Using Visual Studio Extensibility Mechanisms for Requirements Specification

João Leonardo Carmo [1], Carlos Videira [2], Alberto Rodrigues da Silva [3]

[1] INESC-ID, Instituto Superior Técnico,
Rua Alves Redol, nº 9 –1000-029 Lisboa, Portugal
joao.carmo@tagus.ist.utl.pt
[2] INESC-ID, Universidade Autónoma de Lisboa,
Rua de Santa Marta, nº 56, 1169-023 Lisboa, Portugal
cvideira@acm.org
[3] INESC-ID, Instituto Superior Técnico,
Rua Alves Redol, nº 9 –1000-029 Lisboa, Portugal
alberto.silva@acm.org

Abstract. Although software requirements engineering is an essential activity for the successful development of information systems, most of the existing software development environments do not provide support for requirements specification. They concentrate on the development activities, such as code edition, solution management, components testing and deployment, change and configuration management, or integration with modeling artifacts. The Visual Studio .NET IDE is one of the most popular development workbenches, profiting from years of Microsoft experience in developing tools to enhance and automate the tasks of the developer, but still has no support for requirements specification. In this paper we describe how Visual Studio .NET extensibility mechanisms have been used to support the development of a new requirements specification initiative, which we have called ProjectIT-Requirements.

Keywords. Requirements; Requirements Specification Languages; VS .NET Extensibility Mechanisms, ProjectIT-Requirements.

1 Introduction

The development of information systems is a complex process usually initiated with the identification and specification of the requirements of the system to be developed, and in particular of its software components. Requirements describe what the system should do, which is obviously critical for the success of the whole development process. A classical definition from Kotonya says that a "*requirement is a statement about*

a system service or constraint" [2]. Taking into account the importance of requirements for the success of the development process, we should expect a widespread and systematic use of tools and techniques to support the activities that deal with requirements. However, until now the most successful tools we have built to support the development process concentrate on the development tasks and in the modeling activities. Although the list of requirements tools is large (more information available in [http://www.paper-review.com/tools/rms/read.php]), they are used only by a limited number of people, and frequently not the most effective way.

As a result of the experience gathered from previous research and practical projects, the Information Systems Group of INESC-ID [http://gsi.inesc-id.pt/], in Lisbon, Portugal, started an initiative in the area of requirements engineering, called Projec-tIT-Requirements [8]. One of the results of this project is a new requirements specification language, called ProjectIT-RSL [7], which applies the most common linguistic patterns used for requirements specification. Since one of our goals is to provide a common workbench throughout the entire software development process, integrating the requirements with the development artifacts, we decided to use the Visual Studio .NET (VS .NET for short) development environment to apply and prove our ideas and research. Some of the reasons that motivated our choice include:

1. VS .NET is one of the most successful and widely adopted development environments.
2. Microsoft is recognized for the experience and continuous effort in the development of tools to ease the developer's tasks. However, currently Microsoft does not have any specific tool dedicated to support the requirements specification, fully integrated with his development environment (which is for us an obvious and interesting opportunity).
3. Although VS .NET does not support requirements specification, it includes the possibility of adding extra functionalities to its IDE, such as the support for non-native languages, implementing a common basic behaviour (like editing, compiling, and dynamically detecting syntactic and semantic errors). From our point of view, a requirements specification language, based on natural language patterns, should be handled like any other "programming" language.

This paper describes how we have extended the VS .NET environment to introduce features to support requirements specification, and it is organized in the following sections: section 2 gives an overview of the ProjectIT research program, and, in particular, its requirements specification language, ProjectIT-RSL; section 3 presents an overview of the VS .NET environment, and in particular of its extensibility mechanisms; section 4 describes how we have used these mechanisms to support our requirements specification language; finally, section 5 justifies our perception that this proposal has some innovative contributions and describes the work to be performed in the future.

