Proceedings Fourth European EDIF Forum
11–12 Oct. 1990, Warrington, England, P. 170–175

EDIF Level–2 to Level–0 translation

dr.ir. J.T.J. van Eijndhoven
Eindhoven University of Technology, EH 7.31
Postbus 513, 5600 MB Eindhoven
The Netherlands
jos@es.ele.tue.nl

ABSTRACT

A short description of a program is given, which is able to translate one design in a full EDIF level–2 keyword level–3 input into an EDIF level–0 output text. The primary application of this program is simplifying EDIF interfaces to other programs and data sets in our CAD environment. The program evaluates the level–1 expressions and executes the level–2 control statements. Because the experience and support for the EDIF levels 1 and 2 is still very limited, it seems useful to publish some of the gained experience. Implementation warnings are given on parameter passing, lexical analysis, connectivity, and vectors of length one. Other remarks are on the EDIF language itself: the limited functionality of the miNoMax type, the scaling of integer parameters with a units declaration, the level–2 functionality, name scoping concerning libraries, and the structuring of the EDIF grammar definition. Nevertheless the EDIF standard and its EIA reference manual proved to be of high quality, and as result quite a compact full functioning program could be built.

INTRODUCTION

This paper presents the result of some work done to create an EDIF interface to software packages in our group. Writing any format as EDIF is in general quite simple: one needs to generate only one text out of many possibilities. However an EDIF input must be able to handle the full language: understanding a locally defined subset only is quite useless and actually violates the standard.

The functionality to enable an interface to a circuit simulator was one of the first design constraints, and for this purpose EDIF level–1 was at least required: the parameter passing mechanism to change subcircuits in a hierarchical structure was deemed necessary. Implementing a full level–1 interface is quite a job: it requires a symbol table for local variables with full support for name scope and hiding mechanisms, and capability to handle all data types and array dimensions, as well as expression evaluation and assignments. Having this made, the step to full EDIF level–2 with conditional and repeat statements is only a minor hop.

As result a program is developed capable of handling full EDIF level–2 input with the keyword level 3 macro expansion. For one specified design, the program will go through its top–down hierarchical design structure, and execute all level–2 control constructs and evaluate level–1 expressions. As result an EDIF level–0 output text is written, containing this design only. The output will of course contain different cell/view definitions for for each different parameter set applied to any cell/view used in the design. During this parsing and translating process, strict syntactical and semantical checking with extensive diagnostic messages is provided. As result the actual interfaces to different programs/data sets in the IC design environment can be simplified and released from the level–1 and level–2 complexity. Furthermore they are allowed a more relaxed error reporting functionality. The program can optionally strip graphical, simulation, or userdata information from the description, to further simplify its output. Another application of the program will be verifying locally generated EDIF files, since we have yet little experience and software handling full EDIF.

THE PROGRAM

The program is built with a few relatively independent section: A lexical analyzer, a macro expansion unit, a recursive descent parser, and a double phase execution unit. The most important data structures in the program are the keyword table, mapping small positive integers to the keyword names and vice versa, A name hash table, storing all identifiers and strings from the input, and the parse tree: containing a linked–list tree structure mirroring the nested forms of the EDIF input text. This structuring of the program is shown in the following figure:

