Proceedings of the IASTED International Conference
SOFTWARE ENGINEERING AND APPLICATIONS
November 6-9, 2000, Las Vegas, Nevada, USA

CONTROLLING TCAD APPLICATIONS WITH A DYNAMIC DATABASE

R. Klima, T. Grasser, T. Binder, and S. Selberherr

Institute for Microelectronics, TU Vienna,
Gusshausstrasse 27–29, A-1040 Vienna, Austria
Phone: +43/1/58801-36030, Fax: +43/1/58801-36099,
E-mail: Robert.Klima@iue.tuwien.ac.at

ABSTRACT

State-of-the-art TCAD applications like device and process simulators require a huge number of parameters to control the various simulation modes and models. Simple input deck files which are just a sequence of static keywords or the use of the command line are no longer sufficient. Since the values and the usage of these parameters depend on each other it is mandatory to utilize mathematical expressions and hierarchical dependencies. We present an approach which relies on an object-oriented dynamic database to control the simulators. The database provides keywords grouped in sections which can be nested and inherited to build arbitrary hierarchies. For optimal control over the simulation, the contents of the database can be dynamically adapted using runtime information provided by the simulators like update norms, iteration counters and related information.

Keywords: Database systems, object-oriented mechanisms, application control, simulation.

1 INTRODUCTION

TCAD applications like the device-simulator MINIMOS-NT [1] require a wide range of information to perform a simulation. This information controls, e.g., input and output file handling, simulation modes, models, model parameters, iteration schemes, quantities to compute, or input and output curves to step through.

Generally speaking, a simulator needs a lot of information depending on the problem to solve, on the device, or on the decisions made. This information can be given by providing sets of parameters which may also depend on other parameters. Conventional simulators store all possible information either in a static input file or hard coded in the simulator invisible to the user. This makes it difficult for the user to modify this information or to describe new dependencies. Mostly, the user may specify parameters with values which override the simulator default settings.

Aim of the present work was to develop a standardized Input Deck database and a standardized control and description language which meets the requirements of controlling state-of-the-art TCAD applications and simulation environments. The Input Deck database is implemented as an object-oriented dynamic database which replaces a prior static input language [2]. All information are read from input files written in a new programming language called Input Deck programming language (IPL).

Requirements for such an Input Deck are:

- Collection of keywords using equations to describe dependencies.
- Storing default settings.
- An Input Deck should act as a database which can be queried and easily modified by the simulator.
- When the value of a keyword is changed the values of depending variables have to be recalculated simultaneously.
- Application specific functions should consistently enlarge the concept.
- Simple syntax of the programming language.

As examples we present the control mechanism of MINIMOS-NT and of the three-dimensional device generator MKWAFER.

2 CONTROL LANGUAGE

The Input Deck database provides a toolkit which contains a reader and a writer module to read and store all informations in files using its own Input Deck programming language (IPL). The reader module has been implemented using the tools GNU Flex [3] and GNU Bison [4]. The syntax of the IPL is similar to that of the C++ programming language [5, 6]. Several typical features of programming languages are supported, among others, `if()` statements, function definitions, etc. IPL is a description language containing mathematical expressions.

IPL files are text files which are parsed from top down. IPL distinguishes between two kinds of so called base items: variables and sections. Base items describe the entries to the database. With these variables and sections arbitrary database structures and hierarchies can be defined. Base items must be defined before they can be referenced. This enables type checking of items used in derived sections and avoids circular references.

2.1 VARIABLES

A variable is a named container holding an expression. An expression can be either a value, a list of values, a complex formula or – consequently – a list of formulas respectively (see Section 3.4).

Variables have different access types specified by modifiers. First, variables visible to the application are called *keywords*. Variables with no modifier are read-only keywords by default. The application may read them but writing is not permitted. Each write attempt causes an error message to be printed. Thus, unintentional access to read-only data can be recognized by the application developers. Secondly, variables defined with the `ext` modifier are keywords which may be read and written by the application. Thereby, the application can alter variables in the Input Deck database to return runtime information. Thirdly, auxiliary variables specified by the `aux` modifier are invisible to the application. Auxiliary variables can be used for auxiliary calculations or to simplify complex formulas as shown in the following example.

```
ext x = 32;
ext y = 4;
aux r = sqrt(x * x + y * y);
ex    = x / r;
ey    = y / r;
```

The variables `x`, `y`, `ex`, and `ey` denote keywords. The variables `x` and `y` describe the components of a vector defined modifiable to the application. Note that variable `r` is defined as an auxiliary variable which calculates the length of the vector. It is used to calculate the normalized vector components `ex` and `ey`. The keywords `ex` and `ey` are read-only.

