Dipartimento DII
Università Politecnica delle Marche (Italy)

PKtool 2.3.0

User Manual

1 INTRODUCTION

This document represents a user manual that covers the technical and application details of the PKtool environment. Among the topics addressed, there are explained the steps to configure a SystemC description for PKtool analysis, with the support of concrete examples.
This manual is referred to the PKtool version 2.3.0, and provides a comprehensive treatment of the topics introduced in the PKtool overview document. All the information concerning installation can be found in the 'INSTALL' file placed in the top directory of a PKtool release.
The contents are organized as follows:

The first two sections report an in-depth description of some components used in PKtool analysis. Sections 3-10 cover the details related to system configuration and simulation phases. Finally, Section 11 illustrates the default power models made available by PKtool.
In the following, we will simply use the term *module* to indicate an sc_module, i.e. the component provided by SystemC to define modular entities. Moreover, the term 'power estimations' will be often used to indicate estimations in conventional way, without reference to their physical nature (energy, power, commutations).

2 POWER MODELS

2.1 Introduction

The power estimations resulting from PKtool analysis are based on the evaluation of user-selected power models. A power model is commonly defined through predetermined computations (model formulation) that return a power estimation on the basis of specific data (model data). Model formulation can be expressed in several ways, such as analytical formulas, algorithmic procedures, relational tables [1]. Model formulation and model data represent the central elements of a power model in regard to its definition and application. In the following, these elements will be the main references to characterize a power model.
Within PKtool environment, a power model is defined through a C++ description incorporated inside the tool implementation. Such description specifies all the related computations in the form of standard functions requiring model data as input parameters.
As concerns model data, as already discussed in the overview document, we can distinguish two categories:

1) *static data*: information available before the beginning of a simulation and not dependent on the run-time evolution of the system.

2) *dynamic data*: information available only during simulation, on the basis of the run-time system evolution.

Typical examples of static data may be represented by technology and operative parameters; typical examples of dynamic data may be given by signal information, e.g. switching activity.
During a PKtool simulation, model data have to be specified and linked to the applied power models, making so relevant the distinction between static and dynamic data. For this purpose, PKtool provides different solutions for handling model data. In particular, dynamic data are associated to signal information and are automatically handled by means of augmented signal capabilities. As concerns static data, their specification is carried out at the beginning of the simulation through the procedure illustrated in Section 7.

2.2 Power model categories

In PKtool environment, power models can be classified into two categories according to their output estimation type. More precisely, PKtool is compatible with power models that provide estimations in terms of energy or total commutations. From now on, we will refer to these two kinds of power models as *energy models* and *commutation models* respectively.
PKtool can be applied also with power models returning estimations in terms of average power or commutation rate. In fact, such power models can be easily turned into equivalent energy or commutation models by introducing the simulation time in their formulations.
When a module is configured for PKtool analysis, the user must specify if the applied power model is an energy or a commutation model. As will be shown in 6.4, this specification is realized through a macro instruction in the power_module class.
Power models can be further classified with respect to the modalities used for their evaluation; more precisely, we can distinguish between *cumulative models* and *cycle-accurate models* [1]. The first category represents power models evaluated only at the end of a simulation period, providing an overall power estimation. The required model data usually consist in average values or time-cumulated data. Conversely, cycle-accurate power models are evaluated at every cycle of a simulation period, providing distinct estimations referred to the single cycle times. In contrast with

cumulative power models, the required model data are usually cycle-based quantities, e.g. the Hamming distance between consecutive input patterns. From a cycle-accurate power model it is always possible to get an overall estimation, by summing up the partial estimations computed in each cycle.

2.3 Power model libraries and PKtool default libraries

PKtool is not related to a particular power model but makes available a variety of power models that a user can select without limitations. All the available power models are incorporated into model libraries integrated into the software implementation of the tool. Inside a model library, it is possible to include only power models based on the same estimation type, that is energy models or commutation models. This implies two possible kinds of model library, referable to as *energy library* and *commutation library*.

Within a model library, the power models are referenced through a double identifier composed by a non-negative integer index (*model index*) and a character string (*model name*). Such identifiers are unique for each power model of the library, and two power models with the same index or name cannot be present.

The PKtool framework makes it possible to define several model libraries and let them coexist together. However, during a simulation session, only one energy library and one commutation library can be enabled. At the present time, PKtool makes available an energy library and a commutation library called *pk_default_energy_lib* and *pk_default_comm_lib* respectively. During a PKtool simulation such libraries are enabled by default, and the user can have a direct access to the related power models. These libraries and their power models will be described in Section 11.

A user is allowed to define customized power models and make them applicable for PKtool analysis. Such power models can be incorporated into the PKtool default libraries, in addition to the power models already available. The definition of new power models can be realized by following some specific rules. However, the related details are not reported in this user manual and could be dealt with in a future documentation.

3 AUGMENTED SIGNALS

3.1 Introduction

PKtool provides the means for computing characteristic signal data often required by power models, such as bit length, bit commutations, operation statistics. These components are called *augmented signals* and are based on a set of augmented types defined in the PKtool framework.
An augmented signal can be regarded as a smart signal, able to extend its basic behaviour with further capabilities to provide additional information. When a module is configured for PKtool analysis, augmented signals can be used for selecting those signals whose characteristic data are required for power estimations. The instance of augmented signals is realized at code level, by modifying the original types of the signals to be monitored. More precisely, the original types have to be replaced by corresponding augmented types provided by PKtool.
For example, let us suppose this signal is instanced in the original implementation of a module:

```
sc_uint<16> bus;
```

The signal is called *bus* and is associated to the SystemC type sc_uint<16>. In order to convert this signal into an augmented counterpart, it is necessary to modify the instance instruction in this way:

```
sc_uint_aug<16> bus;
```

where the original type has been replaced with the matching augmented type, *sc_uint_aug<16>*. At this point the bus signal has become an augmented signal, so gaining all the specific capabilities.

3.2 Signals that can be augmented

In general, a module can be constituted by two kinds of signals: I/O ports and internal nodes. Currently, it is possible to augment I/O ports without limitations, compatibly with the available augmented types provided by PKtool (3.4). As concerns internal nodes, only the instances defined as class members can be augmented with full functionalities. Internal nodes given by local entities (typically, variables defined inside functions) can be converted into the augmented format with some limitations. More precisely, such signals can be augmented only if defined in unspawned processes (sc_thread, sc_method, and sc_chtread) [3] or functions called by an unspawned process. Moreover, local internal nodes are not associated to identification fields (3.5) and the data that can be extracted from them are limited to total commutations and operation occurrences.
In order to clarify these distinctions, let us consider the following module class:

```
#include "systemc.h"

SC_MODULE( example_mod1 )
{

 // I/O ports

 sc_in<double> input;
 sc_out<double> output;
 sc_in_clk clk;
 // internal nodes defined as class members

 double bus_1;
 sc_int<32> bus_2;
 bool ctr_1;


 // member function defining a spawned process

 void data_gen()
 {
  sc_int<32> bus_3;   // local internal node
  ...
  ...
 };


 // member function defining an unspawned process

 void proc1()
 {
  sc_int<32>  bus_4;  // local internal node
  ...
  ...
 };


 void proc2()
 {
  sc_spawn(sc_bind(&example_mod1::data_gen, this));
  ...
  ...
 };


 SC_CTOR(example_mod1)
 {
  SC_METHOD(proc1)
  sensitive << clk.pos();

  SC_METHOD (proc2)
  sensitive << clk.pos();

 }

}
```

All the I/O ports and the internal nodes bus_1, bus _2, bus _4, and ctr_1 can be converted into augmented signals (bus_4 with limited capabilities), whereas such conversion cannot be made for the internal node bus_3.

3.3 Data provided by an augmented signal

At the moment, an augmented signal can provide the following data:

1) bit length: the bit length with respect to the binary representation.
2) bit commutations: the commutations occurred at single bit level.
3) total commutations: the sum of all the commutations occurred, i.e. the sum of all the commutations at bit level.
4) bit probabilities: the fractions of simulation time in which the bit values are a logic high [2].
5) operation occurrences: the number of times in which a specific operation has been carried out (at the moment available only for the arithmetic operators +, -, *, /)

The last four data are computed and made available in the course of a simulation, on the basis of the run-time evolution; bit length is available from the beginning of a simulation on the basis of the original signal type. Operation occurrences are computed only in application with operator-based power models (10.2).
In order to show the values assumed by these data, let us consider again the augmented signal *bus* introduced in 3.1. As concerns bit length, the corresponding value is 16 in accordance with the sc_uint<16> type. As concerns the other data, their values can be determined only in reference to the signal evolution during a simulation. For this purpose, let us consider the following assignments during a simulation period of [0 ns – 40 ns]:

TIME	SIGNAL VALUE
0 ns	0 (initial value)
10 ns	2
20 ns	5
30 ns	16
40 ns	4

The above scheme reports the values assumed by the signal under the assumption of a clock period of 10 ns. With regard to bit commutations, total commutations and bit probabilities, the corresponding values are reported in the following table:

TIME (ns)	0	10	20	30	40
bit representation	0000000000000000	0000000000000010	0000000000000101	0000000000010000	0000000000000100
bit commutations	0000000000000000	0000000000000010	0000000000000121	0000000000010222	0000000000020322
total commutations	0	1	4	7	9
bit probabilities	0000000000000000	0000000000000000	00000000000000 0.5 0	0000000000000 0.33 0.33 0.33	00000000000 0.25 0 0.25 0.25 0.25

The first row reports the binary representations of the signal; the successive rows show the values referred to bit commutations, total commutations and bit probabilities. Bit commutations and bit probabilities are represented by integer/double vectors with size equal to the bit length. Total commutations are instead given by an integer value that coincides with the sum of bit commutations. From these data, by means of simple manipulations, it is possible to derive other signal statistics such as average bit commutations and commutation density [2].

3.4 Available augmented types

PKtool provides the augmented counterparts for many of the types used for modelling signals in SystemC/C++. The following list reports the augmented types currently available:

ORIGINAL TYPES	AUGMENTED TYPES
I/O PORTS	
sc_in<T>	sc_in_aug<T>
sc_out<T>	sc_out_aug<T>
sc_inout<T>	sc_inout_aug<T>
sc_in_resolved	sc_in_resolved_aug
sc_out_resolved	sc_out_resolved_aug
sc_inout_resolved	sc_inout_resolved_aug
sc_in_rv<n>	sc_in_rv_aug<n>
sc_out_rv<n>	sc_out_rv_aug<n>
sc_inout_rv<n>	sc_inout_rv_aug<n>
INTERNAL NODES	
sc_bit	sc_bit_aug
sc_logic	sc_logic_aug
sc_bv<n>	sc_bv_aug<n>
sc_lv<n>	sc_lv_aug<n>
sc_int<n>	sc_int_aug<n>
sc_uint<n>	sc_uint_aug<n>
sc_signed	sc_signed_aug
sc_unsigned	sc_unsigned_aug
sc_bigint<n>	sc_bigint_aug<n>
sc_biguint<n>	sc_biguint_aug<n>
sc_fix	sc_fix_aug

sc_fix_fast	sc_fix_fast_aug
sc_fixed	sc_fixed_aug
sc_fixed_fast	sc_fixed_fast_aug
sc_ufix	sc_ufix_aug
sc_ufix_fast	sc_ufix_fast_aug
sc_ufixed	sc_ufixed_aug
sc_ufixed_fast	sc_ufixed_fast_aug
bool	bool_aug
char	char_aug
int	int_aug
float	float_aug
double	double_aug
signed	signed_aug
unsigned	unsigned_aug
long	long_aug
short	short_aug

The signal types have been logically subdivided considering their use as I/O ports or internal nodes. Given an original signal type, the rule for having the augmented counterparts consists in adding the term '_aug' in the original type name.
As concerns internal nodes, the available augmented types cover almost all the signals that can be modelled through C++ native types and specific SystemC types. At the moment, the augmentable I/O ports are only those related to the interfaces *sc_signal_if* and the resolved versions.

