ViennaIPD - An Input Control Language for Scientific Computing

Josef Weinbub
Karl Rupp
Siegfried Selberherr

Institute for Microelectronics, Technische Universität Wien
Gußhausstraße 27-29 / E360
A-1040 Vienna, Austria
E-mail: {weinbub|rupp|selberherr}@iue.tuwien.ac.at

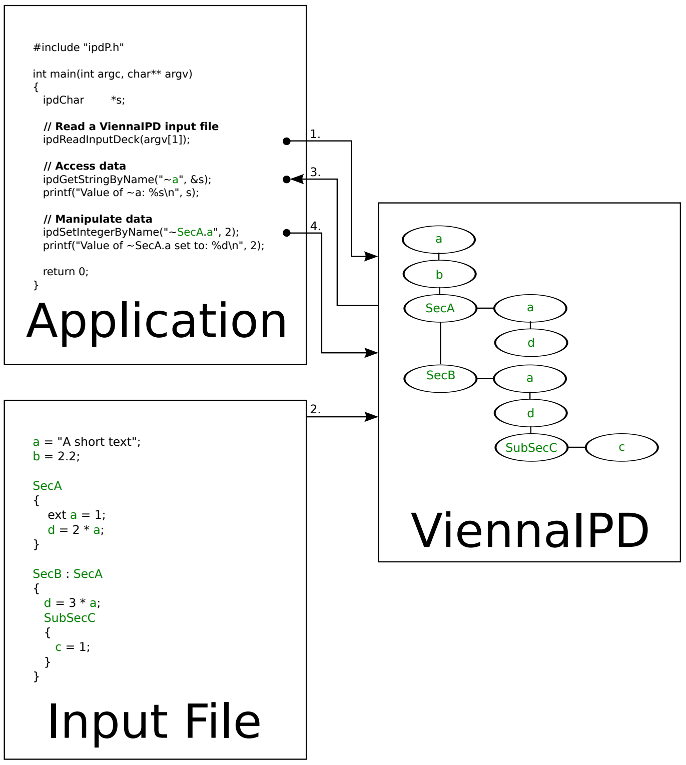
ABSTRACT

A powerful control language, named ViennaIPD, has been developed. The library based software is capable of reading complex input datasets which can be accessed from C/C++ applications by a programming interface. ViennaIPD supports a convenient C-like input file language, object oriented structuring of datasets, powerful inheritance mechanisms and a unit system. The input file language as well as the programming interface is discussed in detail.

INTRODUCTION

Applications for scientific computing require a powerful control language to satisfy the need of control parameters, e.g. material properties, models to take into account, model parameters, process definitions, simulation modes, iteration schemes, and numerical behavior [1, 2, 3, 4]. As a result, control files grow in size, which results in decreased maintainability. Therefore a powerful control language has been developed as part of the MINIMOS-NT simulation software package [5, 6]. To enlarge the field of application the control language has been extracted, revisited, and packaged to be distributable as a standalone control language named ViennaIPD. ViennaIPD offers a powerful inheritance concept, unit conversion capabilities and the ability to directly access existing input files. The latter is especially of interest for reasons of post-processing simulation results. Figure 1 depicts a comprehensible overview how ViennaIPD is used.

ViennaIPD provides a powerful input file language which aims to deal with the manifold parameter demands of simulation software. The library itself is written in C, however, applications can be based on either C or C++. The most fundamental element is a *variable*. To describe dependencies to other variables, a variable contains an expression, that is evaluated at runtime. Different variable types are supported as well as accessibility control. Furthermore, a *section* provides a means of structuring variables to enable associative variable definitions. A section holds an arbitrary set of variables and subsections.



```
#include "ipdP.h"

int main(int argc, char** argv)
{
  ipdChar       *s;

  // Read a ViennaIPD input file
  ipdReadInputDeck(argv[1]);

  // Access data
  ipdGetStringByName("~a", &s);
  printf("Value of ~a: %s\n", s);

  // Manipulate data
  ipdSetIntegerByName("~SecA.a", 2);
  printf("Value of ~SecA.a set to: %d\n", 2);

  return 0;
}
```

```
a = "A short text";
b = 2.2;

SecA
{
   ext a = 1;
   d = 2 * a;
}

SecB : SecA
{
  d = 3 * a;
  SubSecC
  {
    c = 1;
  }
}
```

Figure 1: ViennaIPD workflow. The application passes the filename of the input file to ViennaIPD (**1.**). The input file is used to populate the tree based datastructure (**2.**). The data can be accessed (**3.**) and manipulated (**4.**) by the application.