The data type of a variable is determined implicitly. Variables can hold values or mathematical formulas resulting in values. A value can be a single value or a compound of values. There are different data types for values: integer numbers, real numbers, complex numbers, real or complex numbers with a unit (quantities), boolean values, and strings. The only compound data type supported is an array of values. Since an array of values is defined again to be a value, an array of arrays is also a valid data type. Arrays are useful for storing sets of data, e.g., a curve or coordinates of points.

```
integer_value  = 3;
real_value     = 3.14;
complex_value  = 3.4 + 5.5j;
quantity_value = 1.2 V;
string_value   = "a short text"
boolean_value  = true;
array_value    = [ 2, 3, 5, 7, 11 ];
```

2.2 SECTIONS

Sections are named containers holding an arbitrary number of variables and other sections – so called subsections. Subsections again can hold an arbitrary number of sections and variables. This forms a tree with sections representing the branches and variables and empty sections being the leaves. The overall section holding all items defined by the user – the so called *root section* denoted by the tilde ('˜') – represents the root of the tree. Since the root section is created and opened before reading the first line of the IPL file all sections specified are subsections of the root section. A section is defined by an identifier and the section body enclosed in braces.

```
MySection {
   ...
   myvariable = 1;
   MySubSection {
      ...
   }
}
```

Similarly to variables, all sections are defined read-only by default. Variables defined writable within such read-only sections using the `ext` modifier cause a warning to be printed at parsetime. Sections defined with the `aux` modifiers are invisible to the application, although, variables with any modifier are allowed inside. Such sections are useful to build default sections. Sections defined with the `ext` modifier may again contain variables defined with any modifiers. These sections are used to collect variables defined with the `ext` modifier.

3 DATABASE DESIGN

The Input Deck database acts differently compared to common databases [7, 8] as it has been developed to control complex TCAD simulators. It combines the capabilities of an object-oriented database using a complex inheritance scheme and that of a calculation program by using formulas which may contain references to other variables. The design of this Input Deck database is discussed in this section.

3.1 NAME MAPPING

References to other items form the crucial part of the Input Deck database. A critical understanding and implementation of these references is mandatory to guarantee optimal performance. The Input Deck database uses two different kinds of references. First, a variable reference is the name of a variable used in an expression (see Section 3.4). Secondly, a section reference is the name of a section used in aliases or for specifying base-sections when defining the inheritance (see Section 3.2). In the IPL references can be specified either by the fullname or the relative name of the item.

A fullname is a chain of all section names containing the absolute path to the item starting with the '˜' character for the root section and ending with the name of the item itself. A relative name is built up like a fullname but without using the '˜'. The 'ˆ' character can be used to denote the next upper section in the hierarchy.

```
Outer {
   one = 1;
   Inner {
      two = 2;
   }
}
```

In this small example the section `Outer` contains the keyword `one` and the subsection `Inner` which in turn contains a single keyword `two`. Within section `Inner` the keyword `one` can be referred to by `ˆ.one`, within section `Outer` by `one`, within the root section by `Outer.one`, and anywhere by its fullname `˜.Outer.one`. Within the subsection `Inner` the keyword `two` can be referred to by `two`, within section `Outer` by `Inner.two`, within the root section by `Outer.Innner.two`, and anywhere by its fullname `˜.Outer.Inner.two`

As references are normally used many times in expressions and inheritance trees, virtually thousands of items have to be created. Finding a corresponding item quickly from a reference specified by its name is a challenging task. As explained in Section 3.3 we use a dynamic approach of data representation – the corresponding items are determined at runtime. Therefore, references cannot be represented internally by nodes holding simple pointers to the item. Instead, the name specified must be used.

As shown in Section 3.2 finding but also referring to an item causes the inheritance algorithm to successively generate a sequence of possible item fullnames which are tested for their existence. Storing all path names as strings would slow down the inheritance algorithm. Therefore, we have implemented a different and much faster approach for storing names: Names are split into subnames which are section names or variable names respectively. Each subname is assigned a unique ID number. The '˜' has always ID 0, the 'ˆ' has ID 1. So each name is internally represented by a series of IDs, -1 indicating the end of a chain. E.g., if the subnames `Outer`, `Inner`, `one`, and `two` of the example above are represented by the IDs 2, 3, 4, 5 yields the following name – ID chain pairs result:

Reference	ID chain
`˜.Outer.Inner.two`	$0, 2, 3, 5, -1$
`Inner.two`	$3, 5, -1$
`ˆ.one`	$1, 4, -1$

Compared to references implemented as strings this approach speeds up the inheritance algorithm by a factor of two for our typical applications.

