Interface Between Elfen and INVERSE

(PROGRESS REPORT)

Igor Grešovnik

Ljubljana, January 2004

Contents:

1. SPECIFICATIONS

1.1 Interface Functions

1.1.1 Elfen Functions Used by Inverse

Inverse needs utilities for accessing the database (i.e. reading results and changing input) and for running a certain part of analysis. In the Elfen-Inverse interface, Inverse can run the whole analysis at once, but can also interact with Elfen at certain points during the analysis using interrupts.

1.1.1.1 void define_elfendyn_user_interrupt_function (int (*func) (const char * message));

Installs user interrupt function (*func*), i.e. sets the function that is executed at every interrupt in the analysis. In Elfen, interrupt function is run with a string parameter that enables identification of location where the function was called.

1.1.1.2 void * elfendyn_database_entry_new (const char * recname, int recnum, const char *arrayname, char *datatype, int *arraydim);

A general function for accessing analysis database.
Returns a pointer to the requested array (or NULL if pointer is not available).

Recname is record name, *recnum* is record number, and *arrayname* is array name. Function writes specification of the array type in **datatype* and dimensions of the array in *arraydim*. a*rraydim* must point to space allocated for at least 5 integers at function call. After the function call, the first zero element of *arraydim* indicates the end of dimensions.

Explanations:

Dimensions returned in *arraydim* are actual physical dimensions of the field pointed to by the pointer returned by the function. All pointers returned are zero-offset. Elements in Elfen arrays are stored in reverse order, i.e. according to FORTRAN-like conventions, which means for example that matrices are stored column by column, not line by line.

Remarks:

This function is not yet appropriate as a basis for all data-access functions. Elfen data-base structure features some peculiarities, which should be hidden by such function. From the development point of view, mechanisms for overcoming of these things must be provided by Elfen side, while the Inverse development can only use these mechanisms. In the opposite case it would not be possible to ensure consistent development of the interface.

Peculiarities:

Two dimensions are sometimes combined in one returned by **elfendyn_database_entry_new**, e.g. number of degrees of freedom times number of nodes are treated as one dimension at result fields of nodal displacements and nodal reactions.

Physical dimensions (of the space in memory) can differ from actual ones. Both for implementation of interface: physical dimensions are needed for correct positioning of array elements while actual dimensions are those in which user of the interface is only interested.

Some entries in arrays may not be sorted. User of the interface needs to access these items directly, identifying them by serial numbers (indices). To enable that, Elfen general interface function for accessing the database must in such cases provide an index list.

Important Remark:

Function **elfendyn_database_entry_new** should mainly remain as it is. The required corrections should be implemented by another general function for accessing Elfen database.

1.1.1.3 void ELFRUN(void);

Runs Elfen analysis, i.e. reading input data and evaluation part.

Remark:

In the future analysis should be divided into two parts, for each of it a special function would be called, namely reading of analysis input data and analysis. A function could be also added to run only a single increment of the analysis.

1.1.2 Inverse Functions Used By Elfen

Elfen at this stage only needs the function for invoking inverse.

1.1.2.1 void RUN_INVERSE(void)

Runs the optimisation shell Inverse. Analysis runs the shell before reading input data. When the shell is run, it interprets the command file

Analysis is then run on command from the shell. User also has possibility of running analysis alone without invoking the shell. In that case the analysis is run automatically, i.e. programme flow is exactly the same as is in stand-alone version of Elfen.

1.1.3 Important functions on Inverse side

1.1.3.1 void get_optimisation_shell_arguments_new (int *nargs, char ***argv)

Gives the shell information about shell-related command-line arguments. In **nargs* it writes the number of arguments passed to the shell and in **argv* it writes string command-line arguments.

Explanation:

This function is necessary because Elfen part of the common programme has the **main** function and therefore a direct access to command-line arguments. User tells Elfen to run optimisation shell by adding the **–opt** argument in the command-line when invoking Elfen. All arguments following this argument are passed to the shell (with –opt included) by the **get_optimisation_shell_arguments_new** function. Function extracts these arguments from g_argc and g_argv, which are global variables. For example, Elfen can be invoked for solving an optimization problem like this:

elfendyn test 16 –opt test.cm -d

in this case *elfendyn* is programme name, *test* and *16* are project name and size (required by older analyses), *-opt* is option instructing Elfen to run the shell, and *test.cm* and *–d* are options used by the shell. If argument *–opt* and subsequent arguments were not specified, the shell would not be invoked and only analysis would be run (automatically).

