://ewsoftware.github.io/SHFB/html/bd1ddb51-1c4f-434f-bb1a-ce2135d3a909.htm

[9]. IVsLanguageInfo Interface URL: https://msdn.microsoft.com/en-us/library/microsoft.visualstudio.textmanager.interop.ivslanguageinfo(v=vs.80).aspx

[10]. IVsColorizer Interface URL: https://msdn.microsoft.com/en-us/library/microsoft.visualstudio.textmanager.interop.ivscolorizer(v=vs.80).aspx

[11]. Language Service Interfaces URL: http://msdn.microsoft.com/en-us/library/bb164598(v=vs.80).aspx

[12]. Managed Package Framework Classes URL: http://msdn.microsoft.com/en-us/library/bb164709(v=vs.80).aspx

[13]. W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures URL: http://www.w3.org/TR/xmlschema11-1/

[14]. W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes URL: http://www.w3.org/TR/xmlschema11-2/

[15]. Creating and Using IntelliSense Code Snippets URL: https://msdn.microsoft.com/en-us/library/ms165392(v=vs.80).aspx