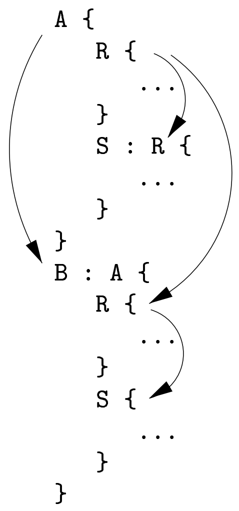
3.2 INHERITANCE SCHEME

The Input Deck database uses a powerful inheritance mechanism to pass existing definitions to sections. As inheritance is only allowed for sections, this mechanism is well suited to pass a complex tree of subsections, commands and keywords to a new section building a default structure which can be specialized later on. In the example below we inherit the whole structure of section A by section B:

```
A {
   ...
}
B : A;
```

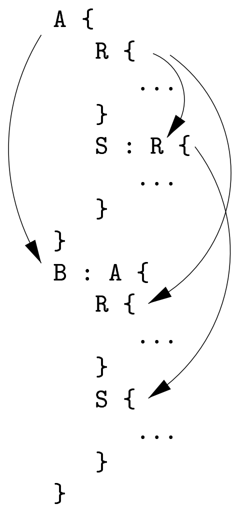
When items are locally modified several investigations take place. The Input Deck toolkit provides a flexible checking mechanism while reading and parsing the input or when a new item is created at runtime. All entries of a base section are assumed to be spelled correctly. A derived section may only override entries of its base section. Name and type of the entries must match exactly or an error message is printed. Thereby type and spell checking of known entries is enabled. Additional entries can be added explicitly by a preceding '+'. The checking mechanism can be disabled for a whole section by enclosing the section name in angle brackets. This is usefull when many or only new items are added to a derived section.

```
vehicle {
   wheels        = ...;
   steering      = ...;
   seat          = ...;
}
car : vehicle {        // full checking
   wheels        = ...;  // OK
   stering       = ...;  // error
  +engine        = ...;  // OK
}
<trailer> : vehicle {  // no checking
   loading_space = ...;  // OK
}
```



```
A {
   R {
      ...
   }
   S : R {
      ...
   }
}
B : A {
   R {
      ...
   }
   S {
      ...
   }
}
```

Figure 1: Implicit inheritance scheme



```
A {
   R {
      ...
   }
   S : R {
      ...
   }
}
B : A {
   R {
      ...
   }
   S {
      ...
   }
}
```

Figure 2: Explicit inheritance scheme

Sometimes it is useful to combine attributes of two sections to form a single section. The Input Deck database allows for multiple inheritance from any number of base sections. The base sections must be given in a list in descending order. This means that items in the first base section override items with the same name in the second base section.

```
amphibian : vehicle, ship {
   ...
}
```

Another powerful feature is what we call conditional inheritance. If the inheritance depends on several conditions multiple inheritance can be used to combine the base sections whereas each base section can be given a single condition. When a condition evaluates to true the corresponding base section is passed on to the inherited section. If no condition is given true is assumed. All evaluations are done at parse time to allow checking.

```
transport : car   ? (load <  loadmax),
            truck ? (load >= loadmax) {
   ...
}
```

In this special case the conditions are disjunct. Using aliases the section transport can be defined as

```
ct = if (load < loadmax, car, truck);
transport : ct
   ...
}
```

The inheritance mechanism of the Input Deck database allows for inheritance of every section, subsection and so on. Such an inheritance scheme has to be well defined or ambiguities will emerge.

The example below seems straight forward. Section S within section A inherits from section R. Modifications in section R immediately take place in section S too since a dynamic approach of data representation is used (see Section 3.3). Section A with all its entities and their properties are passed on to section B. Thus, section B also has two subsections R and S. Any modification performed in section R within section A will even influence the contents of section B.

```
A {
   R {
      ...
   }
   S : R {
      ...
   }
}
B : A {
   R {
      ...
   }
   S {
      ...
   }
}
```

Ambiguity arises between section S within A and section S within B. This is illustrated in Fig. 1 and Fig. 2. We call the two cases implicit and explicit inheritance respectively. With implicit inheritance the inheritance information of all subsections is passed on to the successor. Whereas with explicit inheritance only the inheritance given explicitly by the user is followed. In the first case section S within section B is inherited from section R within section B – the inheritance scheme is defined within section A and was passed to section B. In the second case section S within section B does not contain any explicit inheritance information. But because section B is inherited from section A that section S is inherited from section S of section A.

Modifications to the subsection S in section A will therefore influence the subsection S of section B only in the case of explicit inheritance. We have implemented the explicit inheritance scheme for the following reason: Imagine section A stands for the world, section R is an ape, and section S, which is derived from section R is a human. If we derive section B from section A to create another world its section R is again an ape and section S again a human. This is not the case with implicit inheritance. Section R of the new world would be an ape again but section S something else derived from an ape.