Remarks:

User can invoking only the shell without analysis (for test purposes) by specifying nonexistent project name as the first and *–opt* as the third argument.

1.2 Overview

The implemented interface enables the optimisation shell to successively run analyses and access analysis data at different stages.

Input data for analysis is defined in a standard way. Analysis can be run stand-alone or from optimisation shell, dependent on command-line arguments. Interaction between Elfen and the shell during analysis is event driven. At specific points analysis generates events which can be intercepted by the shell. In the shell command file user defines what happens when a specific event is intercepted. Each event is characterised by name. Events occur at standard places like after reading the input data, before analysis, after analysis, after each increment, and at a specific times during analysis. User can define via controls in the data file at which times events are generated.

1.3 Specification of Shell Interpreter Interface Functions

1.3.1 elf_analyse *{ }*

Runs Elfen analysis.

1.3.2 elfenblock *{ name1 { block1 } name2 { block2 } … }*

Defines blocks of code that are interpreted when specific events are encountered during Elfen analysis. String arguments *name1*, *name2*, etc. are names of events. *block1*, *block2*, etc. are code blocks that are interpreted when the corresponding events are encountered during analysis.

The **elfenblock** commands can be executed arbitrary times. If a block with a certain name has been defined before by the **elfenblock** command, new definition overrides the old one. If a block name is followed by an empty curly bracket without spaces (i.e. "{}"), the corresponding block is undefined.

A special name "any" defines a block, which is interpreted at every event. Another special name "default" defines a block, which is interpreted at every event for which the corresponding block is not defined.

When an event with a given name encounters during analysis, the block of code associated with the name of the event is interpreted first (or "default" block if this block is not defined), and then the "any" block is interpreted (if defined). If a

block with a specific name is not defined, nothing happens instead of interpretation of that block.

1.3.3 **simelfevent** *{ eventname }*

Simulates occurrence of Elfen event named *eventname* in the sense that the corresponding blocks of code are interpreted. This function can be used for testing purposes when setting a project.

1.3.4 **elf_adddim** *{ dim1 dim2 … }*

Dimension correction for accessing Elfen database. After execution of this function, dimensions *dim1*, *dim2*, etc. are appended to the dimensions returned by Elfen general interface function at every access to database via that function. This function was implemented for handling situations when Elfen function does not return all physical dimensions of the field.

To restore initial settings, where no additional dimensions are added, function should be called without any arguments.

1.3.5 **elf_splitdimfirst** *{ dim }*

Dimension correction for accessing Elfen database. After execution of this function, the first dimension returned by Elfen general interface function is split to *dim* and that dimension divided by *dim* at every access to the database using that Elfen function. *dim* is a numerical argument.

To restore initial settings, where no additional dimensions are split, function should be called without any arguments.

Example:

Let’s say that some array in Elfen database has dimensions [24,3,2]. After execution of splitdimfirst { 2 }, user of the shell will see that array as if it was of dimensions [2,12,3,2] every time he would use shell interface that use the basic Elfen interface function for accessing the database.

1.3.6 **elf_splitdimlast** *{ dim }*

Dimension correction for accessing Elfen database. After execution of this function, the last dimension returned by Elfen general interface function is split to *dim* and that dimension divided by *dim* at every access to the database using that Elfen function. *dim* is a numerical argument.

To restore initial settings, where no additional dimensions are split, function should be called without any arguments.

Example:

Let’s say that some array in Elfen database has dimensions [2,63]. After execution of splitdimfirst { 3 }, user of the shell will see that array as if it was of

dimensions [2,3,21] every time he would use shell interface that use the basic Elfen interface function for accessing the database.

1.3.7 elf_getdimi *{ recname recnum arrayname dimspec varname }*

Assigns a specific dimension of Elfen array specified by record name *recname*, record number *recnum* and array name *arrayname*, to calculator variable named *varname*.

d*imspec* specifies which dimension to assign. If *dimspec* is less than 0, 0 is assigned if the specified array does not exist and a non-zero value if it exists. If *dimspec* is zero, number of dimensions of the field is assigned to the calculator variable named *varname*. If it is greater than zero, the appropriate dimension is assigned or –1, if the number of array dimensions is lesser than *dimspec*.