3.5 Identification fields

An augmented signal is univocally associated to an identifier set. Such identifiers allow to distinguish different augmented signals instanced in a same context. More precisely, an augmented signal is identified by three fields: module name, signal category and numeric index. The first field is the univocal name of the belonging module [3]. In regard to signal category, there are four possible values:

1) input port
2) output port
3) input-output port
4) internal node

The signal category *internal node* concerns the internal nodes that can be augmented in compliance with the limitations described in 3.2. The numeric index is a positive integer with the aim to distinguish augmented signals of the same category and belonging to the same module. Such index is assigned on the basis of the construction order, as specified by the standard C++ rules [4]. For

example, if inside a module there are instanced N augmented input ports, the first-built port is assigned to the index 1, the second-built port to the index 2, and so on.
In order to clarify the identification rules, let us consider the following module class:

```
SC_MODULE( example_mod2 )
{
 // ports

 sc_in<bool> clk;

 sc_in_aug<sc_int<32> > in1;
 sc_in<int> in2,in3;
 sc_in_aug<sc_uint<64> > addr;

 sc_out<bool> error;
 sc_out_aug<sc_uint<64> > out1, out2;

 sc_inout_aug<int> log1;
 sc_inout<int> log2;


 // internal nodes

 int_aug<64> st1;
 bool st2;
 unsigned st3;
 sc_uint_aug<16> st4, st5;

 // rest of the code
 ...
 ...
}
```

The class body comprises both ordinary and augmented signals. These latter are given by the input ports *in1* and *addr*, the output ports *out1* and *out2*, the input-output port *log1*, and the internal nodes *st1*, *st4*, and *st5*.
Now, let us consider the values assumed by the identification fields for this instance of example_mod2 :

```
example_mod2   module("master");
```

The name given to the module is *master*, which represents the module name identifier for all the augmented signals. The following table summarizes the identification fields for each augmented signal:

signal	module name	signal type	numeric index
in1	master	input port	1
addr	master	input port	2
out1	master	output port	1
out2	master	output port	2
log1	master	inout port	1
st1	master	internal node	1
st4	master	internal node	2
st5	master	internal node	3

4 STEPS FOR CONFIGURING A MODULE FOR PKTOOL ANALYSIS

4.1 Characterization of the steps

PKtool analysis are executed at level of the single modules constituting a monitored system. For each selected module, it is necessary to realize a configuration procedure based on some systematic steps. This section reports an overview of these steps, mentioned below in their logical order:

a) Inclusion of the header files for making visible PKtool class library.

b) Specification of the output estimation type (energy or commutations).

c) Specification of the power model to be applied.

d) Instance of the augmented signals whose data are required for power estimations.

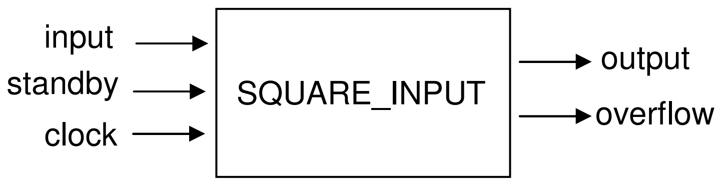
e) Assignment of the static data required by the applied power model.

f) Definition of a power state characterization, in the case of a configuration based on this approach.

For simplicity reasons, we will often refer to these steps through the alphabetical letter associated in the above list.
The steps a), b), c) are always to be carried out. The necessity of the steps d) and e) depends on the nature of the data required by the applied power models. The step f) is optional, and is to be considered only for analysis based on a power state configuration. The steps a), b), d) and f) are realized via code-level instructions, whereas the steps c) and e) are based on a specification procedure at the beginning of a PKtool simulation.

4.2 Application example for describing the configuration steps

The configuration steps will be illustrated by means of examples on a simple module with the following top level structure:



Such module is called *square_input* and its I/O layout is composed by three input ports (input , standby, and clock) and two output ports (output and overflow). The main process computes the square of the unsigned values sent to the input port, reporting the results onto the output port. This process is synchronous with respect to the positive edges of the clock. If the input value is greater than 255 an overflow condition occurs, which is communicated through the overflow port. If the standby port is set to true, the system enters into a standby state and stops its calculation activities for a period of three clock cycles. After this time, the ordinary behaviour is resumed if the standby port is set to false.
What follows is a possible SystemC representation of the square_input class:

```
#include "systemc.h"

SC_MODULE(square_input)
{

 sc_in<bool> standby;
 sc_in<sc_uint<8> > input;
 sc_out<bool> overflow;
 sc_out<sc_uint<16> > output;
 sc_in_clk clk;

 const unsigned over_value;

 void process()
 {
  while(true)
  {
   if(standby)
   {
    output = 0; overflow = false;
    wait();
    wait();
    wait();
   }
   while(standby) wait();
   if( input.read() >= over_value ){ output = 0; overflow = true;}
   else { output = (input.read()*input.read()); overflow = false;};
   wait();
  };
 };


 SC_HAS_PROCESS(square_input);


 square_input (sc_module_name name): sc_module(name),over_value(255)
 {
  SC_THREAD(process)
  sensitive_pos<<clk.pos();
  dont_initialize();

  output.initialize(0);
 };

};
```

5 INCLUSION OF THE PKTOOL HEADER FILE

The configuration for PKtool analysis firstly requires to make visible the PKtool class library. This is achieved by including the PKtool header file in the class implementation of the monitored modules. Such file is called *PKtool.h* and is incorporated into the PKtool software framework.
The inclusion of the PKtool header should be considered in two typical cases. The first situation concerns the instance of augmented signals inside a module class. As described in section 3, this operation entails the use of augmented signal types provided by PKtool, which can be made visible only via the PKtool header.
As a concrete example, let us consider the class defining the module square_input, assuming that the related code is reported in a header file called *square_input.h* :

```
//  square_input.h

#include "systemc.h"

SC_MODULE(square_input)
{

 sc_in<bool> standby;
 sc_in<sc_uint<8> > input;
 sc_out<bool> overflow;
 sc_out<sc_uint<16> > output;
 sc_in_clk clk;

 // rest of the code

 ...
 ...

}
```

If we want to augment the ports *input* and *output*, we must modify the above code in this way:

```
//  square_input.h

#include "systemc.h"
#include "PKtool.h"    //  PKtool header file

SC_MODULE(square_input)
{
 sc_in<bool> standby;
 sc_in_aug<sc_uint<8> > input;
 sc_out<bool> overflow;
 sc_out_aug<sc_uint<16> > output;
 sc_in_clk clk;

 // rest of the code
 ...

}
```

With respect to the original code, we have included the PKtool header before the class body and after the SystemC header. This makes possible the conversion of the ports input and output into their augmented counterparts. SystemC and PKtool headers must always be included according to the order shown in the example.
In general, the inclusion of the PKtool header could not be strictly mandatory in this case. Actually, such inclusion is necessary only if the applied power model requires specific signal data in its formulation, thus making necessary the instance of augmented counterparts for some signals.
The PKtool header must be included also for defining a power_module class (topic discussed more deeply in section 6). This situation can be illustrated through another example, in which we want to define a power_module class for square_input. As ordinary practice, the power_module code can be reported in a separate file that we could call *powmod*_squin.*h*. As concerns the initial #include directives, this file should begin in this way:

```
//  powmod_squin.h

#include " square_input.h"
#include "PKtool.h"            //  PKtool header file


// power_module class
...
...
```

The first header makes visible the module class, whereas the second one the PKtool components for defining the power_module class. Unlike the instance of augmented signals, in this case the inclusion of the PKtool header is always mandatory to configure square_input for PKtool analysis. Nonetheless, if this header has already been specified in square_input.h (for example, to instance augmented signals), its explicit inclusion can be omitted in powmod_squin.h.

6 DEFINITION OF A POWER_MODULE AND ITS COMPONENTS

6.1 Introduction

In order to configure a module for PKtool analysis, it is necessary to define a specific *power_module*. A power_module is a PKtool entity that allows to make a module monitorable by PKtool. Moreover, a power_module is also the place where to realize the configuration steps b) and f). Like an ordinary module, two sequential phases are to be considered for instancing a power_module:

1) definition of the power_module class.
2) instance of power_module objects.

The specific details will be described through a concrete application on square_input module.

6.2 Power_module class

First of all, it is necessary to define the power_module class. The simplest form for the class title is the following:

```
#include "square_input.h"
#include "PKtool.h"

POWER_MODULE_CLASS(square_input )
{
 ...
 ...
}
```

where we have used the parameterized macro *POWER_MODULE_CLASS*, with the name of the module class as parameter. This title definition is the most commonly used, and should be applied when a power_module class does not have to inherit from further classes.
Alternatively, the class title can be defined through a more classical C++ form:

```
struct POWER_MODULE(square_input): square_input, power_module_b,...
{
 ...
 ...
}
```

In this case, the power_module class is implemented by a struct whose name is given by the macro *POWER_MODULE*, with the name of the module class as parameter. This struct must inherit publicly from the module class and the *power_module_b* class; this latter is defined in the PKtool library. As implicitly shown in the code, the power_module class could inherit from further classes.