To demonstrate the inheritance algorithm we modify this example by adding the item `foo` to section R within section A. The inheritance scheme is defined, no inherited section is locally modified:

```
A {
   R {
      foo = 1;
   }
   S : R;
}
B : A;
```

Since the Input Deck database uses the explicit inheritance scheme, finding or referring to the keyword `~.B.S.foo` causes the inheritance algorithm to generate the following fullnames finally resulting in the existing keyword `~.A.R.foo`.

```
~.B.S.foo
~.A.S.foo
~.A.R.foo
```

The implicit inheritance scheme would generate the following sequence shown below. So, if both items `~.A.S.foo` and `~.B.R.foo` exist the Input Deck inheritance scheme would find `~.A.S.foo`.

```
~.B.S.foo
~.B.R.foo
~.A.R.foo
```

3.3 DATA REPRESENTATION

For internal data representation of inheritance two approaches are viable: First, all data structures are copied to represent the structure of the base section and its derived section. New entries are appended, modified entries overwrite existing ones. At any time each entry has its corresponding data representation. We call this the static approach. Secondly, no data is copied, only the inheritance is registered. New and modifying entries are appended. Only the specified entries are stored. We call this the dynamic approach.

The advantage of the first approach is its simple data storage. Finding an arbitrary entry is very easy and fast. On the other hand, the same information has to be stored several times as items are inherited by derived sections, thus consuming memory. In addition this concept proves to be very complicated when a base section gets changed. For instance, when an entry in the base section is deleted it has to be deleted in all sections inherited from it but only if it has not been locally modified. So after each modification the database has to check the consistency of its data. Thus, an inherited section has to know all its parents and each section must know all its successors.

The advantage of the second approach is its self consistency. Modifications of an item are automatically visible in the derived sections too. For instance, if an entry is deleted in a base section no work has to be done on its succeeding sections since it had the same representation as the deleted one. The disadvantage is that finding an entry is more complicated and more time consuming since an item inherited by a derived section is stored in one of its base sections. For the Input Deck database we have chosen the second approach because it is more flexible.

3.4 EXPRESSION TREES

Any data that is read in is stored in nodes. Since all types of nodes are represented by the same data structure they can be handled similarly. Nodes can be classified in five groups. The first group consists of variables, sections, and aliases which can be queried directly by the application. Other nodes are used to store information on the input files as they are read, e.g. nodes for `include` commands or just comments. References are nodes used within expressions to refer to variables. Function nodes can be operators, built-in functions, functions defined by the application or the user. Finally, value nodes hold constant values or results from calculations.

An expression is stored as a forest which is a special kind of a tree [9, 10]. Each operator and each function stores a tree holding a list of operands. For instance, a sum is represented by a single '+' operator node and one node for each summand stored in a list. Fig. 3 shows the expression tree the Input Deck database will use to store the equation

$$z = \frac{1}{2} \cdot \ln\left(\frac{1+r}{1-r}\right) \tag{1}$$

For evaluating expression trees the Input Deck evaluation module is used which is part of the Input Deck database. It is important to note, that this module uses

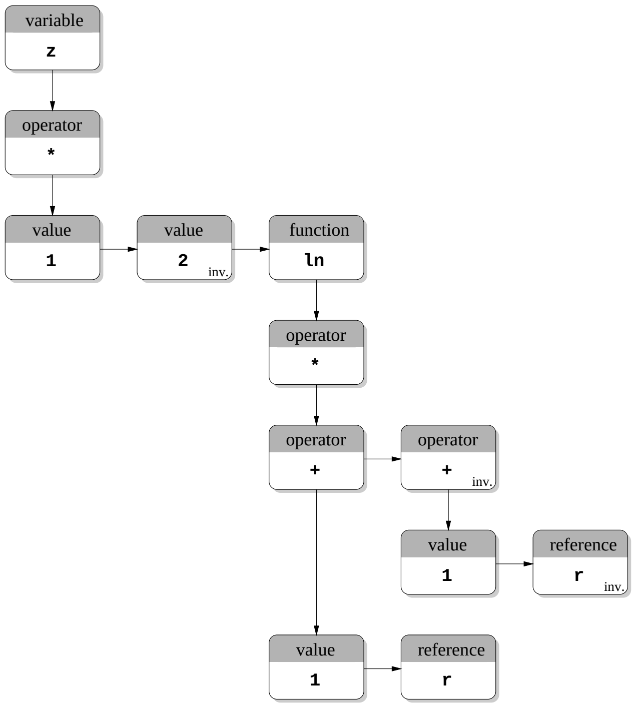


Figure 3: Example of an expression tree

an object-oriented approach when evaluating expressions. Thus, operators and functions are overloaded for different argument data types. Since the data type is determined at runtime the corresponding operator is also chosen at runtime. For instance, adding two variables a and b will deliver an integer if both a and b are integers, and a string if at least one of the variables is a string. If no operator exists for the operand data types created at runtime an error message is reported to the application.

Several arithmetic and logic operands are implemented in the evaluation module. The precedence of all operators is as usual and explained in [5]. The operands '-' and '/' are handled in a special way. Their function is taken over by the operands '+' and '*' where the corresponding nodes are marked as inverse (see Fig. 3).

The advantage of this kind of tree compared to, e.g., regular binary trees [9, 10] is that all operators of the same kind which have the same precedence are represented only by one node. As a consequence, simplification of stored expressions is easy and the total number of nodes is small. These simplifications are important for performance reasons.