2 An overview of ProjectIT-Requirements and ProjectIT-RSL

The Information Systems Group of INESC-ID has been developing efforts to increase the productivity of the software development process. Among others, it is worth mentioning XIS [5], whose main contribution was the development of the XIS/UML profile intended to specify, simulate and develop information systems following a MDD approach, and ProjectPro [3], a Web based collaborative system to support the definition of basic concepts (projects, releases, sub-systems, requirements) and their relationships. Recently we started a research program, called "ProjectIT", whose goal is to develop a complete software development workbench with support for requirements engineering, analysis and design, generative programming techniques, and project management activities [6].

Within the context of Project-IT, we started the ProjectIT-Requirements project [9], to combine the benefits of rigorously and precisely defining requirements with the need to use a simple notation understandable by non-technical stakeholders, which leads to the idea of using a controlled natural language. A controlled natural language is a subset of the words used in our common natural language, where the selected terms have their usual meaning, but can also be interpreted in specialized contexts (normally using tools).

After having defined the goals of the project [9], we identified its main components: the **Requirements Specification Language**, which we have called **ProjectIT-RSL**, and a basic set of tools needed to reach the goals of the project, which are the **Requirements Editor** and the **Requirements Compiler**, as described in [7]. Our vision was to build a tool for writing "requirements documents", like a word processor, and as we write, it will warn us of errors violating the requirements language and grammar rules we have defined.

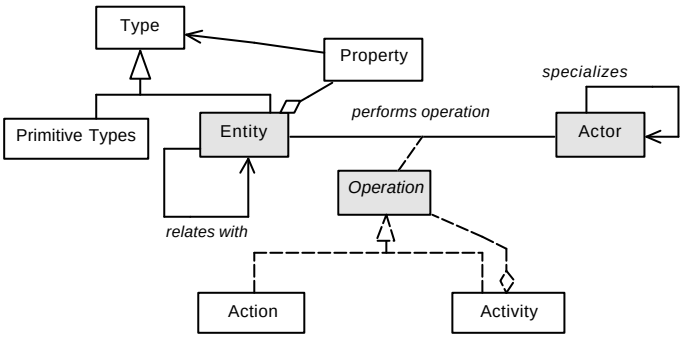

Figure 1 – ProjectIT-RSL metamodel – basic concepts

ProjectIT-RSL is currently a simple language, intended to support the requirements specification of interactive systems, and is represented by a metamodel (an overview is represented in Figure 1) and a grammar that defines the rules to map the metamodel's concepts into sentences, which are validated by the requirements tools. Requirements are usually expressed by natural language sentences that have a subject, a

verb and other words to complement their meaning. We mapped this natural language into a conceptual language saying that *actors* carry out *operations*, which can access one or more *entities* or entity's properties. For example, we identified the following common patterns:

```
<Simple Requirement>: <Actor> <Operation> <Entity>
<Simple Requirement>: <Actor> <Operation> <Entity Property>
<Conditional Requirement>: <Condition> < Simple Require-
ment>
<Requirement>: <Simple Requirement> | <Conditional Require-
ment>
```

These simple patterns enable writing requirements sentences such as:

```
The system manages entities.
The user creates invoices.
The user enters the invoice date.
The system automatically generates the invoice number.
If the invoice total is greater than 1000€, then the in-
voice discount will be 10%.
```

3 Visual Studio .NET Extensibility Mechanisms

Microsoft has been providing tools to support the work of the developer for some decades. In the beginning of the nineties Microsoft launched the first version of Visual Basic, one of the first development environments to provide a graphical interface for developing applications for the Windows platform. Later, Microsoft added a similar support for different languages besides Basic (C++ and Java), but using independent IDE's. At the same time, Microsoft added other features to ease the developer's tasks, such as wizards to create different kinds of projects or the possibility of creating add-ins to automate or enhance some activities. In 1998, Microsoft launched Visual Studio, integrating in one single tool previously separated languages, although the development environments remained independent.

The release of VS .NET was an important step in Microsoft's development tools evolution [1]. Besides integrating in a single environment many development tools (for example, code and designer editor, debugger, object browser, data editor, to name just a few), VS .NET IDE provides a shared development environment for multiple natively supported languages that previously needed independent environments, such as Visual Basic, Visual C++, and the new Visual C#.