6.3 Constructor and destructor

The power_module constructor may be specified through two possible options. The simplest way is

```
POWER_MODULE_CTOR(square_input)
{
 ...
 ...
}
```

where we have used the parameterized macro *POWER_MODULE_CTOR*, with the name of the module class as parameter.
The second way is based on a more complex instruction:

```
PK_HAS_PROCESS(square_input );

POWER_MODULE(square_input) (::sc_core::sc_module_name nm, ...):
                         square_input(nm, ...), PK_PMB_CTOR, ...
{
 ...
 ...
}
```

First of all, we must specify the parameterized macro *PK_HAS_PROCESS* ; then, we must report the constructor title in an explicit form. The name of the constructor must be given by the parameterized macro *POWER_MODULE*. The constructor parameters can include an arbitrary number of elements. However, as shown in the example, it is mandatory to specify an sc_module_name parameter that will be passed to the module constructor. In the initialization list, it is mandatory to report the constructor of the module class and the macro PK_PMB_CTOR; this latter stands for the constructor of power_module_b. If necessary, the initialization list may include the constructors of further entities, such as internal members or inherited classes.
The power_module constructor can be expressed through the first and simpler option if these three conditions are true:

a) except for the mandatory sc_module_name parameter, the power_module constructor does not require further parameters.

b) the module constructor requires only an sc_module_name object as parameter.

c) except for the constructors referred to the module and power_module_b class, the initialization list does not have to include the constructors of further entities.

if one of these conditions is not verified, the power_module constructor should be expressed through the second option.
Within a power_module class the constructor covers an important role and must be always defined. In particular, as shown in sections 6.5 and 8.2, the constructor represents the place where to report sensitivity specifications.
As concerns power_module destructor, it must be defined in this way:

```
POWER_MODULE_DTOR
{
 ...
}
```

where we have used the macro *POWER_MODULE_DTOR* as destructor title. This is the unique way to express the power_module destructor; the use of a more classical form would lead to a wrong definition and a probable compilation error. Unlike the constructor, a destructor may be omitted inside a power_module class because its definition is not strictly necessary and depends only on specific needs.

6.4 Output estimation type

The configuration step b) concerns the specification of the output estimation type, i.e. if the output estimations are expressed in terms of energy or total commutations. Such specification implicitly defines also the kind of power models that can be applied, i.e. energy models or commutation models. This step is realized by reporting one of the following macros inside the power_module class:

PK_USES_ENERGY_MODELS

PK_USES_COMMUTATION_MODELS

The first macro must be selected for energy estimations, whereas the second one for commutation estimations. If we want to set energy estimations in our example, we must report this instruction into the power_module class:

```
POWER_MODULE_CLASS(square_input)
{
  ...
  ...

 PK_USES_ENERGY_MODELS

  ...
  ...
};
```

6.5 Sensitivity specifications

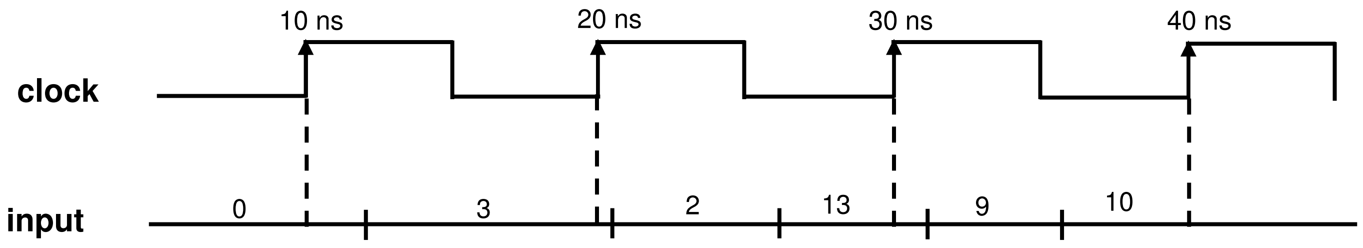
The power_module constructor is the place where to define the sensitivity specifications for some tasks that can be part of a PKtool analysis. These tasks consist in the updating of augmented input ports and the evaluation of cycle-accurate power models.
With regard to the first case, augmented signals need to have their characteristic data updated during a PKtool simulation; such updating is normally required when signal values change. In the current PKtool implementation, augmented input ports are not able to carry out this operation in an autonomous way. Actually, a user intervention is necessary to indicate when to execute the updating procedure. More into details, the user has to define an appropriate sensitivity specification inside the power_module constructor. From now on, we'll refer to such specification as *augmented input port sensitivity* (AIP sensitivity).
As an example, considering the power_module related to square_input, the AIP sensitivity could be defined in this way:

```
POWER_MODULE_CTOR(square_input)
{

 PK_INPORT_SENSITIVITY
 sensitive << clk.pos();

 ...
 ...
}
```

The AIP sensitivity is defined by the macro *PK_INPORT_SENSITIVITY* followed by explicit sensitivity instructions. These latter specify the events on which the updating procedure is to be carried out. The AIP sensitivity is set with the same syntax and rules for the sensitivity of an ordinary SystemC process. In the example, the AIP sensitivity consists in the positive edges of the clock signal. This means that the characteristic data of all the augmented input ports will be updated on these events, during a PKtool simulation. As general rule, the AIP sensitivity should include those events that can cause a change in the input port values. For a clocked module, the AIP sensitivity could consist in the clock triggering events.
Now, let us consider the effects of AIP sensitivity through an example concerning the port *input* instanced in square_input. Let us suppose that this port shows the following evolution during a PKtool simulation :



The clock period is set to 10 ns. The above representation shows the values assumed during the first 40 ns. For simplicity reasons, this example will be focused only on the signal data related to total commutations (3.3).
In compliance with the AIP sensitivity previously specified, the values used for computing the commutations are sampled on the positive clock edges:

TIME	SIGNAL VALUE
10 ns	0
20 ns	3
30 ns	13
40 ns	10

Accordingly, the following commutations are computed:

TIME	COMMUTATIONS	TOTAL COMMUTATIONS
10 ns	0	0
20 ns	2	0 + 2 = 2
30 ns	3	2 + 3 = 5
40 ns	3	5 + 3 = 8

The temporary values between consecutive clock cycles are not considered with this AIP sensitivity. In particular, coming back to the previous signal evolution, the values 2 and 9 are not sampled for updating total commutations.
For applications where we want to evaluate also the contributions of temporary values, the AIP sensitivity should be specified in different way. More precisely, the sampling events should occur whenever the signal value changes. This behaviour can be achieved through these instructions:

```
POWER_MODULE_CTOR(square_input)
{

 //AIP sensitivity

 PK_INPORT_SENSITIVITY
 sensitive << input;

 ...
 ...
}
```

Now the AIP sensitivity should sample all the values assumed by the input port, comprising also the temporary values:

SIGNAL VALUE	COMMUTATIONS	TOTAL COMMUTATIONS
0	0	0
3	2	0 + 2 = 2
2	1	1 + 2 = 3
13	4	3 + 4 = 7
9	1	7 + 1 = 8
10	2	8 + 2 = 10

Defining the AIP sensitivity does not represent a mandatory specification. It could be required only if augmented input ports are instanced in a module to be monitored.
Besides AIP sensitivity, the power_module constructor is also the place where to define sensitivity instructions for cycle-accurate power models (2.2). For brevity, we will refer to this specification as *cycle-model sensitivity*.
Cycle-model sensitivity should be defined when a cycle-accurate power model is applied. This kind of power model should be evaluated in each simulation cycle, providing partial estimations referred

to single cycle times. For enabling such evaluations, it is necessary to specify explicitly the corresponding cycle events by means of specific sensitivity instructions. In the typical case of a clocked module, the cycle concept is based on the clock synchronization and the cycle events should consist in the clock triggering events. The following code shows how we can define cycle-model sensitivity inside power_module constructor:

```
POWER_MODULE_CTOR(square_input)
{


 PK_CYCLEMODEL_SENSITIVITY
 sensitive << clk.pos();


}
```

Cycle-model sensitivity is defined by the macro *PK_CYCLEMODEL_SENSITIVITY* followed by specific sensitivity instructions. These latter declare the events on which the power model is to be evaluated; the form of such instructions is the same used for the sensitivity of SystemC processes. In the example, being square_input a clocked module sensitive to positive clock edges, the cycle-model sensitivity has been associated to such events.
The sensitivity specifications so far discussed have been shown by means of distinct instructions. However, if the triggering events are the same, it is possible to apply a simpler and unified specification, which can be referred to as *module sensitivity*. To illustrate the use of module sensitivity, let us consider the following situation:

```
POWER_MODULE_CTOR(square_input)
{

 // AIP sensitivity

 PK_INPORT_SENSITIVITY
 sensitive << clk.pos();


  // cycle-model sensitivity

  PK_CYCLEMODEL_SENSITIVITY
  sensitive << clk.pos();

}
```

The power_module constructor reports the definitions of an AIP sensitivity and a cycle-model sensitivity based on the same triggering events. In this case, the previous instructions can be alternatively specified by module sensitivity:

```
POWER_MODULE_CTOR(square_input)
{

  // module sensitivity
  PK_MODULE_SENSITIVITY
  sensitive << clk.pos();
}
```

Module sensitivity is defined by the macro *PK_MODULE_SENSITIVITY* followed by specific sensitivity instructions. These latter must be the same reported in the AIP sensitivity and cycle-model sensitivity.
It is possible the coexistence between module sensitivity and explicit sensitivity specifications without ambiguities:

```
POWER_MODULE_CTOR(square_input)
{

 // AIP sensitivity

 PK_INPORT_SENSITIVITY
 sensitive << inport_event;


 // cycle-model sensitivity

 PK_CYCLEMODEL_SENSITIVITY
 sensitive << model_event;



 // module sensitivity

 PK_MODULE_SENSITIVITY
 sensitive << clk.pos();

}
```

In this example, the explicit sensitivities are always prevailing over the module sensitivity. This means that the updating tasks for augmented input ports and cycle-accurate power models are executed on the notifications of inport_event and model_event.
It is also possible a hybrid situation as illustrated below:

```
POWER_MODULE_CTOR(square_input)
{

 // cycle-model sensitivity

 PK_CYCLEMODEL_SENSITIVITY
 sensitive << model_event;


 // module sensitivity (specifies implicitly AIP sensitivity)

 PK_MODULE_SENSITIVITY
 sensitive << clk.pos();

}
```

The updating task for a cycle-accurate power model is executed only when model_event is notified. However, in absence of an explicit specification, the AIP sensitivity results automatically included in the module sensitivity. As a consequence, if augmented input ports are instanced, their updating tasks are carried out on the positive clock edges.