When the forest shown in Fig. 3 is evaluated by the evaluation module temporary values are created. For instance, before the logarithm is calculated the expressions $1 + r$ and $1 - r$ are evaluated. The results are stored in temporary values which are used to perform the division. To avoid memory allocations during calculation operator nodes, function nodes, and reference nodes contain space to hold a temporary value. Thus, no memory is allocated during evaluation and, therefore, evaluation of expressions is fast.

During evaluation the evaluation module marks nodes which do not depend on external variables or time dependent functions as constant since they will not change. For all subsequent calculations of a node marked constant simply its value is returned. Thus, constant expressions are evaluated only once.

3.5 FUNCTIONS

The Input Deck function module which is part of the Input Deck database manages all functions which can be used in IPL. Functions can be used within expressions. A function is called during evaluation of an expression by the evaluation module. Functions always have a return value which can be of any data type. The function module stores the name of a function, the number of regular parameters, the number and names of optional parameters, and the kind of the function. There are three different kinds of functions, depending on where they are defined.

Several built-in functions are implemented which are either mathematical or supplementary functions, respectively. Mathematical functions are defined for complex numbers in general. Many-valued mathematical functions are implemented, too. Supplementary functions are conversion functions including some special functions. Conversion functions are used to convert the result of an expression to a given value type. Not all combinations are possible, e.g. converting a text to a boolean is not possible. Special functions can be used, e.g., to extract the unit of an expressions value, to extract the real or imaginary part of a complex number, or to get the value of an environment variable, or the name of a given item. The if() statement can be used for conditional evaluations whereas only the *if* or the *else* expression gets evaluated depending on whether the condition evaluates to true or false which is important for performance reasons.

```
a = asin(z);
b = if (x > y,
        string(realpart(a)),
        string(imagpart(b)));
```

In this short example, we calculate the inverse sine of z. The variable z may be a complex number. If x is greater than y we convert the real part otherwise the imaginary part of the result to a string.

Application specific functions are another powerful feature of the Input Deck function module. The application can register an arbitrary number of functions which are coded in C in the application. Application specific functions overwrite built-in functions. Because functions have to be registered before they are used all functions have to be registered before the IPL files are loaded. When these functions are used in expressions, the evaluation module calls the corresponding function of the application.

User-defined functions can be easily defined in IPL. Therefore, the functionality of the programming language can be enlarged and customized. User-defined functions overwrite built-in and application specific functions and are defined similarly to variables. The function name is followed by an argument list enclosed in parentheses, e.g.,

```
my_sqrt(x, y) =
  fabs(if (x > y,
      x * sqrt(1 + y * y / (x * x)),
      y * sqrt(1 + x * x / (y * y))));
```

4 APPLICATION INTERFACE

The Input Deck database offers an extensive application interface (API). The application can open as many different databases within the Input Deck database it needs simultaneously and use them in parallel. Moreover disjunct IPL files may be merged.

The application may query or modify variables and sections. They can be found by their name or by using iterators. Finding an item by its name presupposes that the application knows the exact *fullname* of the item. By defining a working section relative names can be given omitting the current section name. Using this method the application can easily query certain items or check for their existence.

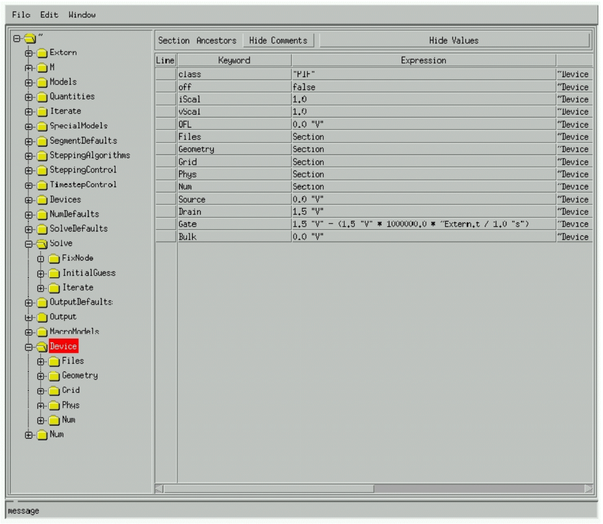
Additionally, iterators can be used to step through a section, up- and downwards, or through the whole input deck. Only non-auxiliary items can be found. An iterator must be initialized with a specific item to start from. The user can use as many iterators as necessary defining simple filters to match either sections or variables or both.