One of the most remarkable features provided by VS .NET is the possibility of extending the IDE to add extra functionalities. Microsoft included in VS .NET different extensibility options, including macros, add-ins and wizards. The extensibility API available in VS .NET allows the developer to create tool windows (such as the project explorer and the toolbox), that can be used with languages already installed in the IDE. The existing editors and designers (grouped in the category of document windows)

can be accessed using this API, but the extensibility model doesn't allow the creation of new document window types. For applications and external products that need these more advanced features and to allow their integration into the IDE, Microsoft provided the Visual Studio Industry Partner Program SDK (VSIP for short, with more information available in http://www.vsipdev.com).

3.1 Visual Studio Automation Object Model

The basic extensibility mechanisms are based upon a common Automation Object Model, as shown in Figure 2, which was first developed for Visual C++ 5.0. The Automation Object Model is a COM object model that provides a higher level abstraction access to most features of VS .NET, and includes more than 140 different objects, giving an experienced developer a high level of control over the IDE.

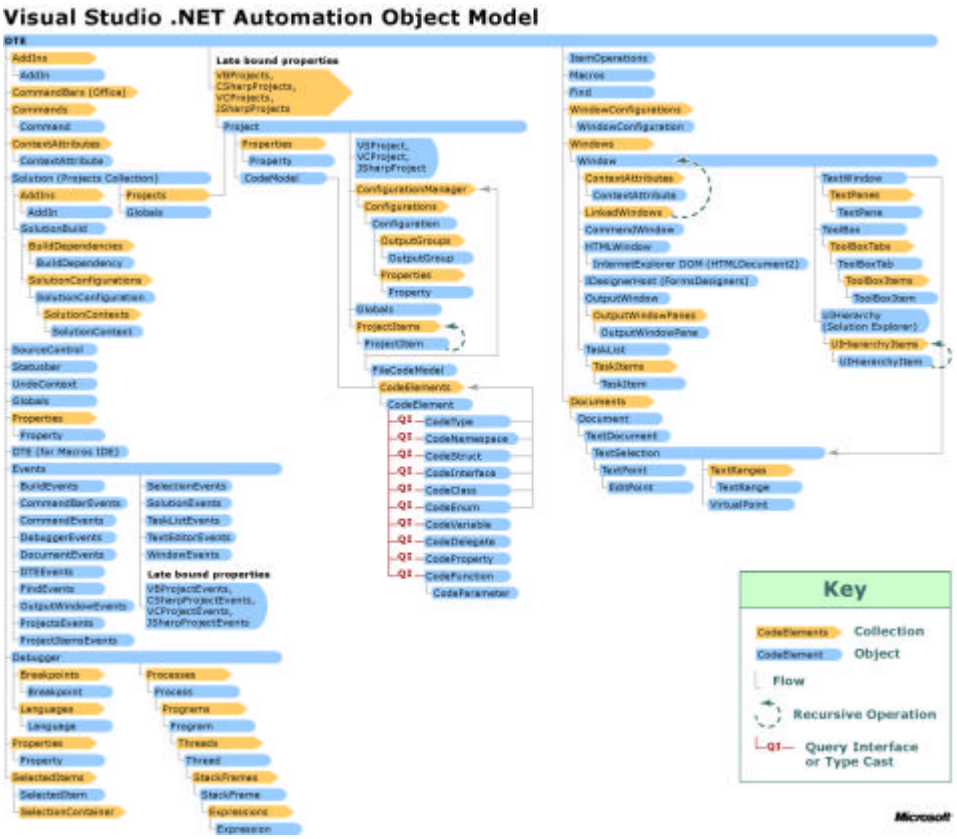

Figure 2 – The Automation Object Model (from http://msdn.microsoft.com/library/en-us/vsintro7/html/vxgrfAutomationObjectModelChart.asp)

Macros, add-ins, and packages (the unit of deployment in VSIP) can use this model to interact with the IDE and with its components, such as toolbars, menus, the project system, the form designers, the code editor, and the various tool windows.

