7 December 2021

3D-ICE 3.1.0

This document provides a brief summary of the usage of 3D-ICE. This includes illustrative examples of generating the input stack and floorplan files, and the various functions used for printing the results.

User Guide

Table of Contents

1. License and Copyright

Authors:

Arvind Sridhar[1]
Alessandro Vincenzi[1]
Giseong Bak[1]
Martino Ruggiero[1]
Thomas Brunschwiler[2]
Matthias Jung[3]
Éder Zulian[3]
Federico Terraneo[4]
Darong Huang[1]
Luis Costero[1]
Marina Zapater[1]
David Atienza[1]

1 Embedded Systems Laboratory, Department of Electrical Engineering, EPFL, Lausanne, Switzerland.

2 Advanced Thermal Packaging Group, IBM Research Laboratory, Zurich, Switzerland.

3 Microelectronic Systems Design Research Group, University of Kaiserslautern, Kaiserslautern, Germany.

4 Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milano, Italy.

Contact Information:

EPFL-STI-IEL-ESL
Bâtiment ELG, ELG 130
Station 11
1015 Lausanne, Switzerland

Email: 3d-ice@listes.epfl.ch
(SUBSCRIPTION NECESSARY!)
URL: http://esl.epfl.ch/3d-ice

This research has been partially funded by the EC H2020 RECIPE project (GA No. 801137), and the ERC Consolidator Grant COMPUSAPIEN (GA No. 725657), and by the EC EUROLAB-4-HPC CSA Grant (GA No. 800962) through a Short Term Collaboration Project for Dr. Federico Terraneo with EPFL.
This research has been partially funded by the Nano-Tera RTD project CMOSAIC (ref.123618) - which is financed by the Swiss Confederation and scientifically evaluated by SNSF, and the PRO3D project- financed by the European Community 7th Framework Programme (ref.FP7-ICT-248776).

2. What is new in 3.x?

3D-ICE 3.x features several new additions:

- Supported non-uniform discretization for floorplan elements. Floorplan elements can have different cell sizes in thermal modeling and simulation.
- Non-uniform discretization for layers is also supported.
- Supported non-uniform discretization in both steady and transient simulation, and it is compatible with all other functions in the uniform mode.
- Added models of two COTS heat sinks, the HS483 air sink and the cuplex kryos 21606 water block.
- Added Modelica libraries for heat sink modeling to produce FMIs for co-simulation.
- Added grid pitch mapping to FMI interface to allow simulation of heat sinks using a coarser grid.
- Made wrappers for FMI, C++ and Python for the co-simulation interface for pluggable heat sinks.
- Added a pluggable heat sink C interface that makes it possible to extend 3D-ICE with third party heat sink models.
- Added co-simulation support with a separate thermal model for the heat sink.
- Made it possible for the last layer of the 3D-ICE stack to have a different size than the other layers, to support heat spreader modeling and passive air convection packages.

3. What is new in 2.x?

3D-ICE 2.x features several new additions and improvements over the first release of the simulator in September 2010. You would find the following main changes in this package (changes specific to version 2.2 are highlighted in **bold**):

- A new porous medium model has been implemented for the simulation of convective heat transport in interlayer microchannel cavities.
- Various new and enhanced heat transfer geometries have been introduced for the interlayer cavities.

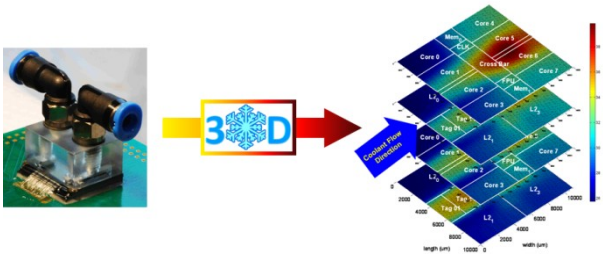
- The user interface has been improved considerably. All the input-output information to the simulator has been transferred to the stack descriptor file, requiring little or no programming of the main C file by the user. This feature brings it closer to SPICE-like circuit simulators that use netlists for circuit descriptions. Hence, 3D-ICE can now function as both a stand-alone thermal simulator as well as a software thermal library. This functionality allows generating thermal **and power** maps of the IC for each die of the 3D stack.
- The user now has the option of doing steady state thermal analysis in addition to transient analysis. This is useful for obtaining steady state temperatures for some average/corner case heat dissipation scenarios, without having to wait for the transients to settle; as well as for generating initial temperature states for certain types of transient simulations.
- A new network interface has been created for online thermal simulations of IC architectures, in tandem with functional emulation on other devices. Under this system, 3D-ICE installed on a computer acts as the server, which communicates with a device emulating the desired architecture via an Ethernet cable. This feature is particularly useful for those who wish to build and test online-thermal-performance-management schemes with Hardware-Software co-simulation of ICs with liquid cooling without actually building the device [6].
- Extensive new documentation has been provided with this release to help simplify the usage of the simulator– this includes this extended user guide, research publications and a new *Doxygen* source code documentation. Doxygen has been used to create an organized html-based visualization of the 3D-ICE software library files and can be accessed by running in the `make doc` command inside the 3D-ICE folder, and then opening `3d-ice/doc/html/index.html` on your web browser.
- Floorplan elements can now be of irregular shapes, as long as they can be represented as a combination of rectangles (defined as an IC element). This feature has been introduced to support the more complex floorplanning that is seen in realistic architectures. **A new technique to align the IC elements on the undelaying thermal cells has been implemented. It allows declaring IC elements smaller than thermal cells as well as posing more than one IC element on the same thermal cell (see section 6.B).**
- **The heat sink model has been extended to support two different techniques to model the dissipation of the heat through the top surface of the IC (see section 6.A.IV).**

