PICGAL: Practical use of Formal Specification to Develop a Complex Critical System

Lionel Devauchelle[1], Peter Gorm Larsen[2] and Henrik Voss[2]

[1] AEROSPATIALE espace et defense, department SY/YI -BP 3 002, F 78 133 Les Mureaux CEDEX, FRANCE
[2] IFAD (The Institute of Applied Computer Science), Forskerparken 10, DK-5230 Odense M, Denmark

Abstract. This paper reports on the experiment PICGAL which aims to assess the benefits of using VDM to develop high reliability related software in the space industry in a practical way. The application used in this project is a code generator from a next generation environment to be used in the development of ground application software for boosters such as ARIANE V. The experiment is constructed as a parallel development of the code generator; using the conventional approach and using formal specification. This allows detailed measurements of the effects resulting from the introduction of VDM. This work is adding to the existing body of evidence of the effect of using a moderate amount of formal methods in an industrial context in a new critical domain. This paper provides an overview of the domain, the application and it shows how the formal specification has been structured. Finally, results and key lessons are presented.

1 Introduction

AEROSPATIALE Espace & Defense is prime contractor or industrial architect of space launchers and vehicles, and the critical role of software in this sector is increasing continuously. In the experiment described in this paper, we focus on the highly reliable ground application software development environment used to develop the required control command functionality. In order to decrease the cost of such developments as well as the associated time, a future generation environment entitled SCALA is being studied by AEROSPATIALE.

The SCALA environment will include a code generator from the control specification language SCALA to the ANSI C programming language. The dependability of the ground application software will rely on the reliability of the code generator. This component is a high dependability related piece of software, which therefore should be carefully checked and validated. In the PICGAL[3]

[3] PICGAL is an acronym for "Process Improvement experiment of a Code Generator to the Ariane Launcher" and the project is supported by the European Commission (ESSI project – PIE no 21 716).

The copyright for this paper is held by AEROSPATIALE. It must not be reproduced without written permission from AEROSPATIALE. Requests should be made to the first author of this article. © · AEROSPATIALE · 1997.

project this code generator is developed twice. One development uses the conventional development approach at AEROSPATIALE with natural text in the early phases of the life-cycle, and another development uses VDM with the IFAD VDM-SL Toolbox and the associated VDM-SL_{to}C++ Code Generator. During the project a number of metrics are being collected to measure the effect of using the VDM technology.

The Vienna Development Method (VDM) [5] is one of the most mature formal methods, primarily intended for formal specification and subsequent development of functional aspects of software systems. Its specification language VDM-SL [1] is used during the specification and design phases of a software development project, and it supports the production of correct high quality software. VDM-SL is standardised under the auspices of the International Standard Organization (ISO) [8].

A modular extension to ISO VDM-SL is supported by the IFAD VDM-SL Toolbox [2, 7, 9]. The Toolbox supports extensive type checking, LaTeX pretty-printing facilities, test coverage, code generation, interpretation and many debugging facilities. A large subset of IFAD VDM-SL can be execute by the existing interpreter of the Toolbox. As part of the modular extension the IFAD VDM-SL Toolbox supports Dynamic Linked modules (DL modules) [4]. DL modules are used to describe the interface between modules which are fully specified in VDM-SL and parts of the overall system which are only available as C++ code. This facility enables users to employ existing C++ libraries while a specification is being interpreted/debugged.

Related work has been carried out using an earlier version of the same technology by British Aerospace for communications software [3, 10, 6]. The results of this work were very promising and showed that the use of formal specification for critical components was slightly less expensive than the conventional way, and in addition an exceptional situation was discovered already at the system design phase which was never discovered with the conventional development process. At the time of this experiment no code generator for VDM-SL was available so the final code was produced manually. In the PICGAL experiment we use the VDM-SL_{to}C++ Code Generator.

In line with the experiment at British Aerospace we have focused, on purpose, on using VDM-SL as a high-level language to describe a model of the system. Thus, we have not stressed formal proof or refinement at any stage in this project. The motivation for this more pragmatic approach is that we feel that it is better first to learn to think in terms of VDM concepts and be able to validate such a model using well-known techniques such as testing. However, we consider this to be a first step towards a more rigorous development of industrial software systems. When the engineers are confident with this technology it may be feasible to introduce more components from the VDM methodology and introduce formal verification. We believe that making too large a step in one go would be too difficult for industrial engineers in most cases.

The copyright for this paper is held by AEROSPATIALE. It must not be reproduced without written permission from AEROSPATIALE. Requests should be made to the first author of this article. © · AEROSPATIALE · 1997.

This paper is organised as follows: in the next section a short introduction to the application domain and the purpose of the SCALA environment is given. In Section 3 it is explained how the experiment is organised in a parallel development of the SCALA to C code generator. This section also includes material about the background for the use of VDM and the comparisons and tests to be performed during the experiment in order to assess the benefits of using VDM in this application. The following three sections present an overview of the specification and the way it is being validated. Section 7 presents some results from the project and this is followed by a section illustrating how we envisage this kind of technology to be used in the space domain in the future. Finally, a few concluding remarks are given.