1.3.8 elf_getdatai *{ recname recnum arrayname varname ind1 ind2 ind3 … }*

Assigns value of a specific element of Elfen's array to the calculator variable named *varname*. *recname, recnum* and *arrayname* identify the array while *ind1*, *ind2*, etc. are indices of the desired element in that array.

Warning:
Elfen array must be of the admitted type for this operation (i.e. real or integer).

1.3.9 elf_setdata *{ recname recnum arrayname val ind1 ind2 ind3 …}*

Sets value of a specific element of Elfen's array to value *val*. *recname, recnum* and *arrayname* identify the array while *ind1*, *ind2*, etc. are indices of the desired element in that array.

Warning:
Elfen array must be of the admitted type for this operation (i.e. real or integer).

1.3.10 elf_fprintelements *{ recname recnum arrayname }*

Prints all elements of the Elfen array identified by *recname, recnum* and *arrayname*, to the output file of the shell. Indices are printed in square brackets before the value of each element. Elements are printed in separate lines.

Warning:
Elfen array must be of the admitted type for this operation (i.e. real or integer).

1.3.11 elf_printelements *{ recname recnum arrayname }*

Does the same as **elf_fprintelements**, except that data is printed to the standard output of the programme.

1.3.12 elf_dprintelements *{ recname recnum arrayname }*

Does the same as **elf_fprintelements**, except that data is printed to both the standard output and output file of the shell.

1.3.13 elf_fprintelements2d *{ recname recnum arrayname }*

Prints all elements of the Elfen array identified by *recname*, *recnum* and *arrayname*, to the output file of the shell. Array is printed in a two-dimensional form where number of columns equals the last dimension of the field. Indices (except the last ones) are printed in square brackets at the beginning of each row of the print.

Warning:

Elfen array must be of the admitted type for this operation (i.e. real or integer).

1.3.14 elf_printelements2d *{ recname recnum arrayname }*

Does the same as **elf_fprintelements2d**, except that data is printed to the standard output of the programme.

1.3.15 elf_dprintelements2d *{ recname recnum arrayname }*

Does the same as **elf_fprintelements2d**, except that data is printed to both the standard output and output file of the shell.

1.3.16 elf_setcomponents *{ recname recnum arrayname expression }*, elf_setcomp

Sets all elements of Elfen array identified by *recname*, *recnum* and *arrayname*, to the value of mathematical expression *expression*.

Calculator functions **elf_ind** and **elf_dim** can be used in the expression. **elfi_ind** returns a specific index of the element in Elfen array that is currently set, and **elf_dim** returns a specific dimension of Elfen array. Both functions require one argument, which identifies the sequential number of the returned index or dimension, respectively.

Warning:

Elfen array must be of the admitted type for this operation (i.e. real or integer).

1.3.17 **elf_setcomponentscond** *{ recname recnum arrayname (expressioncond) expression },* **elf_setcompcond**

Sets all elements of Elfen array identified by *recname, recnum* and *arrayname*, for which conditional expression *expressioncond* is not zero, to the value of mathematical expression *expression*. Note that conditional expression must always be in round brackets.

Calculator functions **elf_ind** and **elf_dim** can be used in both expressions. **elfi_ind** returns a specific index of the element in Elfen array that is currently set, and **elf_dim** returns a specific dimension of Elfen array. Both functions require one argument, which identifies the sequential number of the returned index or dimension, respectively.

Warning:

Elfen array must be of the admitted type for this operation (i.e. real or integer).

1.4 Specification of Shell Expression Evaluator Interface Functions

1.4.1 **elf_getdimc** *[recname, recnum, arrayname, dimspec]*

Returns the specified dimension of Elfen array identified by *recname, recnum* and *arrayname.*

d*imspec* specifies which dimension is returned. If *dimspec* is less than 0, 0 is returned if the specified array does not exist and a non-zero value if it exists. If *dimspec* is zero, number of dimensions of the field is returned. If it is greater than zero, the appropriate dimension is returned or –1 if the number of array dimensions is lesser than *dimspec*.