These features enhance the developer's productivity in different ways, including:

1. Finding additional information for a project opened within the IDE, using the Project Object Model (although doing it frequently, many people are not aware they are using an automation object model feature).
2. Finding, creating and inspecting code such as namespaces, classes, methods, properties, and parameters using the Code Model.
3. Creating additional windows to implement specific features (or to enhance existing ones), using Tool Windows.
4. Adding additional objects to the toolbox, and using them in a designer window.
5. Developing Wizards to automate some tasks, including code generation.

A good description of the basic extensibility mechanisms is provided in [4].

3.2 VSIP (Visual Studio Industry Partner Program)

Macros and add-ins offer the developer a wide range of possibilities to extend VS .NET using the Automation Object Model, but they are not enough to support more advanced features. As already said, they cannot be used to create new document windows, which are required by programming languages. To support these features and to further extend VS .NET, Microsoft provided VSIP, which allows the highest level of integration of specifically developed features with VS .NET. VSIP supports building custom editors and designers to integrate new .NET languages and other productivity tools in the IDE. Some examples of companies that have successfully integrated tools in VS .NET IDE include ActiveState (developed Visual Perl .NET and Visual Python .NET), Fujitsu (developed NetCobol .NET) and Compuware (with its DevPartner suite of tools).

The current version of VSIP is mostly built using C and C++ code, and it is not managed. This is about to change with the next release of VSIP SDK, renamed to Visual Studio 2005 SDK. The architecture of the current version of VS .NET is still based primarily on COM interfaces, and Visual Studio 2005 SDK provides interoperability assemblies that enable a Package to access these interfaces using managed code in managed languages, and a wizard to accelerate the initial project creation.

4 Extending the VS .NET IDE for Requirements Specification

From the above discussion, and taking into account that we wanted to have a development workbench based upon VS .NET that supported the specification of require-

ments using ProjectIT-RSL, it is quite obvious that we needed the support of the VSIP libraries. Although ProjectIT-RSL is not a programming language, this is completely transparent from the point of view of VS .NET and VSIP features. The addition of a new language to VS .NET IDE requires basically the definition of three things: (1) a **language editor**; (2) a **new type of VS .NET project** to aggregate the new language's files; and (3) a **language compiler**. There is also the option of including a **language debugger**, but the current ProjectIT-RSL version does not implement it. In the following sections we discuss the issues related with these three components.

4.1 Developing ProjectIT-RSL's Visual Studio .NET IDE Editor

To develop the ProjectIT-RSL editor, we used the VSIP libraries to extend the VS .NET natively supplied editor. VSIP's libraries provide a complete and integrated set of functionalities, but VSIP's documentation doesn't have many intuitive examples to explain all of VSIP's extension capabilities. Simultaneously, the size of the code and the complexity of VSIP's features sometimes make the process of invoking the VSIP interfaces a complex task. Those are the reasons why we have chosen to use the **Babel** library, included with VSIP, which provides a higher abstraction layer above the lower-level VSIP libraries, making the task of creating a new language editor a simpler one, just by using well-known Lex and Yacc grammar files.

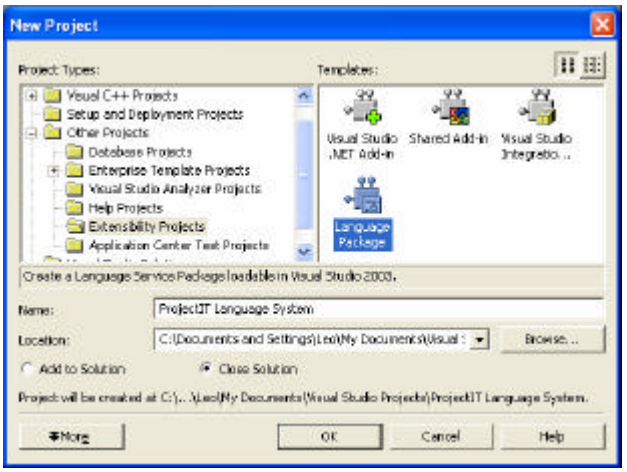

Figure 3 – ProjectIT-RSL VSIP Language Package project creation

To benefit from VSIP and Babel, we created a special type of project called **Language Package** (or **Babel Language Service**), under the group of extensibility projects (as shown in Figure 3). After choosing this type of project, a VSIP wizard creates a solution with pre-defined components (as shown in Figure 4).