2 The Application Domain

In this section an introduction to the control command systems domain and the SCALA environment is given.

2.1 Control Command Systems

The control command systems are ground systems used to check-out the different parts of a booster during its assembly process and to perform the lift-off. Their main functions are:

- to bring the booster or a part of it into operation;
- to test the availability of the booster equipment;
- to synchronise the operation of the booster with external events during lift-off; and
- to monitor the equipment in order to keep the booster safe and to avoid damage to the environment.

An overview of a control command stand is shown in Fig. 1. Many of these components are not of relevance to this project, but it provides an idea about the environment in which the work presented here is carried out. For this paper the most important part is the software architecture which is divided into three layers:

Level 1 includes the operating system and the hardware handlers;
Level 2 provides basic services for the application software such as sequencing, and basic check-out functions; and
Level 3 is the application software layer. It is to this kind of software that the SCALA code generator described in this paper is to be applied.

2.2 The Purpose of the SCALA Environment

In the development of Level 3 software components of a control command application there is currently a relatively long turn-around time when requirements changes are introduced. To increase the productivity and decrease the

The copyright for this paper is held by AEROSPATIALE. It must not be reproduced without written permission from AEROSPATIALE. Requests should be made to the first author of this article. © · AEROSPATIALE · 1997.

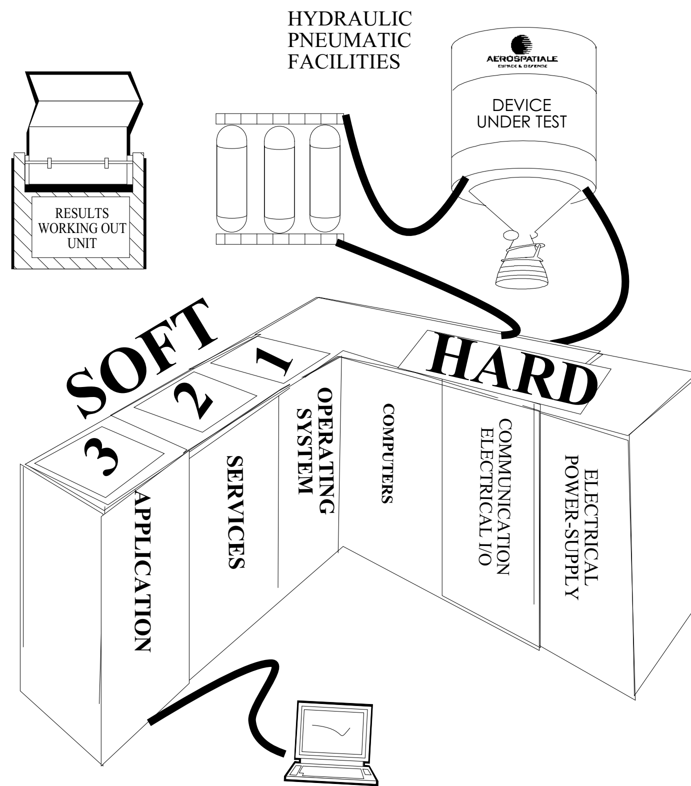

Fig. 1. Control Command Stand Overview

turn-around time of the application software, AEROSPATIALE is planning to produce a new development environment named SCALA. This environment will include the following tools:

– a specification editor,
– a specification validation tool, and
– an automatic code generator.

The most critical part of this environment is the code generator. The reliability of the developed application will depend on it. This automatic code generator will automatically translate SCALA specifications into the ANSI C programming language.

SCALA will be used by booster equipment engineers or system engineers to specify easily readable, comprehensive and unambiguous control command requirements as shown in Fig. 2.

The copyright for this paper is held by AEROSPATIALE. It must not be reproduced without written permission from AEROSPATIALE. Requests should be made to the first author of this article. © · AEROSPATIALE · 1997.