Therefore any kind of data set hierarchy can be built. In this context, inheritance of sections is supported to enable, for example, reusability of section elements. Finally a large set of functions is provided to enable mathematical computations within the input files.
The paper is organized as follows: The first section provides an overview of the ViennaIPD control language, whereas the second section outlines the programming interface. Furthermore, the third section depicts a use case example. Finally, the fourth section compares ViennaIPD to various other input control approaches.

INPUT FILE SPECIFICATION

The input file specification is based on a comprehensible language. The usage is similar to common programming languages, e.g. C. Additionally, object oriented features are provided to enable structuring of large data sets, i.e. inheritance mechanisms. Furthermore, a powerful unit system is provided which supports the validity of computations based on quantities.

Variables

An arbitrary number of variables can be defined. The general definition of a variable is

```
<type> <variable_name> = <expression>;
```

`<type>` must be a valid variable type. The available variable types are listed in Table 1. Note that if no `<type>` is mentioned, the default key type is used. `<variable name>` can be of any length, can contain capital and small letters, an underscore, and numbers.

<type>	Description
ext	external variable - read- and writeable
key	keyword variable - read-only (default)
aux	auxiliary variable - hidden

Table 1: Types of Variables

However, the name must not begin with a number. An expression can be assigned to a variable by using the assignment operator `=` and a valid data-type (Table 2).

Type	Example
Boolean	`a = true;`
Integer	`a = 3;`
Real	`a = 3.1415;`
Complex	`a = 4.3 + 3.1 j;`
Quantity	`a = 3.1415 "m/s";`
String	`a = "This is a string";`
Array	`a = [1, "pi", 3 A];`

Table 2: Types of Data

The following snippet depicts a few examples which make use of the available variable types.

```
1       some_float = 3.3;
2   key some_text  = "some Text";
3   ext x          = 32;
4   ext y          = 4;
5   aux a          = sqrt(x * x + y * y);
6       ex         = x / a;
7       ey         = y / a;
```

Lines 1,2,6,7 depict keyword variables. Hence, the assigned data can be accessed only, not altered. Lines 3,4 depict external variables, for which data can be accessed and altered. Line 5 outlines an application case for the auxiliary variable. This variable cannot be accessed by the programming interface, it is hidden.

However, this variable type can be used to temporarily store data, for example the result of a computation.

Sections

ViennaIPD supports structures of variables, so called sections, to manage large input files. A section is defined with

```
<section name> { ... }
```

The following snippet depicts a section named `Solve` with a read-only keyword variable.

```
Solve { maxStepSize = 1e30; }
```

Sections can be nested to arbitrary depth

```
BaseSection {
   SubSection {
       SubSubSection { }
   }
}
```

ViennaIPD enables also inheritance of sections.

```
BaseSection { x1 = -3; }
DerivedSection : BaseSection;
```

`DerivedSection` is inherited from `BaseSection`. Consequently, `DerivedSection` contains all elements (variables and subsections) of `BaseSection`.
Note, that the variables of `BaseSection` *transparently* exist in `DerivedSection`. Hence, if a variable in `BaseSection` changes, it changes in `DerivedSection` too.
Furthermore, multiple inheritance is supported.

```
BaseSection1 { a = 1; }
BaseSection2 { b = 3; }
DerivedSection : BaseSection1, BaseSection2;
```

`DerivedSection` consists of two variables `a` and `b` inherited from `BaseSection1` and `BaseSection2`, respectively.
Additionally, conditional inheritance enables to inherit from sections based on specific conditions.

```
Section : BaseSection1 ? expression1,
          BaseSection1 ? expression2,
          ... { }
```

Conditional inheritance is an extension to multiple inheritance. Several base sections can be specified for inheritance, an expression can be assigned to each one. The following snippet depicts the usage.

```
Switch = 0;
DerivedSection : BaseSection1,
                 BaseSection2 ? (Switch > -1),
                 BaseSection3 ? (Switch <  1)
{ }
```

`DerivedSection` is always inherited from `BaseSection1`. As `Switch` is 0, the conditions for `BaseSection2` and `BaseSection3` both hold.