1.4.2 **elf_getdatac** *[recname, recnum, arrayname, ind1, ind2, …]*

Returns value of a specific element of Elfen's array. *recname, recnum* and *arrayname* identify the array while *ind1, ind2*, etc. are indices of the desired element in that array.

Warnings:

Elfen array must be of the admitted type for this operation (i.e. real or integer).

Note that there exists interpreter function with the same name.

1.4.3 **elf_ind** *[indspec]*

Returns a specific index of the element in Elfen array that is currently being set by one of the functions for setting elements of Elfen arrays (e.g. **elf_setcomp**). The only argument *indspec* identifies which index is returned. If *indspec* is 0, number of indices is returned.

1.4.4 **elf_dim** *[dimspec]*

Returns a specific dimension of the Elfen array whose elements are currently being set by one of the functions for setting elements of Elfen arrays (e.g. **elf_setcomp**). The only argument *dimspec* identifies which dimension is returned. If *dimspec* is 0, number of dimensions is returned.

1.5 Specification of Test Evaluator and Interpreter Functions

Explanation:

For testing functions for accessing Elfen database, an internal shell array for emulation of Elfen array. Its specification is “test”, 0, “double” (record name, record number and array name) and it holds elements {1.1, 2.1, 1.2, 2.2, 1.3, 2.3} and a single related dimension (i.e. 6). All general functions for accessing Elfen database can be tested on this field. Also special functions for accessing this particular array are implemented in order to show how special functions for accessing individual arrays should be implemented in the future and how they should work. When reading the below specifications, note that interpreter functions have argument lists included in curly brackets and evaluator functions in square brackets.

Remark:

In the future, for all standard sets of data there should exist similar functions (e.g. for node coordinates, nodal displacements and reactions, mesh topology, material data, etc.)

1.5.1 **elf_settestdata** *{ val ind1 ind2 }*

Sets the element specified by indices *ind1* and *ind2* of the internal test array (i.e. the array identified by “test”, 0, ”double”), to value *val*.

1.5.2 elf_testdatadim *[dimspec]*

Returns the dimension specified by *dimspec* of the internal test array (i.e. the array identified by “test”, 0, ”double”).

d*imspec* specifies which dimension is returned. If *dimspec* is less than 0, 0 is returned if the specified array does not exist and a non-zero value if it exists. If *dimspec* is zero, number of dimensions of the field is returned. If it is greater than zero, the appropriate dimension is returned or –1 if the number of array dimensions is lesser than *dimspec*.

1.5.3 elf_testdata *[ind1, ind2]*

Returns the value of element specified by indices *ind1* and *ind2* of the internal test array (i.e. the array identified by “test”, 0, ”double”).

2. FURTHER WORK

2.1 Agreements & Plans

We have agreed to continue to develop interface and to work on commercialisation of the shell in the scope of a common software package.

A rough plan will be made bout further steps at the next meeting (probably in April). After that, Rockfield will undertake necessary activities for providing its part. When implementation of that is finished, we will organise the next working meeting. After that meeting, everything should be prepared for creation of commercial optimisation package. This includes completion of the current basic part of the interface and its upgrade, namely entity-based definitions. Plans for development of shape optimisation modules will be also defined at this stage, but final implementation will depend on the implementation of 2D and 3D shape optimisation modules in the shell and Elfen.

2.2 Questions to Clarify

2.2.1 It is possible to run only analysis (if only Elfen's arguments are specified) or only shell (if analysis is not run from command file).Elfen releases and interface with Inverse

To enable normal development work, it would be the best if interface functions are available in all Elfen libraries. This will also make work in Rockfield easier because no special versions of Elfen will be needed for linking with Inverse. When a few problems are solved (i.e. that Elfen analysis will run stand-alone when optimisation argument –opt is not stated among command-line arguments) this should not be disturbing at all. Note there are not only a few short interface functions and the only one that really changes something in Elfen is the one for running analysis.

2.2.2 Elfen – Inverse demo for PC

It would be good to make demo version available.

2.2.3 Demonstrative optimisation examples

These examples should show applicability of the common programme and attract users.