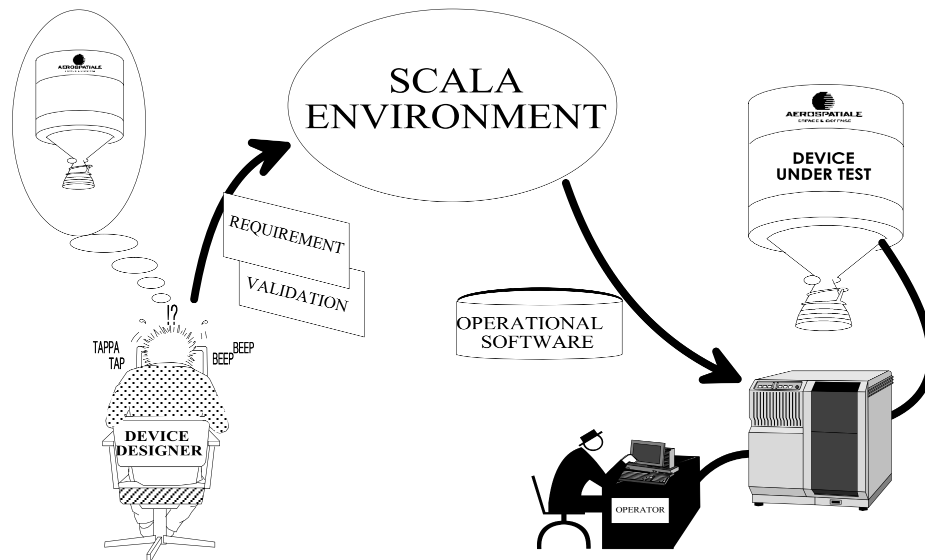

Fig. 2. SCALA Development Environment Use

3 Description of the Experiment

3.1 Parallel Development

Fig. 3 provides an overview of the different phases of the two parallel and separate developments of the code generator. The initial requirements phase determined the purpose of the generator but do not indicate its structure, the algorithms to be used or the output code. The general design phase is common to both developments: the baseline and the PICGAL line. With the structure of the generated code, this phase provides the functions of the generator as well as some required algorithms (e.g. the naming of data).

The conventional development consists of the following steps:

- software design to refine the general design in accordance with the implementation constraints;
- coding and unit testing; and
- integration.

The VDM development consists of the following steps:

- architecture design to refine the general design in accordance with the VDM modelling principles;
- VDM modelling;
- specification test;
- code generating through the VDM tool and complementary coding in C++.

The copyright for this paper is held by AEROSPATIALE. It must not be reproduced without written permission from AEROSPATIALE. Requests should be made to the first author of this article. © · AEROSPATIALE · 1997.

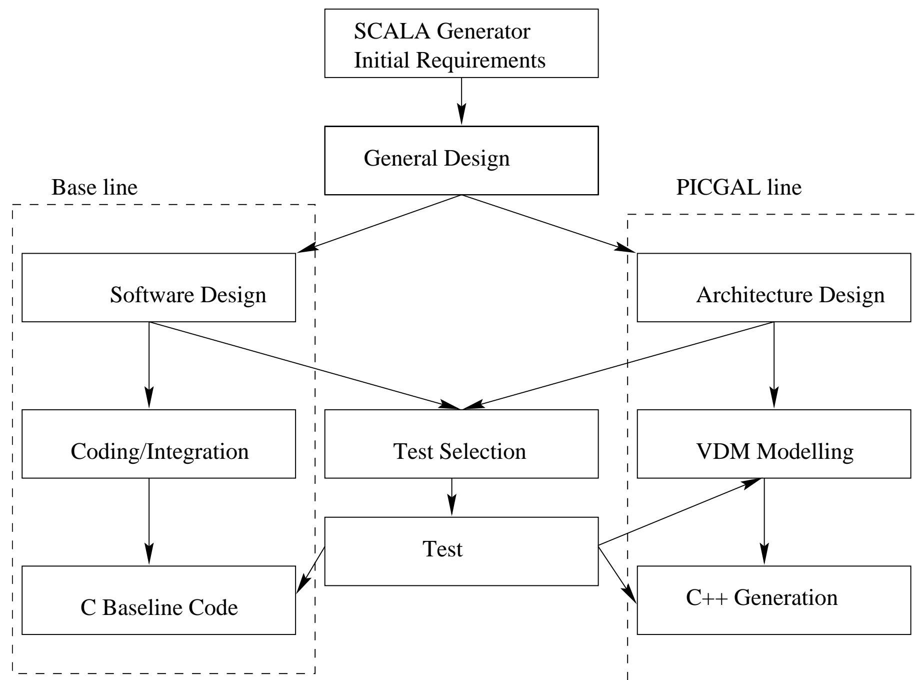

Fig. 3. PICGAL Project Phases

In addition, a test environment testing both systems with the same test cases is developed. In order to prevent the results from being biased by the skills of the developing engineers, one key engineer is used on both developments. In order to illustrate the benefits of the VDM approach this engineer always work on the baseline development before dealing with the corresponding VDM work.

3.2 Background for the use of VDM

The AEROSPATIALE team had no prior experience with formal methods. VDM was chosen because of the availability of strong tool support in form of the IFAD VDM-SL Toolbox. The team was trained by IFAD in VDM modelling and use of the different features of the VDM-SL Toolbox during two one week courses.

During the project, IFAD has been acting as a VDM consultant. Periodically IFAD has reviewed the VDM model developed by the AEROSPATIALE team. However, both in the consultancy and in the reviews IFAD has only pointed out problems and suggestions for the kind of constructs which could be used to improve a given specification. No part of the specification has been written by IFAD so that a fair comparison in the experiment can be provided.

The copyright for this paper is held by AEROSPATIALE. It must not be reproduced without written permission from AEROSPATIALE. Requests should be made to the first author of this article. © · AEROSPATIALE · 1997.