To discover the **changes in the library that are related to the implementation and the interface of the software library** please see the CHANGELOG file in the package.

4. Who needs 3D-ICE?

3D-ICE, or "**3D Interlayer Cooling Emulator**", is a Linux based *generic simulation platform* written in C to simulate the transient thermal behavior of 3D IC structures with inter-tier microchannel heat sinks. It is intended for various purposes including, but not limited to:

- Performing thermal analysis of 2D or 3D ICs during early stages of VLSI circuit/architecture design by electronic engineers.
- Simulating run-time and design-time thermal management strategies such as DVFS, dynamic work allocation, variable coolant flow rate, architectural floorplanning etc.
- Testing of microchannel heat-sink performances by microfabrication engineers and heat-sink designers.
- Evaluating accuracies of new and existing heat transfer correlations by experimental heat transfer engineers.



3D-ICE is based on the conventional compact modeling of heat transfer by conduction in solids, and advances a novel compact modeling methodology, called the Compact Transient Thermal Modeling (CTTM), for heat transfer by convection in microchannels. The user is free to use microchannel heat sinks of any dimension with the corresponding heat transfer performance data depending upon the accuracy/speed needs of the user. This simulator is ideal for situations where a quick estimate of chip temperatures is required, when the electronic designer is still iterating between various floorplanning and operating strategies in order to optimize for electronic performance and thermal safety/reliability of the final system.

In addition, the format of inputs, outputs and the problem construction/solving in 3D-ICE have been modeled on the popular compact modeling simulator **HotSpot**, making it easier for users who are familiar with this tool or with conventional circuit simulators. With numerous functions to access a variety of thermal data during the simulation, the user can reach deep into the heart of the thermal simulation, use the data to interface with other (popular or custom) tools and automate any kind of design/run-time optimization algorithms using 3D-ICE. More functionality will be added in the future versions of 3D-ICE to make this interfacing even more automatic and easy.

For more details on the theory and discussions about the accuracy/speed of the modeling technique used in 3D-ICE, please refer to the publications [1] and [2] available with this library.

5. Before you begin

The 3D-ICE library has been written and developed using:

- gcc 7.4.0
- bison 3.0.4
- flex 2.6.4
- blas 3.7.1

Make sure that these tools are installed on your system before compiling and that the corresponding variables in makefile.def point to the respective binary file. Newer versions should be supported as well, unless backward compatibility issues have been introduced in those packages.

Additionally, the C shell is required to build the SuperLU library.

On debian-based (e.g: Ubuntu) you can install them by typing:

```
$ sudo apt install build-essential bison flex libblas-dev csh
```

You can then extract the 3D-ICE sources with:

```
$ unzip 3d-ice-<version>.zip

$ cd 3D-ICE
```

A. Compile SuperLU

To use 3D-ICE, you must also download and compile the SuperLU 4.3 library [4].

A convenience script is provided, you can run it by executing the following command:

```
$ ./install-superlu.sh
```

B. Compile 3D-ICE

Check and edit the `SLU_VERSION` and `SLU_MAIN` variables in `./makefile.def` to make it point to the main folder of SuperLU. Next, select the value of `SLU_LIBS` according to the choice done above when compiling SuperLU (if you used the convenience script above, this should be already set up for you).

You can then compile the 3D-ICE sources with:

```
$ make
```

Now you can find the executable `3D-ICE-Emulator` in the `3d-ice/bin/` folder:this binary file will serve as the thermal simulator application for all your 3D-ICE projects. The `bin` folder will also contain two other binaries `3D-ICE-Server` and `3D-ICE-Client`: these two executable will serve as example of an implementation of the network interface to execute a remote client/server thermal simulation.