6.6 Instance of power_modules

The selection of a module for PKtool analysis requires the replacement with a matching power_module. Similarly to augmented signals, this is simply achieved by modifying the original module type in the instance instruction. This operation is shown through the following description:

```
#include "square_input.h"

int sc_main ()
{

 // module instances

 square_input   squin_1 ("squin_1");
 square_input   squin_2 ("squin_2");

 // rest of the code
 ...
};
```

In the example two square_input modules, *squin_1* and *squin_2,* are defined in an sc_main function [3]. All the connection instructions and other possible entities are not involved in power_module instance and, therefore, have been omitted.
If we want to select squin_1 for PKtool analysis, we must modify its instance instruction in this way:

```
#include "powmod_squin.h"


int sc_main ()
{

 // module instances

 POWER_MODULE(square_input)  squin_1("squin_1");

 square_input   squin_2("squin _2");

 // rest of the code
 ...

};
```

where we have wrapped the module type in the macro POWER_MODULE. This is the only action to be done in the sc_main function to realize a power_module conversion. The header file "powmod_squin.h" must be included in order to make visible the power_module class.
At this point, squin_1 has become a power_module and is automatically selected for PKtool analysis. On the other hand, this does not happen for squin_2 since its original type has not been modified. During a PKtool simulation, such module is not involved in PKtool analysis and retains its basic behaviour as in an ordinary SystemC simulation.
In the previous example we have seen the instance of a power_module at the most global level, i.e. inside an sc_main function. Nonetheless, a power_module can be also instanced within a hierarchical architecture, in particular as submodule of another module. For example, we can consider a complex module that realizes a polynomial expression, and includes a square_input submodule to compute the square term:

```
SC_MODULE(polynomial)
{
 // internal square_input module

   square_input  sq_term ;
      ...
      ...

 // constructor

 polynomial( sc_module_name): sq_term("sq_term"), ...
 {
   ...
 }

};
```

The submodule is called *sq_term*; in the code there are reported only the instructions related to its instance and construction. If we want to convert *sq_term* into a power_module, it is necessary to modify only the instance instruction:

```
#include "powmod_squin.h"


SC_MODULE(polynomial)
{
 // internal square_input power_module

 POWER_MODULE(square_input)  sq_term ;
      ...
      ...

 // constructor

 polynomial( sc_module_name):  sq_term("sq_term"),...
 {
   ...
 }

};
```

Also in this case, the original module type must be wrapped in the macro POWER_MODULE. The construction and connection instructions are not to be modified. The power_module related to sq_term is indirectly instanced whenever a polynomial module is instanced. When this happens, such power_module is automatically selected for PKtool analysis.
The power_module conversion can be carried out also for a module created dynamically. To show this situation, let us consider a variant of the polynomial class in which sq_term is created dynamically in the constructor body:

```
SC_MODULE(polynomial)
{
 // internal square_input power_module

 square_input*  sq_term ;
     ...

 // constructor

 polynomial(sc_module_name)
 {
   sq_term = new square_input("sq_term");

   // connection instructions
      ...

 };
};
```

In this case, the power_module conversion requires to modify the instructions involving the original type; in particular, the pointer declaration and the construction:

```
#include "powmod_squin.h"


SC_MODULE(polynomial)
{
 // internal square_input power_module

 POWER_MODULE(square_input)*  sq_term ;
     ...

 // constructor

 polynomial( sc_module_name)
 {
   sq_term = new POWER_MODULE(square_input)("sq_term");

   // connection instructions
    ...

 };

};
```

7 POWER MODEL AND STATIC DATA SPECIFICATION

7.1 Introduction

Unlike the settings so far described, the configuration steps c) and e) do not require a realization at code level but are based on a procedure at the beginning of a PKtool simulation. More precisely, such steps are based on an interaction with the command prompt window. The specific details will be shown through a simulation example that involves the power_module squin_1, as instanced in the sc_main function shown in 6.6 .

7.2 Interaction with the command prompt window

When a SystemC simulation is started on the system to which squin_1 belongs, automatically also a PKtool simulation is activated. First of all, this text appears on the command prompt window:

```
-------------------------------------------
    POWER_MODULE: squin_1
-------------------------------------------


OPTIONS FOR SPECIFYING THE POWER MODEL


1: interaction with window
2: reading from configuration file
3: no monitoring


select an option (1, 2, or 3) =
```

The headline declares the name of the considered power_module; all the data that will be communicated regard such power_module. Initially, the user is required to select one of three options identified by the numbers 1, 2, and 3. The first two options have to do with the modalities to communicate the power model data; the third option disables the power_module in the current PKtool simulation. From now on, we will refer to this initial task as *preliminary window menu.*
The option 1 allows to specify the data through an interaction with the command prompt window, whereas the option 2 through a pre-existent text file (configuration file); this latter must be defined according to suitable layout rules. The first time that the power_module is included in a PKtool simulation, it is always convenient to select the option 1. In this way, the configuration file will be automatically created by PKtool with the same data specified in the window interaction. This modality will be further explained in 7.4. Finally, the option 3 should be considered when several power_modules are instanced but only a subset of them is to be monitored for PKtool analysis.
Coming back to our example, let us select the option *interaction with window* by writing 1 in the request sentence and pressing return. As result, this text is displayed:

```
estimation type: energy
model library selected: pk_default_energy_lib
available power models: 9
related numeric indexes: 0 1 2 3 4 21 22 23 41

power model =
```

The first four sentences are information about the enabled model library: estimation type, name, number of power models, related model indexes. These information reflect the fact that the estimation type has been set in terms of energy, as specified in the power_module class (6.4), determining consequently the applicable power models.
The power model is selected through the final request sentence, which asks the user to insert the related numeric index. In this simulation, we might assume a power dissipation modelled by the power model with numeric identifier 3. This model is called *model_3*, and is based on the following formula:

$$\text{Energy} = c \, C_{ap} \, V_{dd}^{2} \, C_{omm}$$

which is derived from the dynamic energy consumption in CMOS technology. In the formula, *c* is a float proportionality coefficient, *Cap* an equivalent capacitance, V_{dd} the applied power supply. These parameters represent static data and must be provided by the user. Comm is the sum of the commutations of all the augmented signals, as occurred during the simulation. Comm is a dynamic data and is automatically computed in the course of the simulation, by means of augmented signal capabilities.
For selecting this power model, the user must specify its numeric index in the request sentence. In this way, the configuration step c) is realized.
Thereafter, this text is displayed:

```
power model:  model_3     numeric index: 3
coefficient  (units) =
```

Now the user is asked to provide the static data of the power model. First of all it is required the proportionality coefficient; as specified, this value must be reported in units. In this example, we might assume a coefficient equal to 3.
In the following of the interaction, there are displayed the request sentences to assign the other static data of the model:

```
power supply  (V) =
```

As concerns the power supply, we might assume 3.3 V.

```
capacitance  (nF) =
```

The equivalent capacitance might be set to 12 nF.
At this point, all the static data have been communicated and the configuration step e) is thus realized. The specification procedure reaches its termination and PKtool is provided with all the information for executing power estimations on squin_1 module. The SystemC/PKtool simulation

resumes its course, continuing with the appearance of an ordinary SystemC simulation. At the end of the simulation, the estimation results will be reported in a suitable text file.
The considered example has shown the simplest case in which only one power_module is instanced. If several power_modules had been instanced, the specification procedure would have been carried out for each of them, following a sequential path based on their construction order.

7.3 Configuration file

At the end of the interaction with the command prompt window, PKtool automatically creates a configuration file whit all the data specifying the selected power models. This file is formatted according to suitable rules, and is reported in the directory where the system project files are located. During a PKtool simulation, a configuration file is created for each power_module.
In our example only one configuration file is created, in reference to the squin_1 power_module. Such file is called *pk_squin_1_cfg*, in compliance with the naming rule:

pk_*pmname*_cfg

where *pmname* is the name of the power_module.
Considering the case of pk_squin_1_cfg, the content of a configuration file is represented by this text:

```
 1)  Configuration file    power_module: squin_1
 2)
 3)  monitored power_module (Y/N)= Y
 4)
 5)  enable window menu (Y/N)= Y
 6)
 7)
 8)
 9)  estimation type: energy
10)  model library selected: pk_default_energy_lib
11)   available power models: 9
12)   related indexes: 0 1 2 3 4 21 22 23 41
13)
14) power model: model_3  numeric index: 3
15)  coefficient (units) = 3
16)  power supply (V) = 3.3
17)  capacitance (nF) = 12
```

In the real configuration file there are no line indexes, here inserted only for a better reference to the text. Lines 9-12 show the information concerning the power model library. Subsequently, it is reported the selected power model (line 14) with the specific static data (lines 15-17). Line 3 declares if the power_module is enabled for PKtool simulations, with reference to the preliminary window menu. More precisely, its value is assigned to Y (yes) if the option 1 or the option 2 is set. In the case the option 3 is selected, this setting is assigned to N (no). Line 5 specifies if the preliminary window menu is to be enabled. This other setting represents an optimization that allows to speed up the initial phase of a PKtool simulation. Normally, this setting is assigned to Y (yes), so enabling the interaction with the preliminary window menu. If it were assigned to N (no), the preliminary window menu would be automatically skipped and the reading from the configuration file would be the option implicitly selected. The settings of lines 3 and 5 will be further discussed in the next section, where they will be simply referred to as line-3 and line-5 settings.

7.4 Reading from configuration file

Let us consider again the preliminary window menu for the power_module squin_1:

```
-------------------------------------------
    POWER_MODULE: squin_1
-------------------------------------------


OPTIONS FOR INSERTING CONFIGURATION AND MODEL DATA


1: interaction with window
2: reading from configuration file
3: no monitoring


select an option (1, 2, or 3) =
```

If the option 2 is selected, the configuration steps c) and e) are realized by reading the data directly from the configuration file, without further interactions with the command prompt window. This brings to an easier specification, representing the best solution when several PKtool simulations are carried out with the same configuration data.
The option 2 can be selected only if the configuration file is already defined. As previously said, the user can avoid to define directly this file because it is automatically created by PKtool when the option 1 is selected. The data reported in the file are the same specified by the user through the interaction with window.
The first time that a power_module is involved in PKtool simulations, the configuration steps c) and e) should be always realized by selecting the option 1. In this way, the configuration file will be created without any user intervention, making the option 2 applicable for the next simulations. Such solution is always correct until the power_module configuration is left unchanged with respect to the data of the steps c) and e). In case the user wants to modify such data and run PKtool simulations under a different configuration, the approach to follow depends on which data have to be modified. More precisely, if the user wants to change only the static data required by the power model, this can be made directly on the configuration file. For the next PKtool simulations it will be still possible to select the option 2, since the new static data will be correctly read from the updated file.
As an example, let us consider the configuration file for the power_module squin_1, as reported in the previous section. If we want to specify different static data, by changing the proportionality coefficient from 3 to 5 and the power supply from 3.3 V to 3.8 V, we should modify the file in this way:

```
 1)  Configuration file    power_module: squin_1
 2)
 3)  monitored power_module (Y/N)= Y
 4)
 5)  enable window menu (Y/N)= Y
 6)
 7)
 8)
 9)  estimation type: energy
10)  model library selected: pk_default_energy_lib
11)  available power models: 9
12)  related indexes: 0 1 2 3 4 21 22 23 41
13)
14) power model: model_3  numeric index: 3
15)  coefficient (units) = 5
16)  power supply (V) = 3.8
17)  capacitance (nF) = 12
```

After saving the file, the new static data will be enabled for the next PKtool simulations.
A different situation takes place when the user wants to change the power model to be applied. In fact, this operation cannot be made through a direct modification of the configuration file. In this case, at least in the first simulation with the new power model, the proper solution would be to select the option 1 and communicate the new power model and its static data via the window interaction. In this way, a new configuration file will be automatically created by PKtool with the new data. As long as the applied power model remains the same, such file will be valid to run PKtool simulations through the option 2 of the preliminary window menu.
The configuration file reports two specifications introduced in the previous section as line-3 and line-5 settings. The line-3 setting allows to select the option 3 of the preliminary window menu directly from the configuration file. This means that if this setting is assigned to N, the power_module will not be monitored in the next PKtool simulations, and the preliminary window menu will be automatically skipped. In order to make the power_module monitorable again, this setting must be assigned to Y by the user.
The line-5 setting allows to skip automatically the preliminary window menu, with the implicit selection of the option 2. This is what happens if this setting is assigned to N. In this way, at the beginning of a PKtool simulation, the user is exempted from the interaction with the preliminary window menu and the explicit selection of the option 2. The typical situation for exploiting the line-5 setting is when a power_module is involved in several simulations in which the option 2 is used. Whenever the configuration file is automatically created by PKtool, the line-5 setting is always assigned to Y (yes).