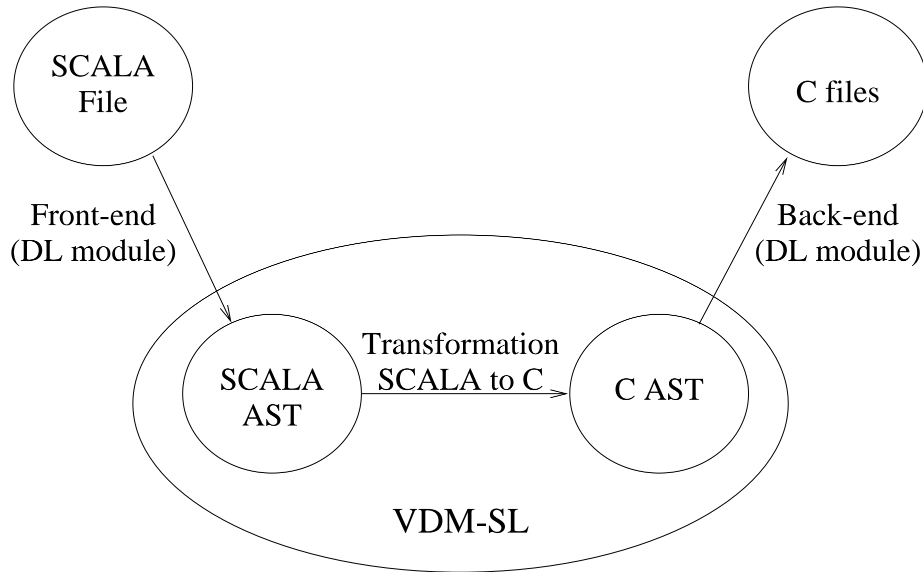

Fig. 4. The general structure of the SCALA to C code generator.

4 Structure of the VDM Model

In this section we will present the architectural structure of the SCALA to C code generator. The general structure of the specification is presented and subsequently broken down into a modular structure showing the dependencies between these modules.

4.1 General Structure of the Specification

The purpose of the final code generator is to translate a SCALA ASCII file into C ASCII files (`.cc` and `.h` files). VDM-SL does not provide I/O facilities so this notation cannot be used for the entire translation. Thus, the code generator has been divided into three components as shown in Fig. 4. The first phase is the front-end. The purpose of the front-end is to parse the SCALA ASCII file and produce an intermediate VDM-SL representation in terms of a SCALA abstract syntax tree. In the second phase the VDM-SL specification transforms the abstract syntax of SCALA into the abstract syntax of C. The third part is the back-end, and it converts the C AST into files containing the concrete ASCII representation of the generated C program.

The front-end and the back-end are written directly in C++. In order to include these parts in the VDM-SL specification dynamically linked modules have been used. This enables the VDM-SL part to be interpreted in combination with the front-end and the back-end.

The copyright for this paper is held by AEROSPATIALE. It must not be reproduced without written permission from AEROSPATIALE. Requests should be made to the first author of this article. © · AEROSPATIALE · 1997.

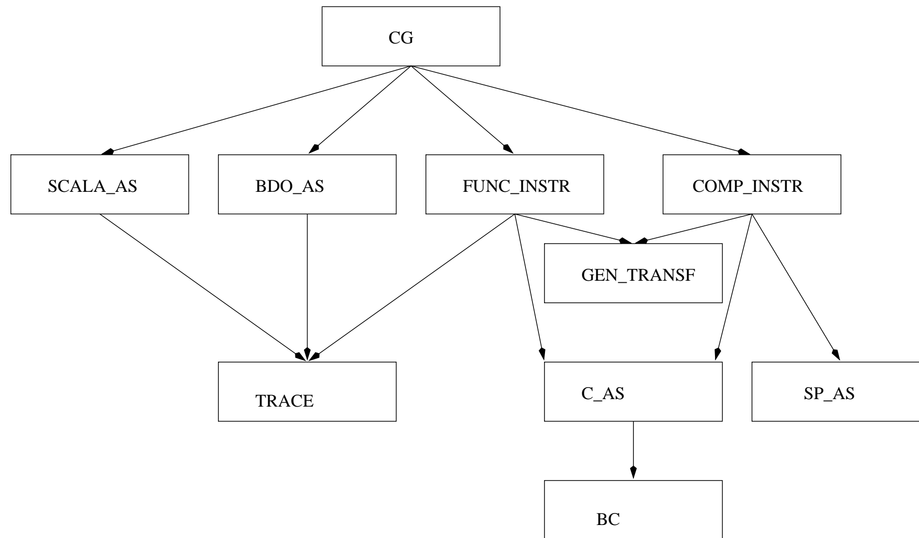

Fig. 5. Specification Module Overview

4.2 Modular Structure of the Specification

The modular structure of the specification and the dependencies between the different modules can be seen in Fig. 5. An arrow between two modules indicates that the first module is using constructs from the second one.