C. Pluggable heat sink and co-simulation interface

To use the pluggable heat sink feature, you also need the following depedencies:

- **OpenModelica 1.16.0** or greater (install guide at https://www.openmodelica.org)
  - Do not install any version earlier than 1.16.0
- Pugixml 1.8.1 or greater (`sudo apt install libpugixml-dev`)
- Python 3 header files (`sudo apt install python3-dev`)
- Pkg-config (`sudo apt install pkg-config`)

You can then compile the 3D-ICE Plugin sources with:

```
$ make plugin
```

D. Testing installation

You can then test the 3D-ICE installation with:

```
$ make test
```

6. Overview of 3D-ICE

This chapter provides a general overview of 3D-ICE: its operating principle, the input files required and the conventions and terminology used in writing these input files. In the next chapter, these input files are discussed in detail.

A. Principle of thermal simulation

3D-ICE is based on the compact modeling of heat flow in solids and liquids applied to a 3D-IC structure with microchannel cooling. As quick recap, the structure is divided into cuboidal *thermal cells* based on the discretization parameters you provide. Next, thermal conductance for heat flow through each face of the cuboid is calculated and connected to the neighboring cells at these faces. Also, a capacitance representing the heat capacity of the cell is calculated. Hence, an equivalent electrical RC circuit is created where the temperatures are represented by voltages and the heat flow is represented by the currents in the circuit. A typical thermal cell representing transient conduction in solids is shown below:

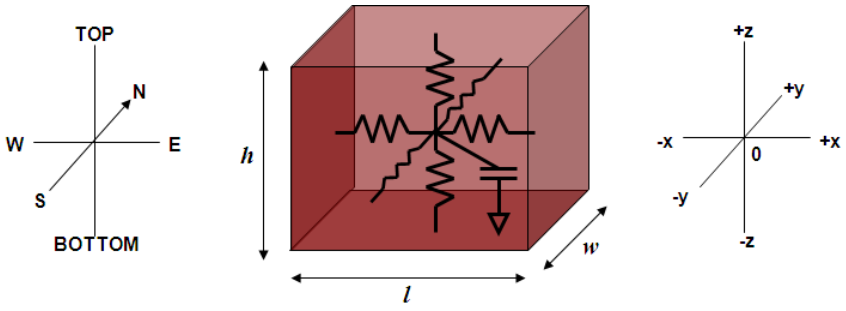


Figure 1: A typical solid thermal cell

In the case of thermal cells representing microchannels or fluidic cavities, while the heat transferred from the cavity walls into the fluid is represented using conventional thermal resistances (or conductances), the convective heat transport in the downstream direction due to mass flow is represented using voltage controlled current sources. There are four different interlayer cooling cavity models implemented in 3D-ICE.

i. Microchannel 4-resistor model

This is the model, which was available in 3D-ICE 1.0, is based on the 4RM-CTTM model described in [1]. Here, the thermal cells are constructed such that in the cavity layer, 2 faces of the thermal cells corresponding to the microchannel completely cover the entire cross-section of the microchannel as shown in Fig. 2. Hence, the discretization along the direction perpendicular to the channels is fixed by the channel geometry. There are four resistances one for each wall of the microchannel representing convective cooling of the wall surfaces. They can be coupled with solid thermal cells on all four directions as shown in Fig. 2. The convective resistance in the top, bottom and the sides can be provided independently into the simulator.

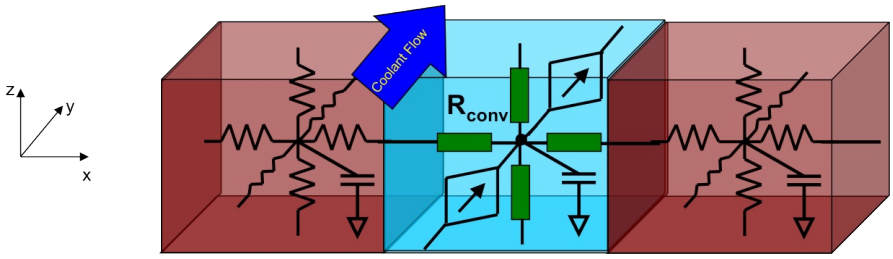


Figure 2: A mc4rm fluid thermal cell adjacent to two solid thermal cells

ii. Microchannel 2-resistor model

This new model for microchannel cavities is based on the 2RM-CTTM model presented in [2]. Here, the dependence of discretization on the channel geometry is completely removed, by homogenizing the entire cavity layer into a single "porous" material. Hence, each cell consists of circuit parameters corresponding to both the flowing coolant as well as the solid walls as shown in Fig. 3. The thermal cell here is two-dimensional because of the elimination of convective resistances on the sides. This model lets the user completely control the granularity of the model by making the thermal cell dimensions (in the x-direction) completely independent of the channel dimensions. Also, when using this model, the top and bottom heat transfer coefficients to be provided in the input files of 3D-ICE must be a projected average of the heat transfer coefficients between the side walls, and the top and bottom walls respectively. For more information about constructing this model, refer to [2].