When an item is found, several operations can be performed depending on its type and the privilege of the application. There are two levels of privileges in the Input Deck database: The application privilege, which is the default, allows the application only to perform uncritical operations like finding items or evaluating and setting keywords. The editor privilege enables an external editor to perform more crucial operations, like renaming, moving, removing of items or modifying properties, inheritance and so on, to edit the whole IPL database. The API exists for the programming languages C, C++, and LISP.

```
#include "ipd++.hh"
...
Ipd  ipd("input.ipd");

for (IpdVarIterator vit(ipd.begin("~.Defaults"));
                        vit != ipd.end();
                        ++vit) {
   cout << vit.name() << " = " <<  double (*vit);
   cout << endl;
}
```

Figure 4: Dump all variables of section `~.Defaults`



Line	Keyword	Expression	
	class	"PIF"	~Device
	off	false	~Device
	iScal	1.0	~Device
	vScal	1.0	~Device
	OFL	0.0 "V"	~Device
	Files	Section	~Device
	Geometry	Section	~Device
	Grid	Section	~Device
	Phys	Section	~Device
	Num	Section	~Device
	Source	0.0 "V"	~Device
	Drain	1.5 "V"	~Device
	Gate	1.5 "V" - (1.5 "V" * 1000000.0 * ~Extern.t / 1.0 "s")	~Device
	Bulk	0.0 "V"	~Device

Figure 5: GUI

In the example shown in Fig. 4 we use the C++ interface of the Input Deck API. First the file “input.ipd” is opened. Using the iterator `vit` all variables of the section `~.Defaults` are read one after the other. The name of each variable and its value casted to a `double` are dumped to standard output.

5 INPUT DECK EDITOR

To visualize the section hierarchies and the inheritance dependencies the Input Deck database is equipped with an Input Deck editor with a graphical user interface (see Fig. 5). The section hierarchy is shown at the left side of the window. When a section is opened its subsections get visible in the hierarchy. The contents of the current section are shown at the right side of the window. All items can be created, overwritten, moved, renamed, or deleted. Locally modified variables are highlighted to distinguish them from inherited ones.

The editor does not access the IPL files directly. It manipulates the Input Deck database. All control and communication with the database are made using the API with the editor privilege set (see Section 4). This gives the editor full control over the Input Deck database.

6 APPLICATION: MINIMOS-NT

MINIMOS-NT [1] is a complex general purpose device and circuit simulator. The simulator can cope with arbitrary device structures and geometries, including heterostructures. MINIMOS-NT supplies a variety of physical models, abstracted material handling and a model server [11] managing user-defined physical models. The Input Deck database is used to control the simulator in many different ways:

The material database (MD) provides the parameter values for the physical models which are handled by a separate module of the simulator, the model server [11]. The MD is implemented as a section which contains subsections for each material. To administrate common material properties, real materials like Si and SiO2 are inherited from abstract materials like semiconductors and insulators which handle the common properties and define the available models and their parameters. For each model class (e.g., permittivity or band edge energies) there is a keyword to select the model instance and a corresponding subsection with the parameters for each model instance. The default database defines the default models and the default values. Each geometric region of a device contains a MD of its own which is inherited from the global MD. This architecture has the following advantages:

- It is easier maintainable compared to hardcoded default values.
- The customer can easily create a personal, e.g. calibrated, version of the MD for his own technology or several versions for different technologies.
- In conjunction with the model server, this concept allows for straight forward extension of the MD. When a new model is added to the model server (either in C++ or in a special interpreter language), it follows the very same rules as for predefined internal models. Each customer can have a library of proprietary models with a proprietary MD. This proprietary MD is inherited from the global MD and substitutes it as the default MD. This minimizes the adaption effort for a new release of the simulator.
- As the simulator deals only with abstract material classes (e.g., semiconductors and insulators) new materials can easily be added without changing the source code of the simulator. This is very important considering the many materials and alloys which are used in state-of-the-art process technologies.

As can be seen, many similar versions of the MD can exist at the same time, one for each geometric region and material. This would be hardly possible in an efficient way using the static approach (Section 3.3) For the device description the device geometry and the doping profiles are stored in a separate file. This separate file contains only information available from process simulators. All other simulation-relevant information (bias voltages, models used) are handled by the input deck with a separate section for each device. Thus using the inheritance feature device libraries can easily be realized and maintained.

In mixed-mode simulations a circuit consists of several devices. Aim of the `Circuit` section is to connect the devices to form a certain circuit. Therefore, for each device in the circuit a corresponding device section must exist. This device section is the same for single-mode and mixed-mode simulation except some additional information which is only needed in the mixed-mode case. Compact models as used in SPICE[12] are also available. They are realized with a device section similar to the device section of distributed devices. This allows for easy switching between the simpler compact models and their much more complex distributed representation.