The main module is the `CG` module which is responsible for the overall code generation of a SCALA specification into a number of C files. It needs information about the abstract syntax of the SCALA notation, `SCALA_AS`, and the operational database information, `BDO_AS` providing information of the data used. The transformation of different kinds of instructions takes place in respectively the `FUNC_INST`, `COMP_INST` and `GEN_TRANSF` modules. The abstract syntax and the functions to build C abstract syntax are placed in `C_AS` and `BC`. The `SP_AS` module provides an interface to the subprograms used as level 2 services. Finally, the `TRACE` module is used to manage trace and error messages.

5 Transformation of a selected SCALA Instruction

The SCALA language contains approximately 30 different statements dealing with actions of setting, acquiring, checking and computing as well as with the associated data management.

In this section we will show how the transformation from SCALA to C of a single SCALA instruction is specified using VDM-SL. The transformation of each instruction is documented using:

The copyright for this paper is held by AEROSPATIALE. It must not be reproduced without written permission from AEROSPATIALE. Requests should be made to the first author of this article. © · AEROSPATIALE · 1997.

1. Examples of the SCALA instruction and its corresponding C code in different cases.
2. Textual description of the steps in the transformation.
3. A formal VDM-SL definition of the transformation.

In order to illustrate the approach we show below how the SCALA instruction `AFFECTER` (meaning "change" in French) is transformed into C. In addition to the items above, we have included the definition of the abstract syntax for the `AFFECTER` instruction and the abstract syntax definition of C needed to understand the VDM-SL definition of the transformation.

The SCALA instruction `AFFECTER` and its corresponding C code can be divided into the following five cases:

SCALA instruction	Corresponding C code
`AFFECTER var1 ind1 var2 ind2`	`var1[ind1-1] = var2[ind2-1]`
`AFFECTER var1 ind1 var2`	`var1[ind1-1] = var2`
`AFFECTER var1 var2 ind2`	`var1 = var2[ind2-1]`
`AFFECTER var1 var2` (in case of numeric copy)	`var1 = var2`
`AFFECTER var1 var2` (in case of string copy)	`strcpy(var1,var2)`

The abstract syntax of the `AFFECTER` instruction is defined below. The `AFFECTER` instruction will always have a variable, `var1`, on the left-hand side of the assignment and a variable or a value, `var2`, on the right-hand side of the assignment. Furthermore, `var1` and `var2` can have optional indices, `ind1` and `ind2`, which can be values or variables.

```
Affecter :: var1 : Variable
            ind1 : [Variable | Valeur]
            var2 : Variable | Valeur
            ind2 : [Variable | Valeur];
```

As shown in the table above, the `AFFECTER` instruction is transformed into an assign statement or a function call to `strcpy`. The C abstract syntax of a function call is defined as:

```
FctCall:: fct : Id
          arg : seq of Expr
```

In order to build C abstract syntax trees a number of auxiliary functions are defined. The function `GenFctCall` is one of those and it is used to build a C function call. A general principle in the specification has been to build C AST's only using such auxiliary functions. That is, nodes like `FctCall` will never be built using a record constructor expression anywhere else than in the auxiliary functions for this purpose.

The copyright for this paper is held by AEROSPATIALE. It must not be reproduced without written permission from AEROSPATIALE. Requests should be made to the first author of this article. © · AEROSPATIALE · 1997.

```
GenFctCall: Id * seq of Expr -> FctCall
GenFctCall(fct,args) ==
  mk_FctCall(fct,args)
```

The function `TransformAffecter` formalises the transformation from `Affecter` to C `Stmt`. The function makes use of some definitions which are not included in this paper. These can be divided into two main categories: 1) General auxiliary functions like `DataType`, which is used to look up the type of the variable on the left-hand side of the assignment, and 2) functions to build C AST's (`GenId`, `GenIntegerLit`, `GenFctCall`, `GenArrayApply` and `GenAsgnStmt`).

```
TransformAffecter: Affecter -> Stmt
TransformAffecter(mk_Affecter(var1,ind1,var2,ind2)) ==
  let id1 = GenId(var1),
      id2 = GenId(var2),
      one = GenIntegerLit(1)
  in
    if DataType(var1) = <car>
    then GenFctCall(StrCpyId,[id1,id2])
    else let e1 = if ind1 = nil
                  then id1
                  else GenArrayApply(id1,GenMin(id1,one)),
             e2 = if ind2 = nil
                  then id2
                  else GenArrayApply(id2,GenMin(id2,one))
         in
           GenAsgnStmt(e1,e2)
pre DataType(var1) = <car> => ind1 = nil and ind2 = nil
```

Notice that the structure of the `TransformAffecter` function is structured into five different cases corresponding to the different cases of the `AFFECTER` instruction. The first let expression simply provides a name for the two identifiers, `id1` and `id2` and the integer `one`. In case the left-hand side variable is defined to be a string the standard string-copy function must be called. Otherwise an assignment statement is made with or without indexing in an array depending on the value of the index parts of the `AFFECTER` instruction.