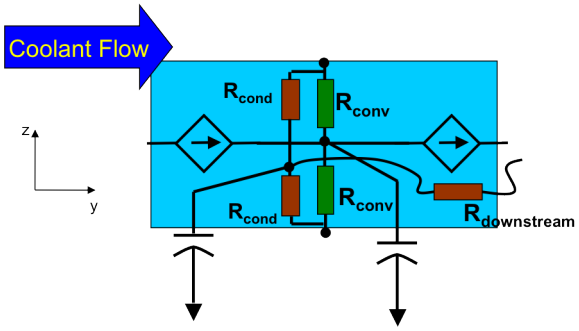


Figure 3: A mc2rm thermal cell

iii. Pinfins in-line

Enhanced heat transfer geometries, such as pin fins can be used in interlayer cooling to improve the heat transfer properties of the liquid-cooled heat sink, when compared to conventional microchannels. Pin fins are also attractive because the can serve as ideal paths for the fabrication of TSVs in 3D ICs. These advantages come at the cost of increased pressure drops. Inline pin fins are one of the two standard types of pin fin HTGs. The top view of such an HTG is shown in Fig. 4.

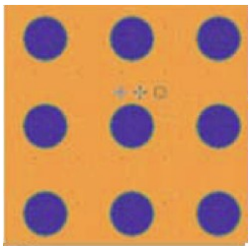


Figure 4: Pin fins inline heat transfer geometry

iv. Pinfins staggered

Staggered pin fins (Fig. 5) is another standard type of pin fin heat transfer geometry. For both these pin fin geometries the 2RM-CTTM modeling method is utilized. The corresponding "porous" thermal cell is illustrated in Fig. 6.

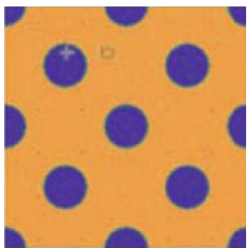


Figure 5: Pin fins staggered heat transfer geometry

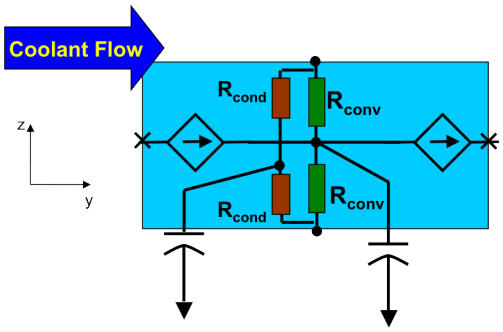


Figure 6: A pin fin thermal cell

In 3D-ICE, the heat transfer coefficients calculations for two specific test cases have been implemented that uses the darcy velocity of the coolant in the cavity. These computations for the convective resistance are based on empirical studies on these geometries performed in [3] and the resulting formulae (where v_{darcy} is the darcy velocity in m/s):

$$\mathrm{HTC}_{\mathrm{inline}} = \frac{2.526 \times 10^{-5}}{\left(\frac{v_{\mathrm{darcy}}}{1\ m/s} + 1.35\right)^{0.64}} + 1.533 \times 10^{-6}$$

$$\mathrm{HTC}_{\mathrm{staggered}} = \frac{2.526 \times 10^{-5}}{\left(\frac{v_{\mathrm{darcy}}}{1\ m/s} + 1.35\right)^{1.52}} + 1.533 \times 10^{-6}$$

B. Inputs to 3D-ICE

3D-ICE accesses all the information needed to emulate a 3D IC from two different types of input files and these constitute a 3D-ICE project:

- **Stack Description File**: The stack description file or the "stack descriptor" is the project file created by the user to describe the 3D IC thermal problem that will be solved. It contains

information about the structure, material properties of the 3D Stack, the description of the various heat sinks in the system, the discretization parameters, analysis parameters, and finally, commands to 3D-ICE for printing out the desired outputs from the simulation. Please refer to Section 6.A when writing this file.

- **Floorplan File**: This file describes the architecture of a 3D IC design as is seen by the thermal model. It contains information about the location and the size of each major logic block in the IC, and the corresponding heat dissipation traces, as a function of time. Each die included in the stack descriptor must have a corresponding floorplan file. Please refer to Section 6.B for instructions on how to write this file.

C. Convention and Terminology

In 3D-ICE, both Cartesian coordinates and cardinal directions are used to describe the location of cells/nodes and direction of heat flow in the structure. The indices of cells/nodes along the *x* direction (WEST-EAST) are sometimes referred to as *columns*, the indices along the *y* direction (SOUTH-NORTH) are sometimes referred to as *rows*, and the indices along the *z* direction (BOTTOM-TOP) sometimes referred to as *layers*. Also, the word *length* is used primarily to refer to dimensions in the *x* direction, the term *width* is used for the *y* direction and the term *height* is used for the *z* direction unless otherwise specified. A typical solid thermal cell along with the coordinate systems used in the library is shown in Fig. 1.