The common project includes the Babel library interfaces that are used by the two files we had to provide in our ProjectIT Language System project, called lexer.lexand

parser.y. They are two well known file types for generating LALR(1)[1] language parsers, a Lex and a Yacc file. We used Win32 *open-source* implementations of these popular Unix tools, called **Flex** ([http://www.gnu.org/software/flex/]) and **Bison** ([http://www.gnu.org/software/bison/]), to produce the lexical and grammar parsers (files lexer.cpp and parser.cpp shown in Figure 4), which represent the state-machines that of our language.

These files are included in our ProjectIT Language System project, which, when compiled, generates a library component (bservice.dll) that must be registered in the Windows registry, so that VS .NET IDE recognises the new language editor, and is therefore able to apply syntax highlighting, auto-complete and on-the-fly language corrections (syntactic and semantic), just like in any other VS .NET IDE supported language.

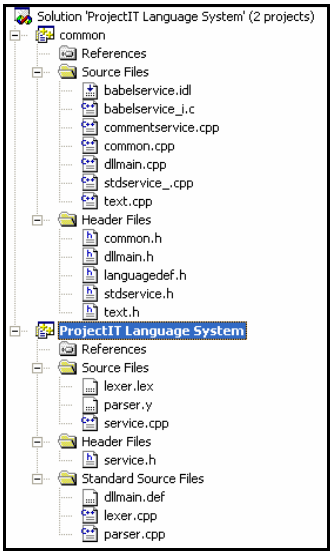

Figure 4 – The ProjectIT-RSL Solution Explorer

Figure 5 resumes the whole process of making ProjectIT-RSL recognised by the VS .NET editor.

[1] LALR(1) – *Look-Ahead(1) Left-to-Right*

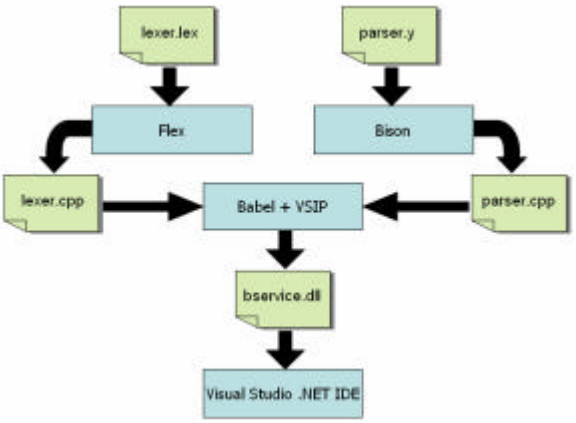

Figure 5 – Creating the ProjectIT-RSL Language System

4.2 Developing ProjectIT-RSL's Compiler

The next step was the creation of the compiler, again using Flex and Bison. The ProjectIT-RSL language definition contained in the files lexer.lex and parser.y was reused, with changes in the actions described by the grammar rules. These rules were mainly syntactic in the case of the editor, but for the compiler we needed to include additional semantic features, since the compiler is responsible for generating a XML document that represents the system described in ProjectIT-RSL. So, we added some extra verification, such as detecting the re-declaration of entities or actors, which will result in compiler errors and not in editor errors, since syntactically speaking the description written in the editor is correct. Figures 6 and 7 show how the code for the same block varies between the editor and the compiler version of parser.y.

```
ActorBlockContent
    : KWIDENTIFIER';' ActorBlockContent
        { $$ = $1;
          g_service->addScope($1, $2,
          ScopeClass, AccessPublic, StorageType,
          $1, $1, $1, &$1 ); }

    | KWIDENTIFIER { $$ = $1;
                     g_service->addScope($1, $1,
                     ScopeClass, AccessPublic, StorageType,
                     $1, $1, $1, &$1 ); }

    | ActorBlockContentErrors

    ;
```