All sorts of general information like input files, output files, logging information is also controlled by the input deck. Due to the many features it is very convenient to write a general input deck which is customized by auxiliary keywords to control similar simulations. A typical case is the evaluation of new models: instead of modifying some parameters like filenames and modelnames throughout the whole input deck manually (which often creates confusion and inconsistencies), these parameters are controlled by auxiliary keywords as shown in the following example.

```
// aux device = "deviceA";
aux device = "deviceB";

// aux model = "modelA";
aux model = "modelB";

Device : DeviceDefaults
{
   Output { file = ~device + "_out_" + ~model); }

   Phys
   {
      // Local version of the material database
      SegmentDefaults
      {
         impactIonizationHD = ~model;
      }
   }
}
```

Figure 6: A short MINIMOS-NT IPL file

As there are many possibilities and levels of accuracy to describe the transport of carriers in semiconductors a general description mechanism is of crucial importance. A typical way of solving a complex equation set is to base its solution on the solution of a simpler one. Thus different equations have to be assembled at each stage of the solution process. As these equations are normally highly nonlinear, they have to be solved iteratively (Newton-scheme) until a distinct error criterion is fulfilled. This solution process may fail which has also to be detected. Some steps of this iteration scheme may be left out under certain circumstances. To take any of the decision stated above it is necessary to have information about the current status of the simulation, like update norms and iteration counters. This information is provided by the simulator who stores it in a special section of the input deck database which in turn can be used to build the desired expressions.

7 APPLICATION: MKWAFER

The program MKWAFER is a recently developed tool to create three-dimensional input-data suitable for device (MINIMOS-NT) or process simulators like MCIMPL [13] or ETCH3D [14]. This tool is part of the wafer-state suit [15].

```
Cube {
    // coordinates are X/Y/Z
    points = [ [0.0, 0.0, 0.0], [1.0, 0.0, 0.0],
               [1.0, 1.0, 0.0], [0.0, 1.0, 0.0],
               [0.0, 0.0, 1.0], [1.0, 0.0, 1.0],
               [1.0, 1.0, 1.0], [0.0, 1.0, 1.0]
             ];
   // positive orientation of faces identifies
   // the inner region of the cube
   solid = [ [0, 1, 2, 3], // bottom face
             [4, 7, 6, 5], // top face
             [5, 6, 2, 1], // front face
             [4, 0, 3, 7], // back face
             [4, 5, 1, 0], // left face
             [7, 3, 2, 6]  // right face
           ];
   Scaling  { x = 1.0; y = 1.0; z = 1.0; };
   Offset   { x = 0.0; y = 0.0; z = 0.0; };
};
```

Figure 7: Include file cube3d.ipd

In Fig. 7 the definition of a simple cubical geometry section named Cube is shown. It contains a list of all points as well as a list of the faces. The parameters Scaling and Offset allow for the combination of several cubes as geometric primitives to form a certain geometry. This section serves as a base class to all sections defined in the IPL file shown in Fig. 8.

```
#include "cube3d.ipd"
Geometry
{
  E1: ~Cube {
     Scaling { x =  80.0; y =  20.0; z = 10.0;}
     Offset  { x =  35.0; y =  60.0; z = 20.0;}
  }
  E2: E1 { Offset { y = 120.0; } }
  E3: ~Cube {
     Scaling { x =  20.0; y =  40.0; z = 10.0;}
     Offset  { x =  95.0; y =  80.0; z = 20.0;}
  }
}
```

Figure 8: Simple device geometry description

Fig. 8 gives the complete definition of the c-shaped emitter contact at the top of the device shown in Fig. 9. The program scans the top-level section Geometry. For each sub-section (E1, E2, E3) contained in this section a separate segment is created. Note that, to account for a short input description, inheritance is used to pass the points and faces of Cube to each sub-section only overriding the values for the scaling and the offset.

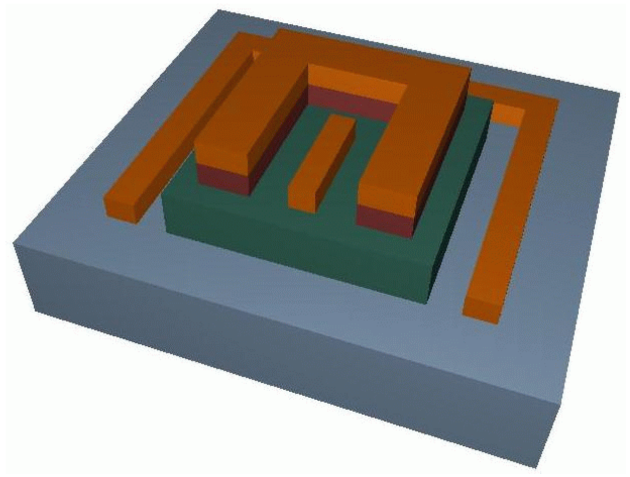


Figure 9: A schematic three-dimensional HBT device

As a last step the resulting geometry is gridded using the meshing tool DELINK [16]. Fig. 9 shows a plot of the final geometry as it is created by MKWAFER.

This tool is primarily intended as a three-dimensional device generator to aid in performing general optimization [17], inverse modeling tasks [18], or model parameter extraction [19] as they are carried out by TCAD frameworks like SIESTA or VISTA [20].

8 CONCLUSION

We have presented a new input deck concept using an object-oriented database which is well suited to control TCAD applications. Sets of parameters can be specified and mathematical expressions can be used. Default values can be changed easily since all parameters are accessible to the user in simple text files. The controlling language IPL is easy to learn since it is similar to C++ but not so complex.