2.2.4 Test examples for the interface

Test examples should include use of all interface functionality. If specification of any interface function would change, this should be detected by running test examples.

Beside for control of the interface, test examples should be used as a reference list of approaches used for solution of different types of problems (e.g. inverse material parameter identification or shape optimisation). Therefore it would be good if all test examples can be run by demo version of Elfen.

2.2.5 Machine dependency of conversion of Elfen types

The problem of converting Elfen types to C types is not solved yet. We would need reliable information about rules of conversion on different platforms. Currently it is working on PC when VISUAL C++ compiler for C and Compaq Visual Fortran compiler for Fortran is used.

2.2.6 Availability of the necessary stuff for project examples

Shape derivatives defined on mesh or on continuous domain; how to get max. values for divided flow; availability of functionals…

2.2.7 Question of finance

Resources for shell development; Rockfield resources for development of the interface and commercial package; Taking care for ability of getting European research projects (and related current problems); Exploitation plan, Rockfield – C3M relation, definition of target area (e.g. forging)… "How to make things work" versus "How to sell things"…

2.2.8 Plan

Dates – when to make further steps and which steps; Make further interactions effective; Employ experience with research and experimental codes (Sava; Tomaz S.); Employ students to work on it?

2.2.9 Problems on Shell Side Exposed by Rockfield

- Poor human potential
- Introduction of code management system necessary (CVS)
- Warnings generated by compiler
- Use of commercial FSQP

2.2.10 Possible Problems in the future

- Thermo-mechanical analysis (how it will be run?) It should be run from one file, so 2 databases are used. Problem is because remeshing could not be used in this case (it is not developed yet).

2.2.11 Remarks

- File Interpreter function exit{} terminates just Inverse

3. SOURCE FILES

ELFEN		**INVERSE**	
File	*Description*	*File*	*Description*
Added files		**Added files**	
avgeti.f (avget.f)	subrutine avgeti	invelf.c	everything that is needed for the interface
elfendyn_database_entry.c (database_entry.c)	function elfendyn_database_entry_new		
elfrun.fpp (elfen.fpp)	copy of elfen.fpp + some changes ??		
nfckrci.f (nfckrc.f)	subrutine nfckrci		
rcdefi.f (rcdef.f)	subrutine rcdefi		
rcincdi.f (rcincd.f)	subrutine rcincdi		
record_info.c	functionrecord_info (transforms long record and array names into short ones)		
Changed files *			
File	*Description*		
elfen.fpp	calls RUN_INVERSE		
immain.f	several calls of interrupt routine OPTSHL are added		
imnonl.f	several calls of interrupt routine OPTSHL can be added		

* changes in Elfen are commented with string: Inverse_specific:

4. COMPILER SPECIFIC

From:	Matthew Ellis <m.t.ellis@swansea.ac.uk>	Date:	10 March 2000

By default, C programs compiled with Visual C++ use a different calling convention to Fortran programs compiled with Digital/Compaq Visual Fortran. The calling conventions define who is responsible for cleaning the stack, and how the name is decorated (the name it is internally assigned during compilation and linking). If different calling conventions are used, the executable either won't link, due to the different name decoration, or will crash on execution due to the stack not being cleared correctly.

Using the default settings, C uses __cdecl and Fortran uses __stdcall. Previous versions of elfendyn used INTERFACE statements to change the calling type and specify the name decoration. The INTERFACE statement is not valid across all compilers, and as it is far easier to pre-process C code than Fortran, the calling convention of the C code was changed and the INTERFACE statements removed.

The following details everything that needs to be done to set the correct calling conventions:

- All C functions called by Fortran must use the same case name as the Fortran uses to call the function. Elfen uses capitals for all Fortran subroutine names and for all C function names called by Fortran.
- All C functions called by Fortran must be defined and declared with the keyword __stdcall. E.g.
  int __stdcall CSPAWN(…)
- All Fortran functions called by C must be declared with the keyword __stdcall. E.g.
  extern void __stdcall ELFEN(…)

Note that the __stdcall keyword is Visual C specific. If the C code is to be portable, then macros must be used to specify the calling convention keywords to use, the case to use and whether or not to add underscores to the start or end of the function name. Rockfield can supply a copy of the macros used in-house upon request.