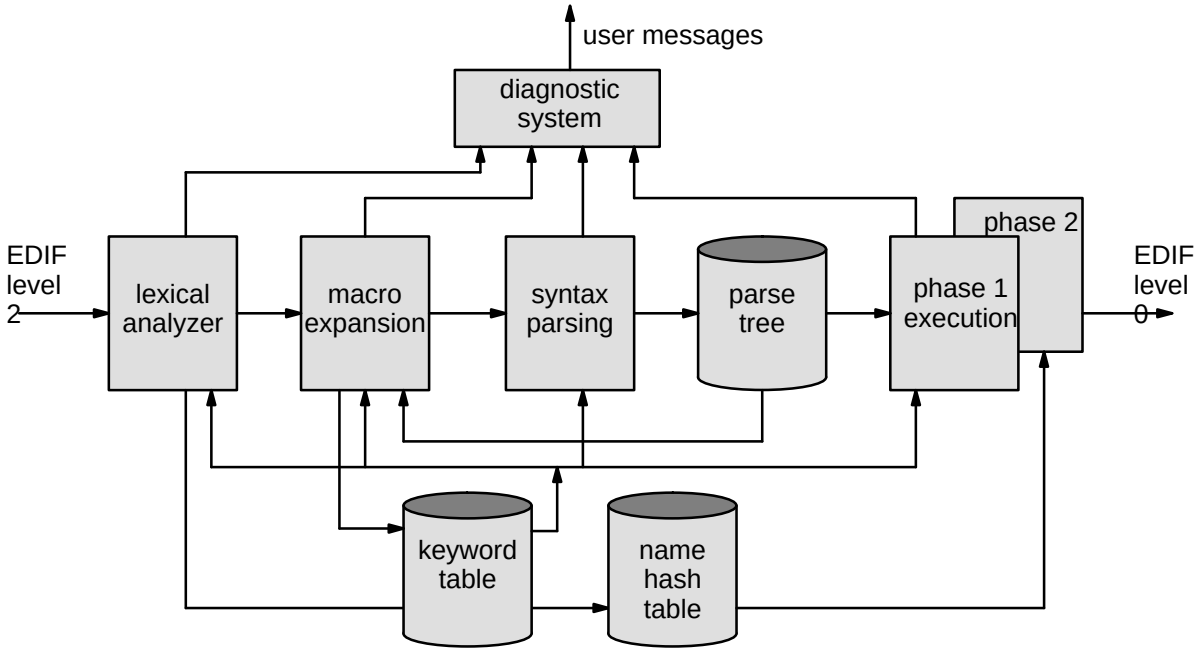

Figure 1: The program structure

The lexical analyzer recognizes in the input stream the following different symbol types: ”(keyword” combinations, identifiers, strings, integers, end–of–form ”)”, and end–of–file. It builds a hash table containing all identifiers and strings, providing a unique memory location for each different object. This way memory is saved and, more important, allows name comparison in the program to be done by (C–) pointer value comparison in stead of full string compares.

The macro expansion utility provides a full level–3 keyword unaliasing and macro expansion, by replacement with the definition already stored in the parse tree. This EDIF macro utility is extremely powerful, allowing compact specifications generating significant and complex structures.

The syntax parsing is done by a recursive descent parser, which is a very simple approach and nicely fits the EDIF grammar structure. It doesn’t make use of large tables and (explicitly implemented) stacks as for instance YACC parsers do.To allow a convenient specification of this parser (the grammar) the standard C preprocessor is used. This results in a compact, easy to read specification, exactly mirroring the EDIF grammar rules as specified in the manual. A few lines of this recursive descent parser are shown as example:

```
RULE Pdesign IS ZERO(n) FORM(Kdesign)
       ONCE          PdesignNameDef() && PcellRef()
       REPEAT G(n,Pstatus) || Pproperty() || Pcomment() || PuserData()
       ENDFORM

RULE Pentry IS FORM(Kentry)
       ONCE          (Pmatch() || Pchange() || Psteady()) &&
                     (PlogicRef() || PportRef() || PnoChange() || Ptable())
       OPTIONAL      Pdelay() || PloadDelay()
       ENDFORM
```

Caption 1: The recursive descent parser

For every grammar rule of EDIF, a corresponding ’rule’ is written this way. The above text is intuitively clear, which is attractive for a flexible and error–free specification. The construct ’ZERO(n) ... G(n,Pstatus)’ is used as guard for at most one occurence of ’status’ in the ’design’ rule.The preprocessor (cpp) macro definitions translate each ’rule’ into a real C function. Every function returns ’false’ if the current input token doesn’t match the start of its ’rule’. If the token does match, the corresponding ruleis read by this functions, which afterwards returns ’true’. For the basic tokens special functions are added such as ’Pinteger()’ or ’Pidentifier()’. If they match the current input token, it is added to a parsetree datastructurewhich obtains the same hierarchical tree structure as the nesting of the forms, and a new token is fetched from the input (the macro preprocessor). If the current input token cannot be matched, an error message is issued, and

the rest of the current form is skipped. The parsing will then continue, checking for more errors.The error message will show the unmatched token, the corresponding line number in the input file, and the keyword of the surrounding form which fails.

After storing the full EDIF input, the actual execution starts, reducing the EDIF level–2 input to level–0 output. This is done in two stages: The first stage processes the specified design hierarchy in top–down order, executing all level–2 control statements, and evaluating all level–1expressions. During this phase all different parameter sets passed to all used views are catalogued. The second stage again executes all used cells/views for each parameter set but now in the linear order of the EDIF input text, actually writing EDIF output during the execution. During this second phase, new view names are introduced if multiple parameter sets are attached to one view in the input, and for each of these parameter sets the complete body is written. Although a single–phase execution scheme could be made, this doesn't seem an attractive option: A hierarchical top–down execution is required to obtain the proper parameter passing. Really writing the EDIF level–0 output during this phase would result in a conflicting (actually opposite) order in the output, with respect to cell definitions and instantiations. Solving this by using different output files which could be merged later, isn't fit very well for current generation workstations, since these are strong in CPU power, but weak in disk I/O. The input to output translation is illustrated by the following example:

```
(cell passpar (view circuit
       (interface
              (parameter par (integer)))
       (contents
              (instance P (viewRef circuit (cellRef getpar))
                     (parameterAssign par (integer par)))
.
.
(cell setpar (view circuit
       (interface)
       (contents
              (instance P1 (viewRef circuit (cellRef passpar))
                     (parameterAssign par (integer 1)))
              (instance P2 (viewRef circuit (cellRef passpar ))
                     (parameterAssign par (integer 2)))
              (instance P3 (viewRef circuit (cellRef passpar ))
                     (parameterAssign par (integer 2)))
```