8 CHARACTERIZATION BASED ON POWER STATES

8.1 General description and application cases

The basic version of a PKtool simulation provides an overall estimation referred to the whole simulation period. However, it is possible to configure more refined analysis, in which to set up partial estimations referred to specific simulation phases. Furthermore, it is also possible to change the applied power model in each of the monitored phases. In other words, different simulation phases can be associated to different power models with respect to model formulation or model data. In PKtool analysis this opportunity may be realized through a *power state configuration*.
A power state defines an operative condition that can be handled as a stand-alone context during a PKtool simulation. In concrete terms, it is possible to achieve partial estimations referred to the time periods in which the considered condition is valid. A power state characterization leads to subdivide the functionality of a module into complementary conditions, each associated to a specific power state. In this way, PKtool analysis may be extended towards the following targets:

a) evaluating the power dissipation in specific time periods.

b) reproducing power management technique based on the run-time change of model data.

c) applying different power models to evaluate different operative conditions.

Briefly discussing these applications, the case a) concerns situations where we want to examine the power dissipated in sub-intervals of the simulation time. In this case, the applied power model could be the same for all the power states representing the system functionality. During a simulation, this power model can be computed several times to estimate the power dissipations referred to each time sub-interval. The case b) is a variant of the case a), with the aim to reproduce power optimization techniques such as dynamic voltage/frequency scaling [5]. In this situation, the power states are associated to a power model that can be subject to variations in its model data, on the basis of the run-time evolution. A power state characterization allows to separate simulation phases in which the model data are fixed. The transition from a power state to another one takes place when the model data are assigned to new values, in consequence of the power optimization strategy. When this happens, a partial power estimation is computed by evaluating the power model using the old model data. In this way, it is possible to cover properly the effects of power optimization techniques in the estimation procedure. In the case c), the target is to differentiate the applied power model on the basis of the operative conditions. This solution may be considered when different working phases are better characterized if associated to different power models.
It is important to underline how power state characterization is a facultative step and should be considered only if the specific analysis targets are to be achieved. In any case, implementing this approach entails an additional overhead in modeling and simulation efforts.

8.2 Realization of a power state characterization

This section shows how to realize a power state characterization when configuring a module for PKtool analysis. A relevant part of this step consists in definitions reported inside the power_module class, which can be illustrated through an example on the square_input power_module.
First of all, it is necessary to define a power state subdivision for the square_input functionality. For this purpose, we could consider three complementary conditions: normal computation, standby condition and overflow condition. These working situations could be associated to three distinct

power states, called respectively ‘normal_st’, ‘standby_st’, and ‘overflow_st’. The declaration of these power states is realized by the following instruction in the power_module class:

```
POWER_MODULE_CLASS(square_input )
{
 ...
 PK_POWER_STATES{standby_st, normal_st, overflow_st};
 ...
}
```

The instruction begins with the macro *PK_POWER_STATES* followed by an enumeration of the power states enclosed in curly brackets. In addition to define the three power states, this declaration sets also a relative order in which, going from left to right, standby_st is the first power_state, normal_st the second and overflow_st the third. In compliance with this order, the power_states are associated to increasing integer identifiers starting from 1: standby_st to 1, normal_st to 2, and overflow_st to 3. It is via such identifiers that the power states can be referenced in some specification tasks.
After that, it is necessary to define a state machine that updates the current power state during a PKtool simulation. From now on, such state machine will be referred to as *powerFSM*. In C++ language a classical way for implementing a state machine is by means of the *switch-case* construct. In PKtool applications, the powerFSM is defined through a different solution based on *updating functions*. More precisely, for each power state a distinct updating function must be defined; this latter describes the rules for determining the next power state when the associated power state is the current one.
In our example, a possible implementation of the updating functions may be the following:

```
// updating function for normal_st

PK_STATE_FC( normal_st )
{
  if ( standby == true ) return standby_st;
  if (input.read( ) >= over_value ) return overflow_st;
  return normal_st;
};


// updating function for standby_st

int stb_cnt;

PK_STATE_FC(standby_st)
{
 while(stb_cnt < 3)
 {
  ++stb_cnt;
  return standby_st;
 };
 if(standby == true)
   return standby_st;
 else
 {
  stb_cnt = 1;
  if(input.read() >= over_value) return overflow_st;
  else return normal_st;
 };
};


// updating function for overflow_st

PK_STATE_FC( overflow_st )
{
 if(standby == true) return standby_st;
 else if(input.read() >= over_value) return overflow_st;
 else return normal_st;
};
```

The title of each function is given by the parameterized macro *PK_STATE_FC* with the name of the related power state as parameter. The body of each function reports the instructions to determine the next power state, specified through the return value. To this end, an updating function can have a direct access to all the public/protected members defined in the module class (in particular the I/O ports), and to all the members defined in the power_module class.
In order to complete the powerFSM implementation, it is necessary to specify when to execute the updating functions. This matter is addressed by defining the *powerFSM sensitivity*. During a PKtool simulation, the updating functions are automatically executed to set the future power state. This task is under the control of the PKtool simulation engine, which calls the function associated to the current power state in suitable triggering events. These latter represent the powerFSM sensitivity.
PowerFSM sensitivity must be specified inside the power_module constructor, with the same instructions used for the sensitivity of ordinary SystemC processes. As general rule, powerFSM sensitivity should consider those events which can cause a power state change, that is the events causing the transitions between the associated operative conditions. For this purpose, it is often sufficient to consider the triggering events for the module functionality.
There are two ways for specifying powerFSM sensitivity. The first solution consists in a dedicated instruction, whereas the second one is based on power_module sensitivity (6.5). In our example, the powerFSM sensitivity is constituted by the positive clock edges, which represent the triggering events for the square_input processes. Using a dedicated instruction, the powerFSM sensitivity is defined in this way:

```
POWER_MODULE_CTOR(square_input)
{

  POWERFSM_SENSITIVITY
  sensitive << clk.pos();

  ...
  ...
}
```

where we have used the macro POWERFSM_SENSITIVITY followed by the specific sensitivity instructions. At this point, the powerFSM is entirely specified, and this completes the power state configuration inside the power_module class.
Alternatively, the powerFSM sensitivity may be implicitly incorporated in the module sensitivity:

```
POWER_MODULE_CTOR(square_input)
{

 PK_MODULE_SENSITIVITY
 sensitive << clk.pos();

 ...
 ...
}
```

If module sensitivity and powerFSM sensitivity were both reported, as in the following example

```
POWER_MODULE_CTOR(square_input)
{

  PK_MODULE_SENSITIVITY
  sensitive << clk.pos();

  POWERFSM_SENSITIVITY
  sensitive << FSM_event;

  ...
  ...

}
```

the explicit instruction always prevails over the module sensitivity. In the considered example, this means that the updating functions will be executed whenever the FSM_event is notified, and only in this case.
When the triggering events of powerFSM sensitivity are the same of the other possible sensitivity specifications (i.e. AIP and cycle-model sensitivity), the simplest solution is to report only the module sensitivity inside the power_module constructor. This may be the case of a module whose functionality is triggered only by clock events.

8.3 Power model specification and configuration file

When realizing a power state characterization, the configuration steps c) and e) are handled similarly to a basic configuration with the extension of the specification tasks for each power state. Considering again the example presented in 7.2, at the beginning of a PKtool simulation the preliminary window menu is displayed on the command prompt window:

```
---------------------------------------------
    POWER_MODULE: squin_1
---------------------------------------------


OPTIONS FOR SPECIFYING THE POWER MODELS


1: interaction with window
```

```
2: reading from configuration file

3: no monitoring


select an option (1, 2, or 3) =
```

The layout and meaning of the text are the same of a basic PKtool simulation. If the power model is specified via window interaction, by selecting the option 1:

```
number of power states: 3
estimation type: energy
model library selected: pk_default_energy_lib
available power models: 9
related numeric indexes: 0 1 2 3 4 21 22 23 41

Initial state =
```

The first five sentences are for informative purpose and report the number of power states, the estimation type and the features of the used model library. The final sentence is a request not present in a basic PKtool simulation, which asks to specify the initial power state through its numeric identifier. This value is used to initialize the powerFSM at the beginning of the simulation. When the powerFSM is triggered the first time, it is executed the updating function related to the initial power state. In our example, we can assume that squin_1 starts from a normal working condition; this latter is associated to the power state normal_st, whose numeric identifier is 2.
The successive tasks concern the specification of the power model and its static data for each power state. This means that the interactive requests described in 7.2 will be repeated for each power state, going from the first one to the last one. In our example, we could suppose an application where different power states are associated to different power models. Continuing in the window interaction, the user must define the power model for the first power state, i.e. standby_st:

```
1st  POWER STATE

 power model =
```

For this power state we might consider the power model *fixed_power* (11.2), based on a constant power dissipation and identified by the index 0.
Thereafter, it is required the power dissipation related to fixed_power:

```
power model:  fixed_power     numeric index: 0
power (mW) =
```

We could assume this quantity equal to 0,2 mW. The power model does not need other static data; the power model specification for the first power state is so completed.
In the following, the power model specification is repeated for the other two power states with the same modalities:

```
2nd  POWER STATE

 power model =
```

The second power state is normal_st and could be associated to the power model model_3, whose numeric identifier is 3. As explained in 7.2, the static data required by such model are the proportionality coefficient, the equivalent capacitance and the power supply:

```
power model:  model_3     numeric index: 3
proportionality coefficient  (units) =

equivalent capacitance  (nF) =

power supply  (V) =
```

As concerns the proportionality coefficient, we could assume the value 3; the equivalent capacitance and power supply might be assigned respectively to 12 nanofarad and 3.3 Volt. The model specification for the second power state is so completed.
Finally, the specification procedure is carried out for the third power state:

```
3rd  POWER STATE

 power model =
```

This power state is overflow_st and could be associated to the power model fixed_power, analogously to standby_st. In the overflow state, the power dissipation required by fixed_power as static data could be assigned to 0.8 milliWatt.
At this point, the window interaction is completed and PKtool is provided with all the elements for estimating the power dissipation in each power state. Like a basic PKtool simulation, a configuration file is automatically created with all the information specified via the window interaction. In this case, the contents of such file are more articulated than a basic format. More in detail, this is the configuration file generated from the window interaction previously illustrated:

```
 Configuration file    power_module: squin_1

 monitored power_module (Y/N)= Y

 enable window menu (Y/N)= Y


number of power states: 3
estimation type: energy
model library selected: pk_default_energy_lib
available power models: 9
related indexes: 0 1 2 3 4 21 22 23 41
initial power state = 2