The precondition of `TransformAffecter` documents that if variable `var1` is a string then the instruction must have the form `AFFECTER var1 var2`. This strategy has been used to document assumptions about the SCALA specification used as input to the code generator. In addition to preconditions it has appeared to be very valuable to use invariants e.g. on the SCALA abstract syntax to document assumptions about SCALA files and instructions.

6 Testing the VDM Specification

The specification of the SCALA to C code generator is tested at two levels:

The copyright for this paper is held by AEROSPATIALE. It must not be reproduced without written permission from AEROSPATIALE. Requests should be made to the first author of this article. © · AEROSPATIALE · 1997.

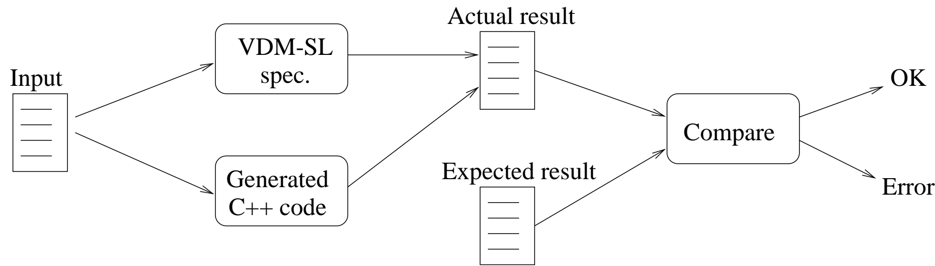

Fig. 6. Specification Test Overview

- at specification level using the Toolbox's interpreter and
- at implementation level using the automatically generated C++ code produced by the VDM-SL$_{\text{to}}$C++ Code Generator.

At both levels test coverage information is collected during the tests and metrics are used for comparison with the conventionally developed SCALA to C code generator.

In order to facilitate the test procedures a test environment has been developed. This test environment is common to both levels of testing, and it is illustrated in Fig. 6. It consists of:

- a graphical user interface;
- test drivers (UNIX scripts) for both levels of testing;
- test cases and expected results;
- editors and browsers to view results of tests, test coverage, create new test cases and update existing test cases;
- report (log) generation facilities;
- history information about earlier tests; and
- configuration management interface to the specification and implementation of the code generator.

This testing environment enables easy measurement of code metrics for the different versions of the developments.

7 Measured Results

7.1 Work Amount

The total amount of work turned out to be the same for both developments if the two weeks of direct training in VDM-SL and the C++ code generator are omitted. However, the additional time to get acquainted with the technology is included in this figure. Since this was the first project where the Aerospatiale team used VDM we estimate that in a new project, this team would be able to develop another system with less effort using the VDM technology.

The copyright for this paper is held by AEROSPATIALE. It must not be reproduced without written permission from AEROSPATIALE. Requests should be made to the first author of this article. © · AEROSPATIALE · 1997.

7.2 Test Coverage, Size and Speed of code

For the final source files from the baseline and VDM development a number of metrics was collected. In the table below some of these are shown:

	C baseline	VDM model	C++ generated from VDM	Total VDM C++
Total lines	3938	4252	11708	14565
Actual lines	2471	2215	11400	14000
Functions	79	175	175	200
Test coverage	82 %	89 %	67 %	67 %
CPU speed	100 %			60 %

The difference between most of these figures can be explained by the difference in the overall architecture of the two developments. The general approach of the C code produced in the baseline project is to directly translating from the sequence of characters in the various input files to the corresponding output files. As shown in Fig. 4 the VDM development is structured into three parts (input processing, transformation between abstract syntax trees, and output processing). Naturally this more structured approach requires more types and functions to be defined than the direct approach. We believe that this is the main reason why the size of the C baseline code and the VDM model is approximately equal (the type definitions of the SCALA abstract syntax and the C abstract syntax alone is 20 % of the VDM model). The VDM approach has also structured the model into significantly more functions than the C baseline development (175 versus 79). Concerning the test coverage it is slightly higher in the VDM model using the same set of test cases. The test coverage of the generated C++ code is rather low but this can be explained by the defensive programming style used by the C++ code generator. The main surprising result in this experiment was that the speed of the generated code actually was significantly faster than the hand coded C code from the baseline development. This is not because the C++ code generator is producing outstanding fast code, so it can only be explained by a better design encouraged by the use of VDM.

7.3 Lessons Learnt

The pragmatic approach focusing on validation of the VDM model using conventional testing techniques has shown to be appropriate for an application such as the code generator from SCALA to C. In comparison to a conventional baseline process, the VDM-SL notation has shown to be an adequate software specification language to be used after the engineering design phase to check requirements and to reduce functional complexity. The formalism is unambiguous allowing direct code generation for a large subset of VDM-SL. The VDM modelling allowed

The copyright for this paper is held by AEROSPATIALE. It must not be reproduced without written permission from AEROSPATIALE. Requests should be made to the first author of this article. © · AEROSPATIALE · 1997.

a better structured, more flexible and general development than the conventional way. This resulted in a better quality code generation from SCALA to C.