All data inquired by the application are stored in keywords. Keywords are grouped in sections which can be nested to build any desired kind of tree. Using inheritance any number of hierarchical trees can be passed to another section building a new hierarchy. Due to this powerful feature this mechanism is well suited to provide hierarchically ordered sets of defaults.

The Input Deck database provides a well defined flexible dynamic inheritance mechanism. Multiple and conditional inheritance is supported. Since inheritance is performed at runtime changes to the internal structures simultaneously take place in the derived sections.

Applications can be controlled by three different kinds of information received from the Input Deck database. First, keywords can be queried or tested for their existence. Keywords can be found either by their name or by using iterators. Secondly, the order of items specified by the user in the IPL input files is preserved. Using iterators each item can be queried in turn. By changing the order the user can influence the order the items are read by the application. Thirdly, an application can add application specific functions. Once a keyword is queried by the application the expression contained is evaluated. If the expression contains an application specific function, it is called by the evaluation module.

References

[1] T. Binder, K. Dragosits, T. Grasser, R. Klima, M. Knaipp, H. Kosina, R. Mlekus, V. Palankovski, M. Rottinger, G. Schrom, S. Selberherr, and M. Stockinger. *MINIMOS-NT User's Guide*. Institut für Mikroelektronik, 1998.

[2] B. Reinisch. Eine universelle Eingabeverwaltung für die Steuerung von Simulatoren. Diplomarbeit, Inst.f.Mikroelektronik, Technische Universität Wien, 1997.

[3] V. Paxson. *GNU Flex Manual, Version 2.5.3*. Free Software Foundation, Cambridge, Mass., May 1996.

[4] C. Donnelly and R. Stallman. *GNU Bison Manual, Version 1.25*. Free Software Foundation, Cambridge, Mass., May 1996.

[5] B. Stroustrup. *C++ Programming Language*. Addison-Wesley, 1997.

[6] G. Satir and D. Brown. *C++ The Core Language*. O'Reilly & Associates, 1995.

[7] J.D. Ullman. *Database Systems*. Computer Science Press, 1982.

[8] B.R. Rao. *Object-Oriented Databases*. McGraw-Hill, 1994.

[9] D.E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley, Reading, Massachusetts, 3rd edition, 1997.

[10] R. Sedgewick. *Algorithms in C*. Addison-Wesley, 1990.

[11] R. Mlekus and S. Selberherr. Object-Oriented Algorithm and Model Management. In *Intl. Conf. on Applied Modelling and Simulation*, pages 437–441, Honolulu, Hawaii, USA, August 1998.

[12] P. Antognetti and G. Massobrio. *Semiconductor Device Modeling with SPICE*. McGraw-Hill, 1988.

[13] A. Hössinger and S. Selberherr. Accurate Three-Dimensional Simulation of Damage Caused by Ion Implantation. In *Proc. 2nd Intl. Conf. on Modeling and Simulation of Microsystems*, pages 363–366, San Juan, Puerto Rico, USA, April 1999.

[14] W. Pyka, R. Martins, and S. Selberherr. Efficient Algorithms for Three-Dimensional Etching and Deposition Simulation. In K. De Meyer and S. Biesemans, editors, *Simulation of Semiconductor Processes and Devices*, pages 16–19. Springer, Leuven, Belgium, September 1998.

[15] T. Binder and S. Selberherr. Object-Oriented Wafer-State Services. In *14th European Simulation Multiconference*, pages 360–364, Ghent, Belgium, May 2000.

[16] P. Fleischmann and S. Selberherr. A New Approach to Fully Unstructured Three-dimensional Delaunay Mesh Generation with Improved Element Quality. In *Simulation of Semiconductor Processes and Devices*, pages 129–130, Tokyo, Japan, 1996. Business Center for Academic Societies Japan.

[17] R. Plasun, M. Stockinger, R. Strasser, and S. Selberherr. Simulation Based Optimization Environment and Its Application to Semiconductor Devices. In *Intl. Conf. on Applied Modelling and Simulation*, pages 313–316, Honolulu, Hawaii, USA, August 1998.

[18] R. Strasser, R. Plasun, and S. Selberherr. Practical Inverse Modeling with SIESTA. In *Simulation of Semiconductor Processes and Devices*, pages 91–94, Kyoto, Japan, September 1999.

[19] V. Palankovski, R. Strasser, H. Kosina, and S. Selberherr. A Systematic Approach for Model Extraction for Device Simulation Application. In *Intl. Conf. Applied Modeling and Simulation*, pages 463–466, Cairns, Australia, September 1999.

[20] R. Strasser, Ch. Pichler, and S. Selberherr. VISTA - A Framework for Technology CAD Purposes. In W. Hahn and A. Lehmann, editors, *9th European Simulation Symposium*, pages 450–454, Passau, Germany, October 1997. Society for Computer Simulation International.