```

```
15)   1st POWER STATE
16)
17)     power model: fixed_power  numeric index: 1
18)     power (mW) = 0.2
19)
20)
21)   2nd POWER STATE
22)
23)     power model:  model_3     numeric index: 3
24)     proportionality coefficient  ( adimensional units) = 3
25)     equivalent capacitance  (nF) = 12
26)     power supply  (V) = 3.3
27)
28)
29)   3rd POWER STATE
30)
31)     power model: fixed_power  numeric index: 1
32)     power (mW) = 0.8
```

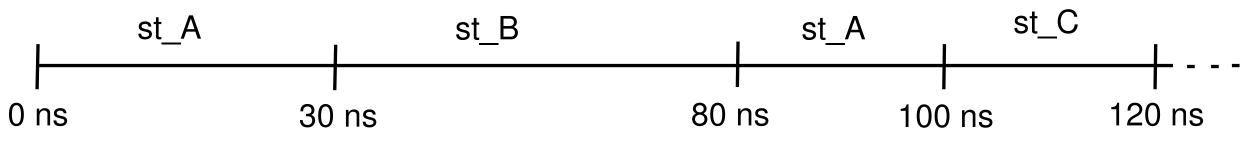
The contents of the first lines are the same of the basic format. In addition, Lines 8-13 report the number of power states and the initial power state. In the following, the power model specifications are reported for each power state (lines 15-18, 21-26, 29-32).
Like a basic configuration, this file can be used for a fast specification of the power models through the option 2 of the preliminary window menu. It is also possible to modify the data reported in the file and read the new configuration from the updated file. The feasibility of this solution depends on which data should be changed. In particular, if the user wants to modify the initial power state and/or the static data required by the power models, these modifications can be made directly on the configuration file. In the successive simulations, it will be possible to use the updated file in application with the option 2 of the preliminary window menu.
The situation is different if the user wants to change the power models associated to the power states, because this cannot be made through a direct modification of a pre-existent configuration file. In this case, at least in the first PKtool simulation with the new configuration, it is necessary to specify the power models via the window interaction. In this way, a new configuration file will be automatically generated with the updated data.

8.4 Behaviour of augmented signals in a power state characterization

The data provided by augmented signals are affected by a power state characterization. During a simulation, such data can be reset and re-computed according to the power state changes, such that their values are referred to the active times of the triggered power states. When a power state change occurs, the data provided by the augmented signals are passed to the power model of the past power state, in order to calculate a partial energy estimation. After that, the augmented signal data are reset and their computation can re-start for the new power state. These operations can be repeatedly carried out whenever a power state change occurs, until the end of the simulation.
In order to show concretely this behaviour, let us consider an example based on this augmented signal:

```
sc_uint_aug<16> bus;
```

Let us suppose this signal belongs to a module configured through three power states; we will refer to these power states simply as *st_A*, *st_B*, *st_C*. Now, let us consider this possible power state evolution during a PKtool simulation:



Let us suppose these assignments for the bus signal:

TIME	SIGNAL VALUE
0 ns	0 (initial value)
10 ns	2
20 ns	5
30 ns	16
60 ns	4
70 ns	7
90 ns	8
100 ns	5
110 ns	10
120 ns	6

For simplicity reasons, in this example we can consider only the augmented signal data related to total commutations. According to the previous evolution, the total commutations provided by the bus signal are the following:

TIME	COMMUTATIONS	TOTAL COMMUTATIONS
0 ns	0	0
10 ns	1	0 + 1 = 1
20 ns	3	1 + 3 = 4
30 ns	3	4 + 3 = 7
60 ns	2	0 + 2 = 2
70 ns	2	2 + 2 = 4
90 ns	4	0 + 4 = 4
100 ns	3	4 + 3 = 7

110 ns	4	0 + 4 = 4
120 ns	2	4 + 2 = 6

The middle column reports the commutations occurred between two contiguous assignments, whereas the most right column the total commutations. When a power state change occurs, the total commutations are reset and their computation re-starts for the new power state. In the above scheme, this behaviour can be observed at the times 60ns, 90ns and 110ns. The past value of the total commutations represents the commutations occurred during the active periods of the past power state. Such data are provided to the power model associated to the past power state, in case such model requires total commutations in its formulation.

9 ANALYSIS RESULTS

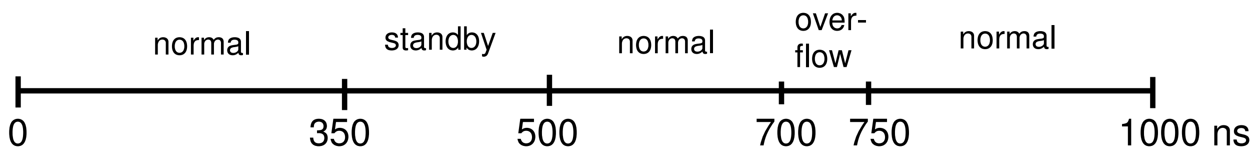
For each power_module, the results of a PKtool simulation are reported in a distinct text file (result file), automatically created at the end of the simulation. Like configuration files, the result files are put in the directory where the system project files are located. A result file contains only the output estimations of the last PKtool simulation; estimation results related to previous simulations are automatically overridden. For a given power_module, the result file is named according to the format:

pk_*pmname*_res

where *pmname* is the name of the power_module. Accordingly, in the case of the squin_1 power_module the result file is called *pk_squin_1_res*.
As an example, we can consider the contents of pk_squin_1_res as could appear at the end of the simulation described in section 7:

```
1) *************** SIMULATION RESULTS ***************
2)
3)
4) overall simulation period: [0 - 650 ns]
5)
6) overall energy estimation: 1.01934e-005 J
7) average power estimation: 15.6821 W
```

The first sentence (line 4) specifies the overall simulation period, in this example supposed equal to 650 ns. Lines 6-7 report the estimation results, given by the energy and average power dissipated in the simulation period. The average power is derived from the ratio between the energy estimation and the simulation period.
What has been shown is a typical result file for a configuration without power states. In the case of a power state characterization, the results are expressed in a more complex format. More precisely, there are reported also the partial estimations referred to the active periods of the power states. To show this format, we can consider the squin_1 power_module in the configuration described in 8.2 and based on three power states. Let us suppose that a PKtool simulation has been executed with the following power state evolution:



In this case, the result file could be defined as follows:

```
1)   *************** SIMULATION RESULTS ***************
2)
```

```
 3)
 4)  simulation period: [ 0 s - 350 ns ]  power state: 2
 5)  partial energy estimation: 5.4e-010 J
 6)  average power estimation: 0.00154 W
 7)
 8)  simulation period: [ 350 ns - 500 ns ]  power state: 1
 9)  partial energy estimation: 3e-011 J
10)  average power estimation: 0.0002  W
11)
12)  simulation period: [ 500 ns - 700 ns ]  power state: 2
13)  partial energy estimation: 3.8e-010 J
14)  average power estimation: 0.0019 W
15)
16)  simulation period: [ 700 ns - 750 ns ]  power state: 3
17)  partial energy estimation: 4e-011 J
18)  average power estimation: 0.0008 W
19)
20)  simulation period: [ 750 ns - 1000 ns ]  power state: 2
21)  partial energy estimation: 4.7e-010 J
22)  average power estimation: 0.00188 W
23)
24)
25)  total state changes: 4
26)
27)
28)  overall simulation period: [0 - 1000 ns]
29)
30)  overall energy estimation: 1.46e-009 J
31)  average power estimation: 1.46e-003 W
```

The text reports the partial estimations referred to the power states triggered during the simulation (lines 4-6, 8-10, 12-14, 16-18, 20-22). For each estimation, it is specified the simulation period and the power state via its numeric identifier. The estimations in standby_st and overflow_st are compliant with the associated power models, based on fixed power dissipations respectively equal to 0.2 mW and 0.8 mW. As concerns the estimations in the normal_st power state, the applied power model depends on the total commutations of the augmented signals.
Line 25 reports the number of power state changes occurred during the simulation. Finally, lines 30-31 show the energy and power estimations related to the whole simulation period. The overall energy estimation is given by the sum of the partial estimations referred to the single power states.

10 SIMULATION TIME SPECIFICATION

During a PKtool simulation, the overall simulation time is measured by PKtool to calculate the average power estimations. In particular, PKtool acquires the simulation time by calling the SystemC function *sc_time_stamp* [3], at the end of the simulation. However, in some situations the overall simulation time is not specified explicitly and the value returned by sc_time_stamp could not be correct with respect to the effective time duration. This could happen when the function *sc_start* [3] is called without a time reference passed as input argument.
To circumvent this issue, PKtool provides the function *pk_set_simtime*, which allows an explicit definition of a simulation time visible only in PKtool analysis. This function should be called in the sc_main, before the declaration of sc_start. The use of pk_set_simtime is shown in the following example:

```
int sc_main ()
{

 sc_core::sc_time sim_time(1000, SC_NS);

 pk_set_simtime(sim_time);

 ...
 ...

 sc_start();

};
```

pk_set_simtime requires an sc_time object as input argument; this latter represents the overall simulation time used in PKtool estimation tasks.

11 DEFAULT MODEL LIBRARIES

11.1 Introduction

PKtool is endowed with some predefined power models that can be directly applied in power estimations. These models are incorporated into two default libraries, *pk_default_energy_lib* and *pk_default_comm_lib*. Such libraries should be considered as dynamic entities that can be enhanced with the addition of new elements.
This section reports an in depth description of the power models contained into the default libraries, excluding the transaction level models; these latter are covered in the specific documentation. Each model will be illustrated underlining model formulation and model data, according to the characterization discussed in 2.1. Model data are dealt with considering the distinction between dynamic and static data. The first category concerns information automatically computed by PKtool and mainly related to time and signals statistics. Conversely, static data must be specified by the user through the procedures shown in sections 7-8.

11.2 Power models in pk_default_energy_lib

At the present time, without considering transaction level models, this library contains nine power models with integer identifiers in the ranges [0 – 4, 21–23, 41].

1) fixed_energy: this power model is associated to the integer index 0, and provides energy estimations (Energy) according to the formula:

$$\text{Energy} = E$$

The model returns a constant energy value as estimation. The required data is the energy value (E), which is expressed in nanojoules and must be provided by the user.

2) fixed_power: this power model is associated to the integer index 1, and provides energy estimations according to the formula:

$$\text{Energy} = PT$$

The model is referred to a constant power dissipation, and the estimation is given by the product between the power (P) and the simulation time (T). The power value is expressed in milliwatts and must be provided by the user; the simulation time is computed by PKtool.

3) model_2: this power model is associated to the integer index 2, and provides energy estimations according to the formula:

$$\text{Energy} = c\ \text{Comm}$$

In this case, the estimation is given by the product between a float proportionality coefficient (c) and the sum of the total commutations of all the augmented signals (Comm). The proportionality coefficient is expressed in nanojoules and must be provided by the user; the Comm term is computed by PKtool.

4) model_3: this power model is associated to the integer index 3, and provides energy estimations according to the formula:

$$Energy = c\, C_{ap}\, V_{dd}^{2}\, Comm$$

which is derived from the dynamic energy consumption in CMOS technology. In the formula, c is a float proportionality coefficient, Cap an equivalent capacitance, V_{dd} the applied power supply, and Comm is the sum of the commutations of all the augmented signals. The proportionality coefficient is expressed in units, the equivalent capacitance in nanofarads, and the power supply in Volt. Such data must be provided by the user. The Comm term is computed by PKtool.