Therefore, `DerivedSection` is additionally inherited from `BaseSection2` and `BaseSection3`. If `Switch` is set to 2, `DerivedSection` is solely inherited from `BaseSection1` and `BaseSection2`.
Elements of inherited sections can be locally modified.

```
BaseSection {
   x = 1;
   y = 2;
}
DerivedSection : BaseSection { y = 4; }
```

`DerivedSection` consists of the two variables `x` and `y`, where the variable `x` is inherited from `BaseSection` and the variable `y` is locally modified. So the variable `DerivedSection.x` equals 1 and the variable `DerivedSection.y` equals 4.

Functions

ViennaIPD enables the use of functions within input files. Functions can be used in expressions which are stored in variables.

```
<type> <variable_name> = <function>(parameters);
```

An arbitrary number of parameters is supported.

```
a1 = func1();               // 0 Parameter
a2 = func2(1 + 2);          // 1 Parameter
a3 = func3(”hello”, a1);    // 2 Parameter
```

ViennaIPD provides built-in functions, for example a large set of mathematical functions, e.g. `sin()`, `pow()`, `sqrt()`, to list a few of them. The following snippet depicts the computation of the sine of a complex number.

```
a = sin(4.42 + j*5.9);
```

Furthermore support functions are provided, e.g. `if()` statement, conversion functions, array sizes, random number generator, to name a few.

Units

ViennaIPD provides a powerful unit system. A unit is used in combination with the *quantity* datatype (Table 2) and consists of a valid unit name, of which many are provided. Quantity examples are:

```
1 q1 = 3.0 A;
2 q2 = 5.5 ”V * A”;
3 q3 = 4.2 ”kg m/s s”;
```

Note that if a quantity is based on several unit names, double quotes have to be used (Lines 2,3). Unit names can be separated by a whitespace or an asterisk `*` (Line 2). The use of a single slash `/` indicates the beginning of the denominator. Hence, Line 3 represents the unit $kg\ m/s^2$. Quantity based computations are checked for semantically correct operations. Therefore, an error occurs, if two quantities with different units are summed up, as this would result in a mixed up physical quantity and, therefore, looses its validity. The following snippet outlines this behavior.

```
Q1 = 3 ”A”;
Q2 = 2 ”m”;
Q3 = Q1 * Q2;   // OK
Q4 = Q1 + Q2;   // Error
```

PROGRAMMING INTERFACE

The ViennaIPD Programming Interface is based on the C programming language. It provides an interface to the internal tree-based datastructures of ViennaIPD. Data can be traversed and accessed.

Interface Setup

To be able to use the interface the ViennaIPD header file has to be included.

```
#include ”ipd.h”
```

The following code snippet depicts the initialization and the population of the datastructure from an input file.

```
// basic initialization
ipdInit(NULL, NULL);
// create a new database
ipdCreateBase(”NameOfTheDatabase”, 0);
// read the inputfile
ipdReadInputDeck(argv[1]);
```

Accessing Data

Specialized getter and setter functions are provided to access and alter specific data types. The following code snippet depicts the reading and writing process based on these functions.

```
// declare a variable of type long
ipdLong i;
// get the "c" integer value
ipdGetIntegerByName(”~c”, &i);
// set the "c" integer value
ipdSetIntegerByName(”~c”, 2);
```

Note the ∼ prefix, which indicates the root level of the datastructure hierarchy. Therefore variables can be accessed by absolute path addressing. For example accessing an integer variable named `a` within a section named `BaseSection` would be implemented by the following:

```
// declare a variable of type long
ipdLong i;
// get the "BaseSection" member value "a"
ipdGetIntegerByName(”~BaseSection.a”, &i);
```

Data can also be accessed without absolute addressing. This approach can be utilized by setting the current section. In relation to the previous example, the current section can be set to `BaseSection` as follows:

```
ipdSetCSByName(”~BaseSection”);
```