Figure 6 – ActorBlockContent in parser.y, editor version

```
ActorBlockContent
    : KWIDENTIFIER';' ActorBlockContent
    {
            if (!Busca_Actor($1, listaActores))
                    Inserir_Actor("", $1, NULL, listaActores);
            else
                    imprime_erro("actor declaration", "this actor has already been declared");
    }

    | KWIDENTIFIER
    {
            if (!Busca_Actor($1, listaActores))
                    Inserir_Actor("", $1, NULL, listaActores);
            else
                    imprime_erro("actor declaration", "this actor has already been declared");
    }

       | ActorBlockContentErrors

       ;
```

Figure 7 – ActorBlockContent in parser.y, compiler version

The output of this step is the compiler, an executable component that results from processing the lexer.lex and parser.y files (compiler version) by the tools Flex and Bison. The VS .NET IDE recognizes this compiler by using the integration package that we will describe in the following section.

4.3 Developing ProjectIT-RSL's new Visual Studio .NET IDE project type

The last thing we have to do to completely integrate ProjectIT-RSL with the VS .NET IDE is to define a new type of project to aggregate the language's files, and which the developers can choose just like any other type of project. We have called it *ProjectIT-RSL Project*.

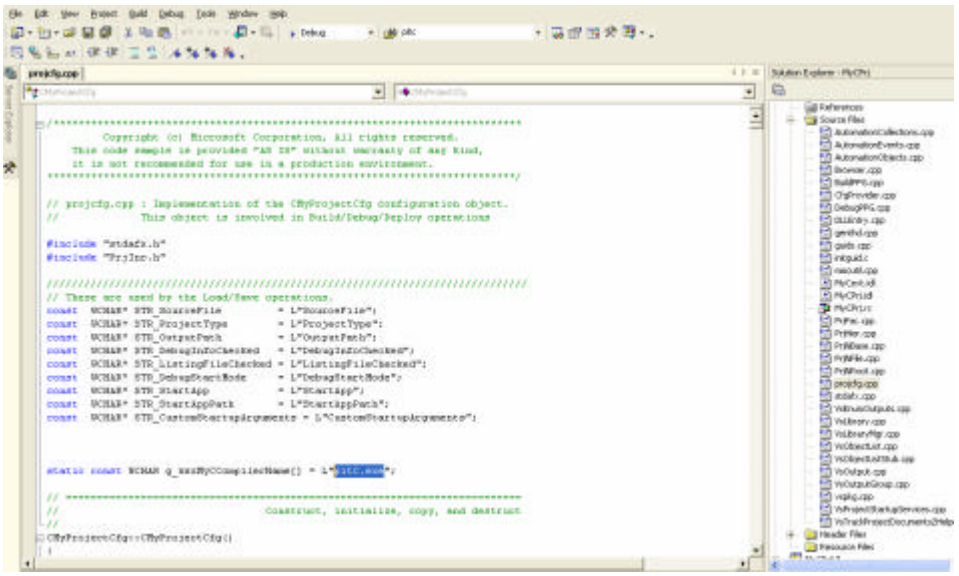

Figure 8 – Defining ProjectIT-RSL type of projects

In order to implement this feature, we used once again the VSIP libraries, but this time without the Babel abstraction layer, which is only useful for developing language services. This was the most difficult step in the process, due to the lack of available documentation, but finally we managed to find out (with the help of the examples provided by VSIP) the needed changes we had to make, such as where to put the name of the compiler file (shown in Figure 8), so that it runs when we build the project. The result of this step is also a library, registered in the Windows registry, and recognised by VS .NET when it is launched. Figure 9 resumes the dependencies between the three components we have built.