The VDM-SL notation proved to be adequate to model this application by providing the necessary constructs, keeping the data in a logic form in the functional description and by being capable of checking the generation algorithm.

Considering the extensive use of the interpreter and of the C++ code generator made in this experiment, most of the benefits accounted for come from the tool support provided by the IFAD VDM-SL Toolbox. In particular the ability to obtain fast feedback from a model using the interpreter and the powerful type-checking capabilities turned out to be valuable.

The following points have been learnt from the experiment :

Introduction of the VDM technology: Initial training (2 weeks) and the assistance of IFAD's consultants were sufficient to enable a correct use of the technology. Naturally, better skilled people would have benefited more from the particular notation existing in the VDM language. Trying to introduce the model to engineers who had not been trained turned out to be difficult. A general introduction to the basic concepts of formal methods is missing in order to spread the understanding of formal languages among the data processing engineers.

Scope of use: Formal methods are applied to the specification field of a critical software component and not of a whole software system. The VDM technology deals with functional requirements without offering easy modelling of hardware or real-time constraints. However, the VDM language covers a large field of data processing and can be used beyond small automatic logical security systems.

The reliability domain: The reliability of the software comes from careful test coverage measurements available from the VDM-SL Toolbox. The VDM language allows the tracking of invariants, pre- and post conditions which are important points in order to prevent large development teams from misunderstandings and software units from discrepancies.

Work efficiency: After requirements capturing, specification using formal methods can be applied to functional requirements in order to be modelled and checked. The work efficiency comes from the use of a high level language associated with an interpreter (which makes partial test during the modelling) and a code generator. However, the use of the VDM notation as a programming language turned out to be as productive as the C language (same programming time for the same size program). The interpreter allows tests at the modelling level in order to clearly identify the behaviour of the VDM concepts right after formulating them.

8 A Possible Future Use

According to the lessons learnt in the experiment, the VDM technology is a highly efficient modelling tool and allows functional test coverage measurement

The copyright for this paper is held by AEROSPATIALE. It must not be reproduced without written permission from AEROSPATIALE. Requests should be made to the first author of this article. © · AEROSPATIALE · 1997.

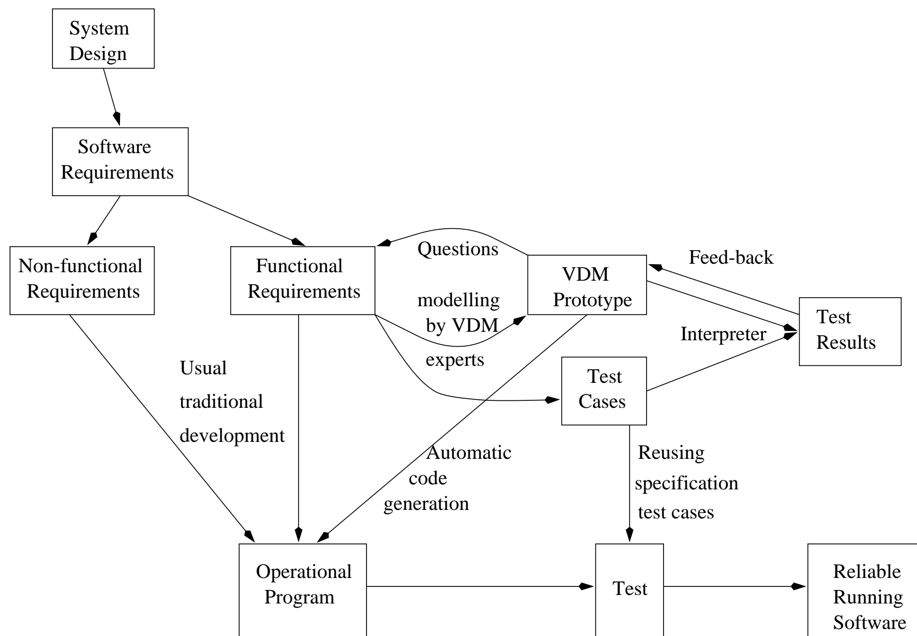

Fig. 7. Future Development Process Using the VDM Technology

at specification level. So, the VDM technology could be used by designers to check the accuracy, the completeness and the coherence of the functional requirements of critical software components.

Even with the use of VDM we feel that the natural language has to be maintained to write such requirements because it is commonly used in space projects and it belongs to the European industrial background. However, this does not prevent developers from using the VDM technology to analyse the requirements. In Fig. 7 it is illustrated how we envisage that the development process could be adjusted to incorporate the VDM technology in a practical way in space applications. In case the natural language explanation and the formal definition are in conflict with each other there must be a clear definition of which one takes precedence. We propose that the formal definition shall be the correct one if such situations should occur, but it is not certain that such an approach would be politically acceptable.