Consequently, the data can be accessed directly

```
// get the integer data associated with the "a"
// variable in the current section
ipdGetIntegerByName(”a”, &i);
```

Iterators

ViennaIPD provides iterators for convenient traversal of datastructures. The following snippet creates a new iterator on the root level

```
// define a new iterator
ipdIterator_t   *iNode = NULL;
// set the iterator to traverse the root level
ipdIteratorNewAtRootSection(&iNode, ipdANY);
```

All elements, e.g. variables, sections, on the root level can be traversed.

```
// traverse as long as there are elements on
// the root level
while (ipdIteratorIsValid(iNode)) {
   // retrieve the name of the current element
   ipdConstString itemName =
      ipdIteratorGetItemName(iNode);
   // is the element a variable?
   if (ipdIteratorGetType(iNode) == ipdVARIABLE)
      printf("variable: %s\n", itemName);
   // is the element a section?
   else
   if (ipdIteratorGetType(iNode) == ipdSECTION)
      printf("section: %s\n", itemName);
   // increment the iterator to point
   // to the next element
   ipdIteratorDoNext(iNode);
}
```

USE CASE EXAMPLE

The following depicts an exemplary ViennaIPD input file which contains a small dataset.

```
1  Electrons {
2     uL300     = 0.143    "m^2/V*s";
3     uLImin300 = 0.008    "m^2/V*s";
4     Cref300   = 1.12e17  "cm^-3";
5     CrefexpT  = 3.2;
6  }
7  Holes : Electrons {
8     uL300     = 460      "cm^2/V*s";
9     uLImin300 = 45       "cm^2/V*s";
10    Cref300   = 2.23e17  "cm^-3";
11 }
```

Note the different units for the same parameters in Lines 2,3,8,9. The unit system automatically converts the units into SI units, hence, the quantities presented in Lines 8,9 are internally converted to: $0.046 m^2/s \cdot V$ and $0.0045 m^2/s \cdot V$, respectively. Further note, that Electrons and Holes have different quantities for the $Cref300$ parameter, which too get converted (Lines 4,10). Due to inheritance, both sections use the same value for the $CrefexpT$ parameter (Line 5). The following depicts the access of some data of the Holes section.

```
double ul, uLImin;
char*  unit;
ipdGetRealQuantityByName("~Holes.uL300",
   &ul, &unit);
ipdGetRealQuantityByName("~Holes.uLImin300",
   &uLImin, &unit);
std::cout << ul     << " " << unit << std::endl;
std::cout << uLImin << " " << unit << std::endl;
```

COMPARISON

A comparison of different input control software packages is presented next. The comparison is based on the following features.

- *convenient scripting language:* The input language should be intuitive to scientists with slight background in programming. This results in an easy familiarization which, furthermore, reduces the overall implementation effort.
- *unit system:* Typically scientific computing relies on physical quantities. Hence, it is of major interest to provide a computation facility which respects units and therefore enables a correct setup of physical quantities.
- *memory consumption:* The memory consumption, for example the peak heap memory usage, should be kept small.
- *execution performance:* The input control software should not decrease the overall execution performance of the scientific application.

ViennaIPD is compared to the common scripting languages, Lua [7], AngelScript [8], and Python [9] in the following. All of the introduced software packages are supported with a convenient and intuitive language.

Lua is a so called extension programming language which extends the functionality of mature programming languages, like C. Lua is a minimalistic, C based library which solely exists embedded in host applications. Lua can be used by different programming languages, e.g. C, C++, Fortran.
AngelScript is a flexible cross-platform scripting library. C and C++ functions can be called within an AngelScript environment, hence code can be reused efficiently.
Python is a mature programming language, but is also used as a scripting language. Python offers a C-API, and a C++ library for Python interoperability is also available [10].
All these input control languages offer interoperability with C and C++. Note, that most of the available programming languages, for example Fortran, are able to interoperate with C.
In the following, benchmark results are presented and discussed. The test platform is a PC with an AMD Phenom II X4 - 965 CPU and 8GB of RAM. The operating system is a Gentoo Linux 64-bit with a 2.6.32 kernel.

Table 3 depicts a comparison of the peak memory consumption and the execution time of a reference application which is driven by the introduced control languages. To depict the overhead of the input control languages, the standalone application, without any input control, is investigated too. The memory usage evaluation is based on Valgrind [11].