The corresponding model for a liquid cell is shown in Fig. 2. Note that the current sources shown here correspond to the fluid flow in the microchannels. In this library, only flow direction NORTH or *+y* is supported, and hence, when microchannels used in a 3D-IC structure, they are always laid out facing SOUTH-NORTH (with the inlet being at the southern end and the outlet at the northern end of the channels). Hence, the northern edge of the IC is expected to be the hottest and the southern end of the IC is expected to be the coldest in any analysis. It is up to you to decide your floorplanning of the ICs accordingly.

The syntaxes used in this document for describing how these input files must be written are based on the following convention:

`[: … :]`	POSIX characters class
`… | …`	OR- either of the two elements must be used
`[ … ]?`	an optional element
`[ … ]+`	one or more of this element must be used
`[ … ]*`	zero or more of this element must be used

Within the files, **keywords must be written in low case** and all the white spaces belonging to

```
[:space:]
```

will be skipped during the parsing. Identifiers (referred to as ID) must match the following expression:

```
[:alpha:] [ _ | [:alnum:] ]*
```

Floating points values (referred to as DVALUE) must belong to

```
[+|-]? [:digit:]+ [ \. [:digit:]+ [ [e|E] [+|-]? [:digit:]+ ]? ]?
```

Please refer to the flex sources in `3D-ICE/flex` for more details. It is also possible to insert comments at the end of a line (`//`) or to comment an entire block (`/* ... */`) of the input files (similar to C or C++).

7. Creating a 3D-ICE project

As mentioned in Section 6.B, a 3D-ICE project consists of writing a stack descriptor file and one or more floorplan files. See the example Stack Description File and Floorplan files providedin the `./bin` folder for reference.

A. Stack Description File

The stack description file (*.stk) is a netlist that specifies all the physical and geometrical properties of the 3D-IC for the simulation. The extension of the file is not relevant-it will beparsed independent of its presence or content.

The stack description file contains EIGHT main sections (mandatory and optional) and they must be declared in this order:

1. Materials
2. Heat sink
3. Liquid-cooled cavity
4. Layers
5. Dies
6. Dimensions
7. Stack
8. Analysis options
9. Output instructions

The following description of each of these sections is NOT in the above-mentioned order for ease of presentation and coherence.

i. Materials

The first section of the file contains the list of materials and their properties to be used in the simulation. At least one material must be declared. Materials are declared with the syntax,

```
material MATERIAL_ID :

        thermal conductivity     DVALUE ;

        volumetric heat capacity DVALUE ;
```

where

- `MATERIAL_ID` is a unique identifier to refer to this material,
- `thermal conductivity` is expressed in W/µm K ,
- `volumetric heat capacity` is expressed in J/µm^{3}K.

Materials declared here but not used in the following sections (channel, dies or stack) will be reported with a warning message (`stderr`).

Example

```
material SILICON :
        thermal conductivity     1.300e-04 ;
        volumetric heat capacity 1.628e-12 ;
```

ii. Layers

A layer is a horizontal section of the stack with a given thickness and made by a given material. A layer declared in this section can be further cited/used when declaring the sequence of stacked elements in the 3D IC later in the file. Layers declared in this optional section of the file can be referred only in the stack section but not when declaring a die. Layers can be declared with the following syntax:

```
layer LAYER_ID :

        height        DVALUE ;

        material      MATERIAL_ID ;
```

where

- `LAYER_ID` is a unique identifier to refer to this layer,
- `height` is expressed in µm,
- `MARIAL_ID` is the (previously declared) identifier of the material composing the layer.

Example

```
layer PCB :
        height 10 ;
        material BEOL ;
```

iii. Dies

A die is a group of layersstacked together to form a single entity that is used when declaring the sequence of stacked elements of the 3D IC later in the file. This can represent an actual IC die in the stack. You can declare multiple dies, and use a single die multiple times during the stack description. The Dies section is mandatoryand must contain at least one die element. A die must contain one *source* layer (the term *source* layer is used to denote those layers of the stack which contain active electronic components, and hence, provide the heat source for the simulation) and zero or more passive layers. The source layer can be placed at any location in the stack of layers in a die.

```
die DIE_ID :

        [layer  IVALUE MATERIAL_ID ; ]*

        source IVALUE MATERIAL_ID ;

        [layer  IVALUE MATERIAL_ID ; ]*
```

where

- `DIE_ID` is the unique identifier used to refer to the declared die,
- `IVALUE` is the height of the layer (in µm),
- `MATERIAL_ID` is the (previously declared)identifier of the material composing the layer.

The order of the layers within the die reflects their vertical disposition in the 3D IC, i.e., the first layer declared is the top most layer in the die (closer to the ambient) while the last one is the one at the bottom (closer to the PCB). Two examples of die declaration and their illustrations (not to scale) are shown below.