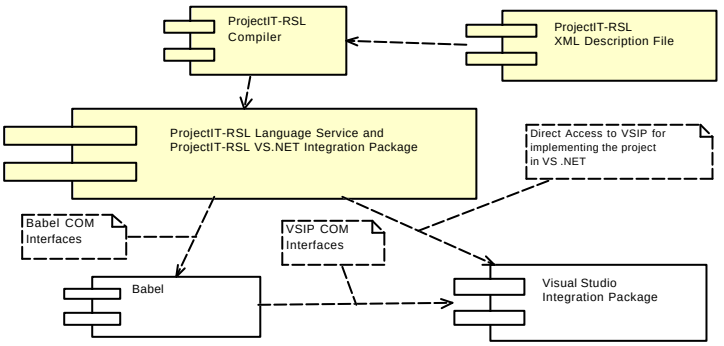

Figure 9 – ProjectIT-Requirements tools architecture

4.4 Creating a ProjectIT-RSL project

The creation of a new ProjectIT-RSL Project follows the normal steps of creating any VS .NET supported project: (1) project creation (Figure 10); (2) code edition (Figure 11); and (3) code compilation.

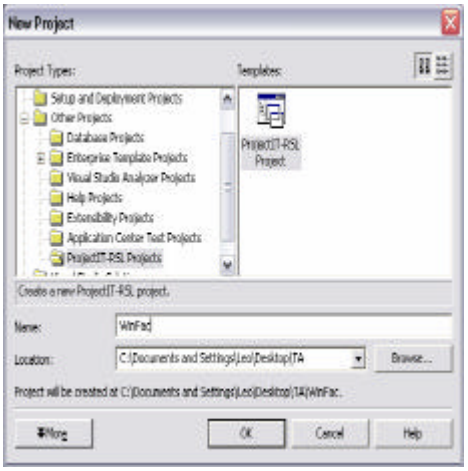

Figure 10 – Creating a ProjectIT-RSL project in VS .NET

This project uses the standard VS .NET editor, extended with the capability of writing ProjectIT–RSL sentences. The editor, shown in Figure 11, supports on-the-fly syntactic language verification and full syntax highlighting, which means that as we write, all expressions are validated (by the component generated by Bison for language checking, and according with the rules previously defined) and in case there is any error, it is immediately detected and highlighted. Besides, the auto-complete feature is also always available giving the writer suggestions of correction or of the next allowed and available term, according with the language.