Software	*Peak Heap Memory* [KB]	*Execution Time* [ms]
No Input Control	0.8	6
Lua	32	18
AngelScript	34	162
Python	2132	627
ViennaIPD	543	32

Table 3: Comparison of peak memory consumption and execution time.

The following table compares the features of the introduced input control languages.

Software	*Units*	*Mem*	*Exec*
Lua	-	++	++
AngelScript	-	++	+
Python	++	o	o
ViennaIPD	++	+	++

Table 4: Feature Comparison of input control languages. Feature status: ++ excellent, + good, o fulfilled, - missing

Consequently, ViennaIPD is especially attractive for applications which need a robust, fast, and light-weight unit system at the input file level.

CONCLUSION

A powerful control language for scientific computing has been introduced. The key features are

- C-like language for input files
- arbitrarily nested expressions
- powerful inheritance mechanisms
- unit system
- C/C++ programming interface

The application usage has been investigated in depth. Furthermore, the input file specification and the programming interface have been discussed in detail. Input file snippets as well as code snippets for the programming interface have been presented to outline the usage. ViennaIPD is part of the Viennese TCAD simulation tools which are provided by the Institute for Microelectronics [12].

REFERENCES

[1] D. Meeker, "Finite Element Method Magnetics - User's Manual," 2007. [Online]. Available: http://www.femm.info/wiki/HomePage

[2] A. Olabi and A. Grunwald, "Computation of magnetic field in an actuator," *Simulation Modelling Practice and Theory*, vol. 16, no. 10, pp. 1728 – 1736, 2008.

[3] J. Iniguez and V. Raposo, "Numerical simulation of a simple low-speed model for an electrodynamic levitation system based on a halbach magnet array," *Journal of Magnetism and Magnetic Materials*, vol. 322, no. 9-12, pp. 1673 – 1676, 2010.

[4] M. Meinel, "FlowSimulator: A Python-controlled Approach to Unify Future CFD Simulation Workflows," in *EuroSciPy 2009*, 2009.

[5] S. Wagner *et al.*, "MINIMOS-NT User's Guide," Institute for Microelectronics, Technische Universität Wien.

[6] R. Klima, "Three-Dimensional Device Simulation with Minimos-NT," Dissertation, Technische Universität Wien, November 2002.

[7] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes, "Lua 5.1 Reference Manual." [Online]. Available: http://www.lua.org

[8] "AngelScript Manual." [Online]. Available: http://www.angelcode.com/angelscript/

[9] "Python Documentation." [Online]. Available: http://www.python.org

[10] "Boost Python C++ Library." [Online]. Available: http://www.boost.org

[11] "Valgrind." [Online]. Available: http://valgrind.org/

[12] "Institute for Microelectronics, Technische Universität Wien." [Online]. Available: http://www.iue.tuwien.ac.at

BIOGRAPHY

JOSEF WEINBUB received the BSc degree in electrical engineering and the degree of "Diplomingenieur" in microelectronics from the "Technische Universität Wien" in 2009. He is currently working on his doctoral degree, where his scientific interests include modern programming techniques for scientific computing, high-performance computing, computational topology and adaptive mesh generation.

KARL RUPP received the BEng degree in electrical engineering from the "Technische Universität Wien" in 2006, the MSc in computational mathematics from Brunel University in 2007, and the degree of "Diplomingenieur" in microelectronics and in technical mathematics from the "Technische Universität Wien" in 2009. He is currently working on his doctoral degree, where his scientific interests include generative programming of discretization schemes such as the finite element method for the use in multiphysics problems.

SIEGFRIED SELBERHERR was born in Austria in 1955. He received the degree of "Diplomingenieur" in electrical engineering and the doctoral degree in technical sciences from the "Technische Universität Wien" in 1978 and 1981, respectively. Dr. Selberherr has been holding the "venia docendi" on computer-aided design since 1984. Since 1988 he has been the chair professor of the Institute for Microelectronics. From 1998 to 2005 he served as dean of the Faculty for Electrical Engineering and Information Technology. His current research interests are modeling and simulation of problems for microelectronics engineering.