5) *model_4* : this power model is associated to the integer index 4, and provides energy estimations according to the formula:

$$Energy = N_g \left(V_{dd}\, I_{leak}\, T + \frac{1}{2} C_{avg}\, V_{dd}^{2}\, \overline{Comm} \right)$$

In the above expression N_g is the number of gates of the monitored module. The terms in the round bracket stand for the static and dynamic components of the energy dissipated by a single gate. I_{leak} is the average leakage current, V_{dd} is the applied power supply, C_{avg} is the average gate capacitance, $\overline{Comm}$ represents the average commutations per gate. This latter value is estimated by averaging the total commutations of all the augmented signals. Finally, T is the simulation time.
N_g is expressed in adimensional units, I_{leak} in nanoamperes, V_{dd} in Volt and C_{avg} in nanofarads. Such data must be provided by the user. T and $\overline{Comm}$ are computed by PKtool.

The next power models (in the range 21 – 23) are table-based power models [1]. In this case model formulation is given by a discrete representation mapped into a lookup table. The energy values stored in the lookup table are addressed by some compact form of information regarding the module environment.
The association between addressing information and table values is typically determined through a preliminary characterization phase [1]. This latter consists in accurate low-level power simulations of the module, based on input training patterns reproducing somehow the addressing information. The measures coming from these simulations lead to define the energy values stored in the table.
Augmented signals can cover an important role in the handling of the lookup table. In fact, in many cases, the addressing information are referred to signal statistics such as input-output switching activity or signal probability. During a PKtool simulation, such quantities can be extracted only from those signals that have been converted into their augmented counterparts. As a consequence, the application of a table-based power model may require the instance of several augmented signals. For example, if the addressing information were constituted by the average commutations of the input ports, the user should augment all the input ports of the module or an appropriate subset of them.

During a simulation, all the tasks concerning lookup table handling are carried out by PKtool in automatic way. As typical static data, table-based power models require the information to build lookup tables, in particular the discrete values of the addressing information and the stored energy estimations. The power models currently available have been derived from the approaches illustrated in [6-8] .

6) table_1: this power model is associated to the integer index 21, and provides cycle-accurate estimations (2.2). The model is based on a one-dimension lookup table; the stored energy estimations are function of the average Hamming distance (average Hd) between consecutive input vectors. During each simulation cycle, an energy estimation is computed by extracting the table value addressed by the average Hd between the current and the previous input vector. At the end of the simulation, the overall estimation is given by the sum of all the cycle estimations extracted from the table. This model is based on the approach described in [6], where its application is shown for datapath components.
In this context, the average Hd is a float value in the range [0-1] defined by this formula:

$$\frac{\sum_{i=1}^{N} h(\mathbf{s}_i)}{\sum_{i=1}^{N} \text{size}(\mathbf{s}_i)}$$

where N is the number of monitored input ports, h( ) is the Hamming distance between two consecutive port values, and size( ) is the bit size of a single port. The application of this model requires to augment all the input ports of the module, or an appropriate subset of them. This will allow to compute the average Hd values during a simulation.
The user must provide the information necessary to build the lookup table, i.e. the addressing Hd values and the stored energy estimations. These information are specified through the interactive procedure at the beginning of a PKtool simulation. We can now consider the essential details of this task.
First of all, it is required the number of elements stored in the table:

```
number of stored energy values (positive integer) =
```

The user must write the corresponding value. In this example, we can assume a lookup table with 4 elements. Then, there are required the values of average Hd used for addressing the table:

```
Hd values (4 float values) =
```

the user must report a sequence of four float values, separated at least by one blank. These values must be in the range [0-1] and must be written in increasing order. In this example, we could consider this sequence: 0.1 0.3 0.5 0.8 .
Finally, the energy estimations stored in the table must be specified:

```
corresponding energy values ( nJ )  =
```

The user must report a sequence of four float values, separated at least by one blank. The energy estimations are expressed in nanojoules, and their order must correspond with the table indexes previously declared. In our example, if this sequence is reported: 3.4 4.5 4.9 7.8, the association between addressing inputs and energy estimations is the following:

$$
\begin{array}{lcl}
0.1 & \longrightarrow & 3.4 \text{ nJ} \\
0.3 & \longrightarrow & 4.5 \text{ nJ} \\
0.5 & \longrightarrow & 4.9 \text{ nJ} \\
0.8 & \longrightarrow & 7.8 \text{ nJ}
\end{array}
$$

After that, the lookup table is completely specified and the simulation resumes its run-time course. During the simulation, it might happen that the average Hd between two consecutive input vectors does not match with any of the indexes addressing the lookup table. In this case, the energy estimation will be a weighted sum of the energy values associated to the two closest indexes. This means that an average Hd equal to 0.45 will cause the indexes 0.3 and 0.5 to be selected, and the output estimation will be an interpolation between the estimations connected to these indexes.

7) table_2: this power model is associated to the integer index 22, and provides cycle-accurate estimations (2.2). The model is based on a two-dimension lookup table; the stored energy estimations are function of the average Hamming distance (average Hd) and the number of stable zero bits (stable zeros) between consecutive input vectors. At the end of the simulation, the overall estimation is given by the sum of all the cycle estimations extracted from the table. This power model is based on the approach described in [6], and represents a more accurate version of table_1. During each simulation cycle, an energy estimation is computed by extracting the table value addressed by the average Hd and stable zeros between the current and the previous input vectors. Stable zeros are handled as normalized values with respect to the number of input bits, and therefore are represented by float values in the range [0 – 1]. The boundary value 1 corresponds to all the input bits retaining a zero value between two consecutive input vectors. The application of this model requires to augment all the input ports of a module, or an appropriate subset of them. This will allow to compute the average Hd and stable zeros of the input vectors during a simulation. The user must provide all the information necessary to build the lookup table, i.e. the stored energy estimations and the addressing values related to average Hd and stable zeros. These information are provided through the specification procedure at the beginning of a PKtool simulation. We can consider the essential details of this task.
First of all, the user is asked to specify the number of values constituting the grid for average Hd:

```
number of Hd values (positive integer) =
```

In this example, we can assume a grid with 4 values.
Afterwards, the average Hd values are required:

```
Hd values (4 float values) =
```

the user must provide a sequence of four float values written in increasing order. A possible sequence could be: 0.2 0.4 0.6 0.8 . Then, the same data are required for stable zeros:

```
number of stable zeros (positive integer) =
```

We can assume a stable zero grid with 5 values.

```
normalized stable zero values (5 float values) =
```

the user must specify a sequence of normalized values in the range [0 – 1], reported in increasing order. A possible sequence could be: 0.1 0.3 0.5 0.7 0.9 .
At this point, the energy estimations stored in the lookup table are required, with reference to the addressing values of Hd and stable zeros previously specified. The modality used for this step consists in the scanning of the stable zero grid for fixed values of Hd:

```
Hd: 0.2     stable zeros: 0.1  0.3  0.5  0.7  0.9

energy values (nJ) =
```

the user must report a sequence of five float values, representing the energy estimations addressed by the (Hd, stable zeros) couples: (0.2 , 0.1); (0.2 , 0.3); (0.2 , 0.5); (0.2 , 0.7); (0.2 , 0.9). This task is repeated for all the other values of Hd, so to cover all the possible (Hd, stable zeros) couples:

```
Hd: 0.4     stable zeros: 0.1   0.3   0.5  0.7  0.9

energy values (nJ) =
```

…
…

After that, the lookup table is completely specified and the PKtool simulation can resume its run-time course.
The procedure now described could present some inconsistencies, due to (Hd, stable zeros) couples not associable to an energy estimation. In fact, Hd and stable zeros are not independent quantities: high values of Hd means that most bits toggle between two input vectors, so excluding high values of stable zeros. For example, a (Hd = 0.8, stable zeros = 0.9) couple would represent an impossible case, which cannot happen during a simulation. However, the specification procedure is not able to recognize the invalid (Hd, stable zeros) couples. More specifically, coming back to the example, when we have this request sentence:

```
Hd: 0.8     stable zeros: 0.1   0.3   0.5  0.7  0.9

energy values (nJ) =
```

It is anyway necessary to specify a sequence of five float values in order the simulation to continue properly, even if no energy estimation is available for the (Hd, stable zeros) couples with high stable zeros. In this case, the user must specify meaningless energy values for the invalid (Hd, stable zeros) couples, given by negative float numbers. This represents a compulsory rule to mark all the invalid combinations and to allow a correct management of the lookup table. Consequently, a possible sequence for the previous energy values could be 5.3 6.2 -1 -1 -1 .
During the simulation, it might happen that the average Hd and/or stable zeros between two consecutive input vectors do not match any of the specified index couples. In this case, the output

estimations will be given by a weighted sum of the table values associated to the two closest index couples.

8) *table_3* : this power model is associated to the integer index 23, and is derived from the approach described in [7-8]. The model is based on a three-dimension lookup table; the stored energy estimations are function of the average input probability (P_{in}), the average input transition density (D_{in}) and the average output transition density (D_{out}). These signal statistics are formally defined in [2] and [7].
The application of this model requires to augment all the input and output ports of the monitored module, or an appropriate subset of them. At the end of a simulation, the previous statistics are extracted from the augmented input and output ports. Then, the overall energy estimation is determined by the corresponding table value.
The user must provide all the information to build the lookup table, i.e. the stored energy estimations and the addressing values of P_{in}, D_{in} and D_{out}. These information are provided through the specification procedure at the beginning of a PKtool simulation. We can consider the essential details of this task.
At the beginning, the user is asked to specify the clock frequency of the monitored module; this value is necessary for computing D_{in} and D_{out} :

```
clock frequency (Mhz ) =
```

Afterwards, the user is required to specify the value grids for P_{in} and D_{in}. Such quantities are float numbers always included in the interval [0,1], and are subject to the constraint explained in [7]: $D_{in}/2 \leq 1 - 2| P_{in} - 0.5 |$. As a consequence, some (P_{in}, D_{in}) couples are to be excluded in the construction of the lookup table. This control task is automatically handled by the specification procedure, which requires the energy estimations only for valid (P_{in}, D_{in}) couples.
First of all, the user must specify the number of values constituting the grid for P_{in}:

```
number of Pin  values (positive integer) =
```

In this example, we could assume a grid with 5 values.
Afterwards, the P_{in} values are required :

```
Pin  values (5 float values) =
```

The user must report a sequence of five float values in increasing order. A possible sequence could be: 0.2 0.4 0.5 0.6 0.8 .
The same data are required for D_{in}:

```
number of Din  values (positive integer) =
```

We could assume a D_{in} grid with 4 values.

```
Din  values (4 float values) =
```