For natural language requirement specifications it is of primary importance to remove all discrepancies, misunderstandings and flaws that would prevent a reliable achievement. In a first step, formal method experts could use the VDM technology to model the functional requirements coming from the system design in order to detect any discrepancy or inaccuracy. This model could be interpreted in order to check the requirements and to study test cases to get an optimal functional test coverage. In a second step, the operational software could

The copyright for this paper is held by AEROSPATIALE. It must not be reproduced without written permission from AEROSPATIALE. Requests should be made to the first author of this article. © · AEROSPATIALE · 1997.

be developed as usual, taking into account all the requirements (operational and functional) and could benefit from the code automatically generated from the VDM model. The test cases used at specification level are selected to a high level of test coverage using the test coverage facility from the VDM-SL Toolbox. All these test cases are then reused for the validation of the final program.

9 Concluding Remarks

In this experiment we have not used the methodology part of the formal method VDM. We have taken a pragmatic approach and mainly used the VDM-SL notation for a high level description of the system under development. In comparison to a conventional process, the VDM-SL notation has showed to be an adequate software specification language after the engineering design phase. The formalism is unambiguous and allow direct correct code generation for a large subset of VDM-SL.

The VDM-SL notation proved to be adequate to model a code generator by providing the necessary constructs, keeping the data in a logic form in the functional description and in giving the capability of checking the generation algorithm. We have to point out that such a critical application does not include any synchronisation or real-time needs. If this had been the case we do not expect that we would be able to use the VDM-SL$_{\mathrm{to}}$C++ Code Generator, and VDM-SL might not be the best notation to use for such an application. We believe the main benefits of this particular notation is the powerful tool support which is provided by the IFAD VDM-SL Toolbox.

The VDM-SL notation was found easy to use after the first one-week training. A second week of training was used to upgrade the engineers from C to C++ and showing how to interface to the generated code from the VDM-SL$_{\mathrm{to}}$C++ Code Generator. However, it turned out that engineers who had not been trained had difficulty in understanding the VDM model being produced.

The specification has been written using an executable subset of VDM-SL on purpose. Dynamic link modules have been used to interface the specified parts with input and output to and from files. This enabled the specifiers to use the interpreter/debugger functionality from the Toolbox on the test arguments used for the final code as well. In addition, this also meant that we did not have to write a lot of C++ code manually.

Acknowledgements

We would like to thank the deputy manager Daniel Claude for setting up this project and the Commission of the European Union for financially supporting the project. Special thanks also go to the members of the development teams in particular to Michelle Lesage, Dennis Couturier and Robert Pastor. Finally, we would like to thank Sten Agerholm, Paul Mukherjee, Anne Berit Nielsen and Ole Storm for their valuable comments to an earlier version of this paper.

The copyright for this paper is held by AEROSPATIALE. It must not be reproduced without written permission from AEROSPATIALE. Requests should be made to the first author of this article. © · AEROSPATIALE · 1997.

References

1. John Dawes. *The VDM-SL Reference Guide*. Pitman, 1991. ISBN 0-273-03151-1.
2. René Elmstrøm, Peter Gorm Larsen, and Poul Bøgh Lassen. The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications. *ACM Sigplan Notices*, 29(9):77–80, September 1994.
3. John Fitzgerald, Peter Gorm Larsen, Tom Brookes, and Mike Green. *Applications of Formal Methods*, chapter 14. Developing a Security-critical System using Formal and Convential Methods, pages 333–356. Prentice-Hall International Series in Computer Science, 1995.
4. Brigitte Fröhlich and Peter Gorm Larsen. Combining VDM-SL Specifications with C++ Code. In Marie-Claude Gaudel and Jim Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 179–194. Springer-Verlag, March 1996.
5. Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990. ISBN 0-13-880733-7.
6. Peter Gorm Larsen, John Fitzgerald, and Tom Brookes. Applying Formal Specification in Industry. *IEEE Software*, 13(3):48–56, May 1996.
7. Paul Mukherjee. Computer-aided Validation of Formal Specifications. *Software Engineering Journal*, pages 133–140, July 1995.
8. P. G. Larsen and B. S. Hansen and H. Brunn N. Plat and H. Toetenel and D. J. Andrews and J. Dawes and G. Parkin and others. Information technology — Programming languages, their environments and system software interfaces — Vienna Development Method — Specification Language — Part 1: Base language, December 1996.
9. The VDM Tool Group. The IFAD VDM-SL Language. Technical report, IFAD, May 1996. IFAD-VDM-1.
10. T.M. Brookes and J.S. Fitzgerald and P.G. Larsen. Formal and Informal Specifications of a secure System Component: Final Results in a Comparative Study. In Marie-Claude Gaudel and Jim Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 214–227. Springer-Verlag, March 1996.

This article was processed using the LaTeX macro package with LLNCS style

The copyright for this paper is held by AEROSPATIALE. It must not be reproduced without written permission from AEROSPATIALE. Requests should be made to the first author of this article. © · AEROSPATIALE · 1997.