The Delft VDM-SL front-end

Nico Plat
Kees Pronk
Marcel Verhoef

Delft University of Technology
Faculty of Technical Mathematics and Informatics
P.O. Box 356, NL-2600 AJ Delft, The Netherlands
E-mail: {nico, kees, marcel}@dutiaa.tudelft.nl

1 Introduction

In this paper a short overview of the Delft VDM-SL front-end (front-end for short) is given. The front-end has been developed at Delft University of Technology. The tool is capable of checking syntactic and well-formedness properties of VDM specifications written in BSI/VDM-SL.

In this paper the functionality and characteristics of the front-end are described (section 2), an overview of its architecture is given (section 3), and an overview of the status of its development and perspectives are presented (section 4). An extensive overview of the front-end is given in [Plat 88].

2 Functionality and characteristics of the front-end

The Delft VDM-SL front-end has the following functionality:

- **Syntax checking.** The front-end accepts the standard ASCII syntax for VDM-SL [BSI/VDM]. Simple error recovery facilities are provided.
- **Static-semantics checking.** The major part of the static-semantic checks is type checking. The type system that has been implemented is described in [Plat 90]. Other checks include scope checking and various minor checks.
- **Generating an intermediate representation of the specification.** The front-end generates an abstract form of the original specification, which is called 'intermediate representation' because it is intended to be used by other tools, e.g. a pretty-printer or a prototyping tool. The form of the intermediate representation itself can be easily maintained by an in-house developed tool called DIAS [Langsæter 91].
The front-end is *batch-oriented*, so, as is the case with conventional compilers, the user must provide an input file with an ASCII VDM specification, which is then analyzed by the front-end. The user has the following options:

1. having only error messages (if any) generated by the front-end;
2. having a listing of the input generated, annotated with error messages;
3. using the front-end in combination with an editor, so that erroneous specifications can directly be changed, after which a new check-edit session can directly be started.

The front-end has been implemented on a number of UNIX-based platforms. Originally implemented on a VAX 11/750, then ported to a Sun 3/60, we are now in the process of porting the front-end to a Sun SPARC-2 platform, which will make a positive contribution to the front-end's performance (currently 10 lines of VDM-SL per second).

3 Architecture of the front-end

The implementation of the front-end is based on an *attribute grammar* (AG). AGs were introduced by Knuth [Knuth 68] as a method for defining semantics of programming languages. An AG is based on a context-free grammar, in which both the terminal and nonterminal symbols can be augmented with *attributes*. An attribute is used to hold a semantical value associated with a symbol in the grammar. By defining relationships between attributes, semantic properties of the programming language can be defined.

A system capable of evaluating the attribute values for a specific programme (or in our case: a VDM specification) is called an *attribute evaluator*. For most classes of AGs such attribute evaluators can be automatically generated. We have used the GAG system [Kastens 82] for the implementation of the front-end. GAG is a member of a family of compiler construction tools originally developed at the University of Karlsruhe. GAG takes an AG written in the language ALADIN as its input, from which the attribute evaluator is constructed. GAG also generates a context-free grammar, suitable as input for the parser-generating system PGS, but we automatically transform this context-free grammar such that it is suitable as input for yacc instead.

The construction process of the front-end is shown in figure 1.

The execution of the front-end can be divided into two major phases (figure 2):

- In the first phase, the source file is scanned, parsed, and an internal representation is generated (the GAG tree) with initial values for the attributes. Error messages may, of course, be generated during scanning and parsing. The error messages are collected in an error message database.

- In the second phase attribute evaluation is performed by executing a number of walks over the GAG tree during which attribute values are computed. In this way the static-semantic properties of the specification are checked. Error messages are collected in the error message data base, and the intermediate representation is generated which is written to a file.
Figure 1: Construction process of the front-end

Figure 2: Architecture of the front-end

When the two phases have subsequently been executed, the main control part of the front-end makes the error messages available to the user.
4 Status and perspectives

The Delft VDM-SL front-end is not a commercial product. It grew out of an interest in both formal methods and techniques for constructing language processors. Nevertheless, as the front-end consumed more manpower and grew in size, and when the market situation for tool support for formal methods had been examined [Plat 89], it became clear that it might be worthwhile to make the front-end more widely available. In order to achieve this we are currently undertaking the following:

- **Extensive testing.** Testing and subsequent repair of the bugs found is essential in order to get the tool accepted by any user. The largest test case we are using is the proto-standard formal definition of Modula-2 [BSI/Modula-2], which has the additional benefit that the latter standard is checked as well.
- **Improving the user interface.** Currently, the user interface is rather old-fashioned. We are designing a window-oriented user interface under X-windows.
- **Providing auxiliary tools.** Specifications are usually written as part of a text document for which a text formatter is used. The de-facto standard for VDM specifications is \LaTeX\, therefore we are currently implementing a tool that is capable of generating ASCII specifications from \LaTeX\ input, and a tool that is capable of generating \LaTeX\ code from the intermediate representation generated by the front-end. A cross-reference generator is also under consideration.

The authors of this paper can be contacted to get more information on the front-end.

References