Caption 2a: Parameters and instances in EDIF input

is translated into:

```
(cell passpar
(view circuit_0
       (interface)
       (contents
              (instance P (viewRef circuit_1 (cellRef getpar))
                     (parameterAssign par (integer 2))))
(view circuit_1
       (interface)
       (contents
              (instance P (viewRef circuit_0 (cellRef getpar))
                     (parameterAssign par (integer 1))))
. . .
(cell setpar
(view circuit
       (interface)
       (contents
              (instance P1 (viewRef circuit_1 (cellRef passpar))
                     (parameterAssign par (integer 1)))
              (instance P2 (viewRef circuit_0 (cellRef passpar))
                     (parameterAssign par (integer 2)))
              (instance P3 (viewRef circuit_0 (cellRef passpar))
                     (parameterAssign par (integer 2)))
```

Caption 2b: Parameters and instances in EDIF output

EDIF IMPLEMENTATION ASPECTS

During the development of the program, a growing admiration was felt for the contributors to the EDIF version 2.0.0 EIA reference manual. Almost all anticipated problems were correctly solved by small remarks on the EDIF semantics and allowed use. Below a few remarks are given that might help or warn other implementors of EDIF level–1 or –2 interfaces.

Parameter assignment

The `(parameterAssign` form is different and considerably more difficult to implement then the `(assign` form. When an `(assign` is executed, the corresponding variable declaration has been executed before.Therefor the variable name is already known in the current scope, and the variable was already initialized with its type, array dimension,and storage space.

In case of the `(parameterAssign` the corresponding `(parameter` declarations contained in another cell/view context, maybe even in another library. Although this declaration is always located before the `(parameterAssign`, it cannot really be processed before. This is illustrated with the following example:

```
(library A (technology (constant width 32))
       (cell A (view A
              (interface
                     (parameter length (integer))
                     (parameter (array data length width) (boolean))
.
.
(library B
       (cell B (view B
              (interface
                     (parameter b (boolean (true)))
              (contents (instance a (viewRef A (cellRef A (libraryRef A)))
                     (parameterAssign (member data 5 1) (boolean b))
                     (constant length (integer 8))
                     (parameterAssign length (integer length))
```

Caption 3: Parameter assign and declaration order

It is clear that in this case the `(parameterAssign` to data must be executed within the scope of cell B to solve the reference to b, and then ”remembered”: the real declaration of data, initializing its name, type, and storage space must wait at least for the `(parameterAssign` to `length`, which is done later. However even just after `length` is assigned, it is not practical to perform the declaration of `data`, since this would require a context switch into the environment of cell A, library A, revealing the value of `width`. Consequently, the `(parameterAssign` to `data` must be evaluated and then ”remembered”, until the contents of library A and cell A are really processed, providing the corresponding declaration. This ”remembering” of assignments without having available the corresponding declaration is never needed for an `(assign` to a `(variable`.

The lexical analyzer

The in UNIX environments omnipresent ’lex’ lexical analyzer generator is not very suitable for EDIF, since the standard ’lex’ cannot easily handle the unlimited length of EDIF strings. Luckily a specialized EDIF lexical analyzer is easily implemented by hand.

Graphics and connectivity

When an application is interested in the electrical network connectivity information only, it seems attractive to just skip and discard all graphical information in the EDIF input. However in EDIF level–1 and level–2 this cannot be done in general: Due to the `(xCoord` and `(yCoord` operators, the results of graphical data, expressions, and transformations can easily influence the basic network connectivity information.

Vectors of length one

A typedValue is used to specify a value to be assigned to a variable, constant or parameter. Its syntax makes no difference between a scalar value or a vector of length one. Especially in the case of a `(parameterAssign` this is cumbersome: Since the `(parameter` declaration is not yet processed and available, the program would assume to have found a scalar value. However in the later processed `(parameter` declaration the difference between a scalar and a vector of length one IS made! Therefor a silent and automatic dimensionality conversion is needed in such a case. This can also be seen as an inconsistency in EDIF.