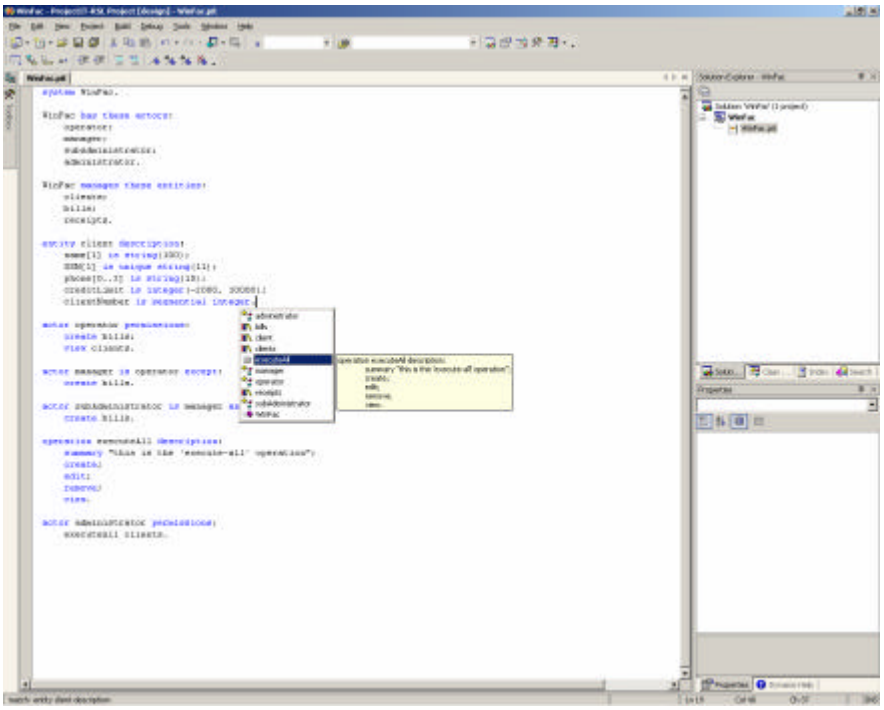

Figure 11 – The ProjectIT-RSL editor

Upon successful compilation, the compiler produces an XML file, which is currently the input to ProjectIT-MDD [6]. The idea is that this XML file will be a common representation of the information used by our requirements specification, model and code generation tools. Because the existing model and code generation tools (currently named XIS-Tools) were based on the XIS-UML profile [5] and developed prior to the existence of ProjectIT, they are not able to process part of the information contained in the XML file (for example, the information about the actors that interact with the system, and which defines the permissions of the users). For this reason, the XIS-UML profile is being reviewed, and renamed to PIT-UML profile, to become a unified metamodel, common to all our tools.

5 Conclusions and Future Work

This paper has described how we have used the extensibility options available in VS .NET, and in particular of the VSIP program, to support requirements specification. In the context of a new initiative we are developing, called ProjectIT, we have defined a new requirements specification language, ProjectIT-RSL, which should be understood by every stakeholder in the software process, and sufficiently precise to eliminate ambiguities that can cause difficulties to the rest of the process. To prove the ideas

behind ProjectIT-RSL, we chose the VS .NET IDE to use as the environment to write and process the requirements specification written in ProjectIT-RSL, using the built-in features provided by BS .NET that were particularly valuable to us (such as syntax highlighting and intellisense).

There are other initiatives that have used the extensibility mechanisms available in VSIP to integrate tools in VS .NET IDE. They include the possibility of allowing existing languages to be recognized in the VS .NET IDE or the addition of new tools and features to the IDE. However, and as far as we know, no one has tried to define a new requirements specification language and use it integrated with VS .NET IDE.

In the near future we will concentrate on the migration of the XIS-Tools [5] to the VS .NET platform and on the unification of XIS-UML profile with the metamodel of ProjectIT-RSL. We will also focus of the issues directly concerned with requirements specification, such as requirements reuse mechanisms and the specification of the goals of the system. When our ProjectIT-RSL and its supporting tools reach a sufficient maturity level, it is our intention to use them in real projects, to better test and demonstrate the ideas we are proposing.

References

1. Johnson, B., Skibo, C., Young, M., *Inside Microsoft Visual Studio .NET*. Microsoft Press, Washington, 2003
2. Kotonya, G., Sommerville, I., *Requirements Engineering Processes and Techniques*, New York. John Wiley & Sons, 1998
3. Moreira, V., ProjectPRO, Plataforma de Gestão de Requisitos, Relatório de Trabalho Final de Curso, Lisboa, July 2003
4. Richter, J., *Applied Microsoft.NET Framework Programming*. Microsoft Press, Washington, 2002
5. Silva, A., Lemos, G., Matias, T., Costa, M., The XIS Generative Programming Techniques, Proc. of the 27th Annual Int. Computer Software & Application Conference, Dallas, 2003
6. Silva, A., *O Programa de Investigação "ProjectIT"*, Technical report, V 1.0, October 2004, INESC-ID, in http://berlin.inesc.pt/alb/uploads/1/193/pit-white-paper-v1.0.pdf
7. Videira, C., Silva, A., *The ProjectIT-RSL Language Overview*, UML Modeling Languages and Applications: UML 2004 Satellite Activities, Lisbon, Portugal, October 2004
8. Videira, C., Silva, A., *ProjectIT-Requirements, a Formal and User-oriented Approach to Requirements Specification*, Actas de las IV Jornadas Iberoamericanas en Ingeniería del Software e Ingeniería del Conocimiento - Volumen I - pp 175-190, Madrid, Spain, November 2004
9. Videira, C., Silva, A., *A broad vision of ProjectIT-Requirements, a new approach for Requirements Engineering*, in Actas da 5ª Conferência da Associação Portuguesa de Sistemas de Informação, Lisbon, Portugal, November 2004