Example

```
die TOP_IC:
       source 2 SILICON;
       layer 50 SILICON;
```

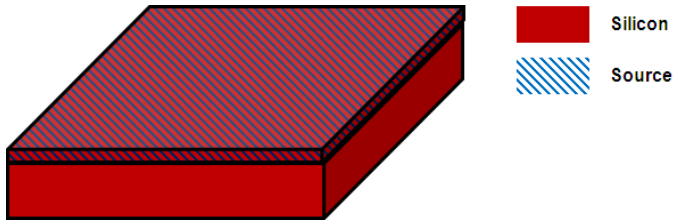


Figure 7: A 2-layer die

```
die BOTTOM_IC:
       layer 10 BEOL;
       source 2 SILICON;
       layer 50 SILICON;
```

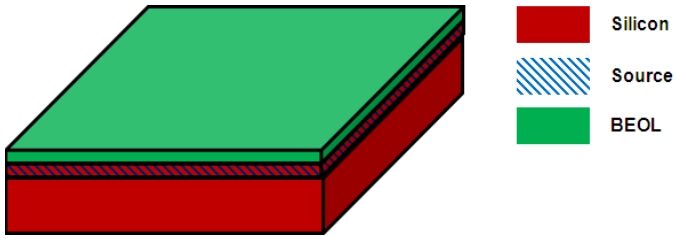


Figure 8: A 3-layer die

iv. Heat Sink

This is an optional section to model convective heat exchange with the surrounding environment and heat dissipation of the IC through the PCB. All the faces of the 3D IC stack are modeled as adiabatic walls by default. When the top or bottom Heat Sink is specified, the corresponding surface of the stack is connected to the ambient via a thermal resistance.

The syntax to add a connection between the IC top and the ambient to simulate convective heat exchange is:

```
top heat sink :

       heat transfer coefficient DVALUE ;

       temperature DVALUE ;
```

The syntax to add a connection between the IC bottom and the ambient to simulate heat exchange through the PCB is:

```
bottom heat sink :

        heat transfer coefficient DVALUE ;

        temperature DVALUE ;
```

where

- `heat transfer coefficient` of the heat sink is expressed in W/µm^{2}K,
- `temperature` is the ambient temperature expressed in K.

Both syntaxes connect the nodes of the 3D-IC directly to the environment, without adding thermal cells.

Example

```
top heat sink:
       heat transfer coefficient 1.0e-7;
       temperature 300;
```

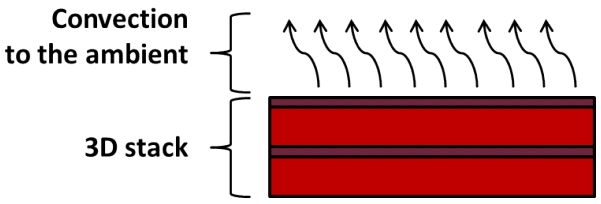


Figure 9: Top heat sink

Pluggable heat sinks

3D-ICE starting from version 3.0.0 supports a pluggable heat sink interface that allow it to perform co-simulation with a separate heat sink model. Refer to the chapter "Co-simulation and plugin interface" for more information about co-simulation in 3D-ICE.

The syntax to add a pluggable heat sink is:

```
top pluggable heat sink :

        spreader length DVALUE, width DVALUE, height DVALUE ;

        material MATERIAL_ID ;

        plugin PATH [, ARGS] ;
```

where

- `length, width, height` are the dimensions of the heat spreader in µm,
- `MATERIAL_ID` is the spreader material
- `PATH` is the path to the heat sink plugin
- `ARGS` is an optional string of arguments passed as-is to the plugin (like the command line arguments of an executable program)

Example

```
top pluggable heat sink:
       spreader length 2e4,
                width 2e4,
                height 1e3;
       material  COPPER;
       plugin "plugin.so", "plugin-args";
```

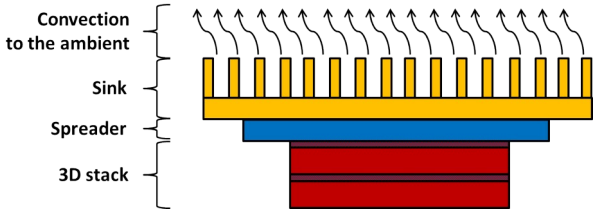


Figure 10: Pluggable heat sink

v. Liquid-cooled cavity

The Liquid-cooled cavity(henceforth referred to simply as "cavity") section provides3D-ICE information about the interlayer liquid-cooled heat sink. Since in a given 3D IC design, all interlayer cavities are designed identically, only one declaration of (and one type of) cavity is allowed. As mentioned earlier, there are four types of cavities in 3D-ICE and the flow of coolant is always the SOUTH-NORTH or *+y* direction. This section can be omitted for simulating a 3D IC without liquid cooling (solid only).