COMMENTS ON THE EDIF STANDARD

Besides remarks on implementation issues, the work on processing the level–1 and level–2 EDIF forms also leads to some remarks on the EDIF standard itself.

Functionality of miNoMax

The EDIF value type `(miNoMax` seems almost useless within its current restrictions. These values can be created with the `(mnm` construct, and copied between variables and parameters with the `(assign` and `(parameterAssign` respectively. However nothing else can be done with them! Since there is no possibility to reference these in `numberValue` contexts, they cannot influence network or graphical information. It seems that –either– a statement is required saying that a reference in a `numberValue` context is allowed and returns the nominal value, –and/or–new operators are needed to address the individual min, nom and max fields.

miNoMax range checking

Suppose a cell/view has an interface declaration containing `(parameter A (miNoMax (mnm 0 1 10)))`. Then an instantiation of this cell/view could specify a `(parameterAssign A (miNoMax –1))`. The program would first silently type upgrade the integer '–1' to an `(mnm (undefined) –1 (undefined))`, and this entire `(mnm` value would then be copied into A. Therefor no errors or warnings would be issued, saying that '–1' falls outside the original [0,10] bounds, because the bounds itself are thrown away.

units on integer parameters

The `(parameter (unit` declaration is to be able to use convenient parameter values, scaled with respect to the 'real world' SI values. The `(scale` forms are required to pass these values out to other libraries. If such a value is passed to another library, the value local to the receiving library is obtained by multiplying the originally assigned value with the ratio of the two `(scales` of this `(unit` in both libraries. For `(integer` values this seems a problem, since multiplying with this ratio could easily lead to fractions. It seems natural to EDIF to have the new value type upgraded to a `numberValue`, as the `(divide` does. This silent type upgrading from integer to number is normal to EDIF expressions, and is determined by the TYPE of the operands and the respective operation, NOT by the actual VALUE of the operands. This would mean that a `(parameterAssign size (integer 6))` to a `(parameter size (integer) (unit distance))` in another library will be ALWAYS type upgraded to a `numberValue` –irrespective of the actual `(scale` values– and hence cause a fatal type clash with the declared integer parameter, since a silent type downgrading from number to integer is never done in EDIF expressions.This can be seen as an inconsistency in the EDIF standard. A note on how to handle this seems required.

level–2 as meta language

From a language design point of view, the EDIF level–1 and level–2 extensions could be seen as a meta–language, providing a full programming language on top of the EDIF level–0 basic data. Seen this way it is a pity that control statements like `(if` and `(while` are allowed at so few locations, and it can be considered dirty that they are not allowed to contain all statements (forms) that can surround them. In particular it is very disappointing that although `(if` and `(while` are allowed in `(interface`, they can not contain `(parameter` or `(port` declarations.

assign in level 2

The EDIF level–1 extension with respect to level–0 is the introduction of `(variables`, `(constants`, `(parameters`, a set of operators, and expressions. It seems strange that the `(assign` form is restricted to level–2 only, whereas its functionality clearly is tied to the other level–1 forms.

name scoping

A more accurate and exhaustive specification of the name scoping mechanism, would improve the quality of the EDIF manual. It seems that `(variables` are not allowed in the `(technology` section just to forbid data exchange between different cells, outside the parameter mechanism.Of course such communication would lead to ill–specified systems, since the EDIF cells have no prescribed execution order. To really enforce this limitation, it would be needed that the `(userData` construct introduces a new scope level.

Grammatical structuring

This program evaluates all level–1 constructs. This requires among others the replacement of variable and parameter names by their values, in the locations where these represent expressions. However from the grammar rules itself it is not immediately obvious which occurrences should be replaced. Take as example the last line of caption 3: '`(parameterAssign length (integer length)`'. It is clear that this line must be translated into '`(parameterAssign length (integer 8)`'. However this is not

immediately clear to the program: both occurrences of length are according to the EDIF grammar definition `valueNameRefs`. Although not impossible to find out now, a better structuring and naming of the grammar rules could help here considerably.

CONCLUSIONS

The EIA reference manual describing the EDIF 2.0.0 standard is of high quality, and contains sufficient information to build a program like this. The resulting program is quite compact, requiring no external files or tables. However a few remarks to the EDIF standard can be made, and small improvements with respect to the level–1 and level–2 extensions seem feasible.

The program can be really useful in simplifying the EDIF input to other design tools and data sets, especially those who itself cannot match internally the level–1 and level–2 complexity.

Another interesting usage of this program might be as a very general cell generator. By providing a flexible and parameterized input description, the program will generate different instances of this cell. It could there for be usefull as cell generator, generating netlist information as well as graphical and simulation data.