The values must be written in increasing order. We could assume the following D_{in} sequence: 0.1 0.3 0.5 0.7 .
At this point, the D_{out} grid is to be specified. As observed in [8], the construction of the lookup table is affected by the lack of direct control on D_{out}. In fact, unlike P_{in} and D_{in}, D_{out} depend on the module functionality, which is out of the user's control. As a consequence, D_{out} values cannot be handled as independent parameters and cannot be fixed a priori. Moreover, the D_{out} distribution may not be the same for different (P_{in}, D_{in}) couples.
For addressing this issue, the three-dimension lookup table is organized as a matrix of ordered lists [8]. A matrix element is uniquely identified by a (P_{in}, D_{in}) couple, and is constituted by a list of (D_{out}, energy) couples ordered for increasing D_{out}. Within the specification procedure, the user is asked to specify the (D_{out}, energy) list for each valid (P_{in}, D_{in}) couple. Accordingly, the specification procedure continues in this way:

```
Pin : 0.2      Din : 0.1

Number of Dout values (positive integer) =
```

The user must specify the number of elements constituting the list associated to the couple (P_{in} = 0.2, D_{in} = 0.1). We could assume a list of 4 elements.
After that, it is required to specify the D_{out} values:

```
Dout  values (4 float values) =
```

The user must report a sequence of four float values in increasing order. We could assume a D_{out} sequence given by 0.1 0.3 0.5 0.7 .
Finally, it is required to provide the corresponding energy estimations:

```
corresponding energy values (nJ) =
```

The user must report a sequence of four float values, corresponding to the table elements addressed by the (P_{in}, D_{in}, D_{out}) keys: (0.2, 0.1, 0.1), (0.2, 0.1, 0.3), (0.2, 0.1, 0.5), (0.2, 0.1, 0.7). This completes the specification of the (D_{out}, energy) list associated to the couple (P_{in} = 0.2, D_{in} = 0.2).
In the following, such procedure will be repeated for all the other (P_{in}, D_{in}) couples. In the example, these couples are given by: (0.2, 0.3), (0.4, 0.1), (0.4 0.3), (0.4, 0.5), (0.4, 0.7), (0.5, 0.1), (0.5, 0.3), (0.5, 0.5), (0.5, 0.7), (0.6, 0.1), (0.6, 0.3), (0.6, 0.5), (0.6, 0.7), (0.8, 0.1), (0.8, 0.3) . The (P_{in}, D_{in}) couples (0.2, 0.5), (0.2, 0.7), (0.8, 0.5), (0.8, 0.7) are not considered because they do not satisfy the constraint $D_{in}/2 < 1 - 2| P_{in} - 0.5 |$.
At this point, the lookup table is completely specified and the simulation resumes its run-time course. At the end of the simulation, it could happen that the computed P_{in}, D_{in}, D_{out} do not match with the indexes specified to address the lookup table. In this case, the power estimation will be given by a weighted sum of the table values associated to the two closest indexes.

Operator-based power models provide estimations taking into account the operations occurred during a simulation. At the moment, the default energy library comprises one operator-based power model associated to the integer identifier 41. This kind of models can be enabled or disabled through the macro PK_ENABLE_OPMODELS, defined in the header file pk_settings.h inside the directory src/PKtool/kernel. For enabling/disabling these models, it is necessary to uncom-

ment/comment this macro and rebuild the PKtool library. By default, operator-based models are enabled.
The application of an operator-based model consists of two main phases: run-time sampling of the operations, evaluation of the model. Augmented signals play a primary role in the first phase, because operation sampling is made feasible by means of their capabilities. More precisely, only the operations involving augmented signals can be sampled. For this reason, when configuring a module for PKtool analysis, the user should augment all those signals involved in the operations to be monitored. No sensitivity specification is to be set for enabling operation sampling. In compliance with the limitations specified in 3.2, it is possible to augment input ports and internal nodes for carrying out operation sampling.
The current PKtool implementation allows to sample only the four main arithmetic operations, i.e. addition (+), subtraction (-), multiplication (*) and division (/) . In the future PKtool versions this operator set could be extended.
In general, it is possible to consider several plausible ways to express operation instructions within a SystemC/C++ description. The operation sampling capability has been developed without including all the possible cases, but trying to cover only the most usual forms. To be more explicit, let us consider the following signals:

```
sc_in<int> in1                    // traditional input port
sc_in_aug<int> in2, in3           // augmented input ports
sc_signed; node1                  // traditional internal node
sc_signed_aug; node2, node3       // augmented internal node
tp v;                             // variable/constant of type tp
```

Assuming addition as reference operation, the following instructions can be sampled during a PKtool simulation:

```
in2 + in3; in2.read() + in3.read(); in2 + node1;
in2.read() + node1; in2.read() + node2; in2 + v; in2.read() + v;
in2 + 3;  in2.read() + 3; in2 + in3 + node2;
node1 + node2; node2 + node3; node2 + v; node2 + 3; node2++;
++node2; node2 += v; node2 += 3;
```

The previous instructions are valid also exchanging, where possible, the operand order. Examples of expressions for which operation sampling is not enabled are the following:

```
in1 + in2; in1 + in2.read(); in3 + in2.read();
```

In the current PKtool implementation, operation sampling has been developed for the following augmented signal types:

a) int_aug, short_aug, long_aug, unsigned_aug, float_aug, double_aug, sc_int_aug<n>, sc_uint_aug<n>, sc_signed_aug, sc_unsigned_aug, sc_bigint_aug<n>, sc_biguint_aug<n>.

b) sc_in_aug<T>, where T can be: int, short, long, unsigned, float, double, sc_int<n>, sc_uint<n>, sc_signed, sc_unsigned, sc_bigint<n>, sc_biguint<n>.

c) sc_in_resolved_aug, sc_in_rv_aug<n>.

At the present time, operation sampling is not available for SystemC fixed point types (sc_fix, sc_ufix, sc_fixed …) .

9) *operator_1*: this power model is associated to the integer index 41, and provides energy estimations on the basis of the operations executed during a simulation. Each operator is associated to a constant energy cost, communicated by the user through the specification procedure at the beginning of a PKtool simulation. The overall energy estimation is given by the sum of the occurred operations multiplied by the specific energy costs. As before explained, the sampled operations are those involving the augmented signals instanced in the module.
In analytical terms, this model provides energy estimations according to the formula:

```
Energy =  (add_nb * add_energy)  +  (sub_nb * sub_energy)  +  (mult_nb * mult_energy) +
             + (div_nb * div_energy)
```

Where *op_nb* is the number of occurred op operations, whereas *op_energy* is the energy cost associated to the op operation.
We can consider the essential details of the procedure for specifying the operator energy costs. After selecting the index 41 in the power model request, this text is displayed:

```
power model: operator_1     numeric index: 41
available operators: +  -  *  /

number of enabled operators =
```

The second sentence informs about the available operators through their symbols. The request sentence asks the user to indicate how many operators are to be sampled among the available ones. In this case, such value must be an integer in the range [1 – 4]. If we want to sample additions, multiplications and divisions, we must report the value 3.
At this point, the procedure requires to select the operators to be sampled:

```
enabled operators =
```

The user must specify the operators by means of their symbols, separated at least by one blank. In our example, we should report this symbol sequence: + * /.
A constant energy cost is required for each operator:

```
operator +
energy cost (nJ units) =

operator *
energy cost (nJ units) =
```

operator /
energy cost (nJ units) =

After that, the specification procedure is completed and the simulation resumes its run-time course. At the end of the simulation, in addition to the overall energy estimation, in the result file (9) will be also reported the number of occurrences and the overall energy contribution for each operator.

11.3 Power models in pk_default_comm_lib

At the present time, this library contains five power models with integer identifiers in the range [0– 4].

1) fixed_comm: this power model is associated to the integer index 0, and estimates total commutations (Commutations) according to the formula:

$$\text{Commutations} = C$$

The model returns a fixed commutation value as estimation. The required data is the commutation value (C), which is expressed in adimensional units and must be provided by the user.

2) fixed_rate: this power model is associated to the integer index 1, and estimates total commutations according to the formula:

$$\text{Commutations} = ST$$

This model is referred to a constant commutation rate, and the estimation is given by the product between the commutation rate (S) and the simulation time (T). S is expressed in Hertz and must be provided by the user; the simulation time is automatically computed by PKtool.

3) model_2: this power model is associated to the integer index 2, and estimates total commutations according to the formula:

$$\text{Commutations} = c\text{Comm}$$

The estimation is given by the product between a proportionality coefficient (c) and the sum of the commutations of all the instanced augmented signals (Comm). The proportionality coefficient is expressed in units and must be provided by the user; the Comm term is computed by PKtool.

4) model_3: this power model is associated to the integer index 3, and estimates total commutations according to the formula:

$$\text{Commutations} = N_g\overline{\text{Comm}}$$

N_g is the number of gates of the monitored module; $\overline{\mathrm{Comm}}$ represents the average commutations per gate. This latter quantity is estimated by averaging the total commutations of all the instanced augmented signals. The number of gates is expressed in units and must be provided by the user; the $\overline{\mathrm{Comm}}$ term is computed by PKtool.

5) *model_4* : this power model is associated to the integer index 4, and estimates total commutations according to the formula:

$$\mathrm{Commutations} = \frac{1}{2} N_g H_{avg} f_{ck} T$$

The theoretical aspects and the formal definition of this model are reported in [9]. N_g is the number of gates of the monitored module; H_{avg} is the average bit entropy; f_{ck} is the clock frequency; T is the simulation time. The H_{avg} term is derived from an approximation of the average gate commutations per clock cycle, under the assumption of independence between consecutive values. H_{avg} is determined by PKtool and its computation requires to augment all the input and output ports of the module, or an appropriate subsets of them. N_g and f_{ck} must be provided by the user.

12 REFERENCES

[1] C. Piguet, "Low-Power CMOS Circuit (Technology, Logic Design and CAD Tools)", Taylor & Francis Group, 2006.

[2] F. Najm, "Transition Density, a new measure of activity in digital circuits", IEEE Transaction on Computer-Aided design, vol. 12, pp. 310-323, Feb. 1993.

[3] IEEE Standard SystemC Language Reference Manual, (IEEE Std 1666 - 2011)

[4] B. Stroustrup, "The C++ Programming Language (Third Edition)", Addison Wesley, 1997

[5] J. Rabaey, "Low Power Design Essentials", Springer, 2009.

[6] Jochens, G.; Kruse, L.; Schmidt, E.; Nebel, W."A new parameterizable power macro-model for datapath components", Design, Automation and Test in Europe Conference and Exhibition 1999. Proceedings, 9-12 March 1999 Page(s):29 - 36.

[7] S. Gupta, F. Najm, "Power macro-modeling for high-level power estimation", in Proc. IEEE/ACM Design Automation Conference (DAC), 1997, pp. 365-370 .

[8] M. Barocci, L. Benini, et al. "Lookup table power macro-models for behavioural library components", IEEE Alessandro Volta Memorial Workshop on Low-Power Design, 1999. Proceedings.

[9] M. Nemani, F. Najm, "Towards a high-level power estimation capability", IEEE Transaction on Computer-Aided design, vol. 15, no. 6, pp. 588-598, June 1996.

CONTENTS