Microchannel 4-resistor model

```
microchannel4rm :

        height DVALUE ;

        channellengthDVALUE ;

        wall lengthDVALUE ;

        [first wall length DVALUE ; ]?

        [last  wall length DVALUE ; ]?

        wall material MATERIAL_ID  ;

        coolant flow rate DVALUE ;

        coolant heat transfer coefficient [ DVALUE |  side DVALUE ,
                                                      top  DVALUE ,
                                                      bottom DVALUE ] ;

        coolant volumetric heat capacity DVALUE ;

        coolant incoming temperature DVALUE ;
```

where

- `height` (in µm) corresponds to the height of the microchannel layer. This must exactly correspond to the microchannel height from the surface of the top wall to the surface of the bottom wall because of the 4RM-CTTM and 2RM-CTTM modeling requirements. Solid walls bounding the top and bottom faces of the microchannel would constitute new layers that should be declared separately,

- `channellength` and `walllength` are the cross sectional lengths of the channel and the wall (in the *x* direction, all in µm). During the discretization of the system (see the Dimensions section for more details), 3D-ICE automatically starts with a wall at the eastern most end of the layer, and alternate channels and walls along the *x* direction. But the user must ensure that dimensions are such that the layer always ends with a wall (in other words, the number of columns must always be an odd number),
- `first wall length` and `last wall length`(in µm) are optional properties that represents the length of the western-most (the first column) and eastern-most (the last column) walls. This option has been included since, during the fabrication of microchannels on the back of the substrate, although the etching mask pattern is predominantly regular, there are chances of irregularities at the ends, or there might be a deliberate use of different dimensions for the first and the last wall to preserve uniformity and symmetry of heat transfer coefficient. If one of these two dimensions(or both) is not declared, then the `wall length` will be used at its corresponding location,
- `MATERIAL_ID` is the identifier of the material composing the walls (the ID must be previously declared in the materials section),
- `coolant flow rate` is expressed (in ml/min) and it refers to the volume of coolant flowing per unit time per channel layer (cavity) in the stack. If you have multiple layers of microchannels, the total flow rate must be divided by the number of channel layers and given as a single input,
- `coolant heat transfer coefficient` is the Heat Transfer Coefficient of convective heat removal from the walls into the coolant (in $W/\mu m^2K$). It is possible to specify a single value of HTC for all the wetted surfaces of the microchannel or specify three different values- one for the *side* wall surfaces, one for the *top* and one for the *bottom* wall surface of the microchannel.
- `coolant volumetric heat capacity` is expressed in $J/\mu m^3K$,
- `coolant incoming temperature` is the inlet coolant temperature expressed in K.

Example

```
microchannel4rm:
       height 100 ;
       channel length 50;
       wall length 50;
       first wall length 25;
       last  wall length 25;
       wall material SILICON;
       coolant flow rate 42;
       coolant heat transfer coefficient
                          side 2.7132e-8,
                           top 5.7132e-8,
                        bottom 4.7132e-8;
       coolant volumetric heat capacity
                               4.172e-12;
       coolant incoming temperature
                                     300;
```

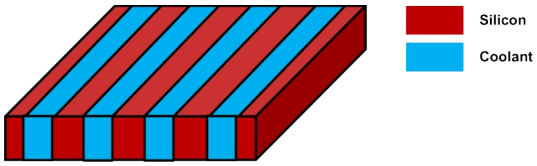


Figure 11: Microchannel cavity layer

Microchannel 2-resistor model

```
       microchannel2rm :

              height DVALUE ;
```

```
channel length DVALUE ;

wall length DVALUE ;

wall material MATERIAL_ID  ;

coolant flow rate DVALUE ;

coolant heat transfer coefficient [ DVALUE |  top  DVALUE ,

                                                bottom DVALUE ] ;

coolant volumetric heat capacity DVALUE ;

coolant incoming temperature DVALUE ;
```

Note that the only change from microchannel-4rm model is the omission of the first- and the last-wall widths and the side-way heat transfer coefficient, as was described in Section 5.A.ii. Also, remember that these heat transfer coefficients are effective "projections" of the heat transfer coefficients on the top and the bottom surfaces of the cavity. If you have the top, bottom and side heat transfer coefficients (HTCs) for a microchannel structure, you can compute these effective HTCs using the following formulae [2]:

$$\mathrm{HTC}_{\mathrm{top,eff}} = \frac{\mathrm{HTC}_{\mathrm{top}} \times \mathrm{channel\,width} + \mathrm{HTC}_{\mathrm{side}} \times \mathrm{channel\,height}}{\mathrm{channel\,pitch}}$$

$$\mathrm{HTC}_{\mathrm{bottom,eff}} = \frac{\mathrm{HTC}_{\mathrm{bottom}} \times \mathrm{channel\,width} + \mathrm{HTC}_{\mathrm{side}} \times \mathrm{channel\,height}}{\mathrm{channel\,pitch}}$$

Example

```
microchannel2rm:
     height 100 ;
   channel length 50;
     wall length 50;
 wall material SILICON;
  coolant flow rate 42;
coolant heat transfer coefficient
               top 5.5698e-8,
            bottom 5.0698e-8;
 coolant volumetric heat capacity
        4.172e-12;
   coolant incoming temperature
                          300;
```

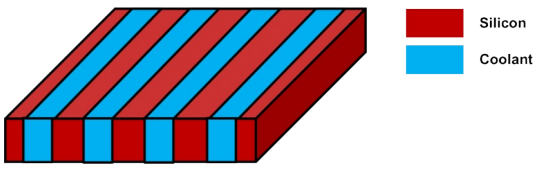


Figure 12: Microchannel cavity layer

Pinfins

```
pinfin:

       height DVALUE ;

       pin diameter   DVALUE ;

       pin pitch     DVALUE ;

       pin distribution [inline | staggered] ;

       pin material MATERIAL_ID ;
```

```
darcy velocity DAVLUE ;

coolant volumetric heat capacity DVALUE;

coolant incoming temperature  DVALUE;
```

where

- `height` (in µm) corresponds to the height of the pinfin layer. This must exactly correspond to the height of the pins in accordance with the 2RM-CTTM model,
- `pin diameter` and `pin pitch` are diameter and the pitch (all in µm) of the pins. 3D-ICE uses this information to compute the pin density and the porosity of the model,
- `pin distribution` defines the type of pinfin distribution- `inline` for inline (Fig. 4) and `staggered` for staggered (Fig. 5),
- `MATERIAL_ID` is the identifier of the material composing the pins (the ID must be previously declared in the materials section),
- `darcy velocity` is expressed (in µm/sec) and it refers to the darcy velocity in the cavity. This is used instead of the coolant flow rate because darcy velocities are more commonly measured quantities in experiments and CFD simulations due to their geometry independence, and more useful while expressing the heat transfer coefficient correlations in empirical studies.

Example (Pinfins inline)

```
pinfin:
       height 100 ;
       pin diameter   50 ;
       pin pitch     100 ;
       pin distribution inline ;
       pin material silicon ;
       darcy velocity 1.1066e+06 ;
       coolant volumetric heat capacity
       4.172638e-12 ;
       coolant incoming temperature
       300.0 ;
```

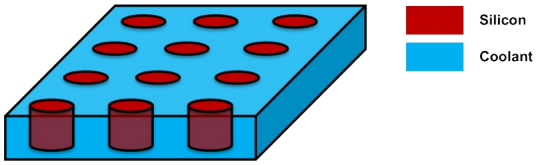


Figure 13: Pinfin inline cavity layer

Example (Pinfins staggered)

```
pinfin:
       height 100 ;
       pin diameter   50 ;
       pin pitch     100 ;
       pin distribution staggered ;
       pin material silicon ;
       darcy velocity 1.1066e+06 ;
       coolant volumetric heat capacity
                          4.172638e-12;
       coolant incoming temperature
       300.0 ;
```

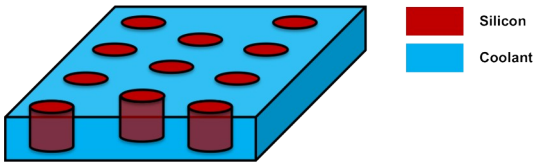


Figure 14: Channel layer

Note: If neither the conventional nor the liquid-cooled heat sink is declared in the Stack Description File, then the parsing will end with a warning message. Note that this would cause the temperatures to blow up unbounded with time in the presence of non-zero heat sources.

vi. Dimensions

This section of the Stack Description File declares the *x-y* dimensions of the entire chip and the discretization sizes for the thermal cells (all in µm).

```
dimension :

        chip length DVALUE , width DVALUE ;

        cell length DVALUE , width DVALUE ;

        [non-uniform  DVALUE ; ]?
```

Uniform mode

By default or when non-uniform mode is disabled (`non-uniform 0;`), 3D-ICE will discretize the chip's uniformily. In this case, the entire chip is discretized based on the same cell length (along *x* direction) and width (along *y* direction) values. The discretization along the *z* direction is not specified, since the height of a thermal cell is taken to be the same as the height of the layer in which it exists as shown for a layer in Fig. 16. However, if you want a finer discretization than that along the *z* direction, then you will have to split the layer into multiple layers of the same material stacked on the top of each other in the declaration of die/stack, as shown in Fig. 17 (here $h_1+h_2=h$).

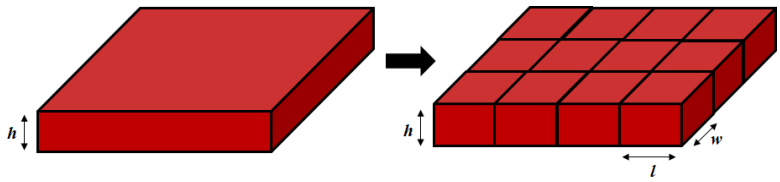


Figure 15: Discretization of a single layer

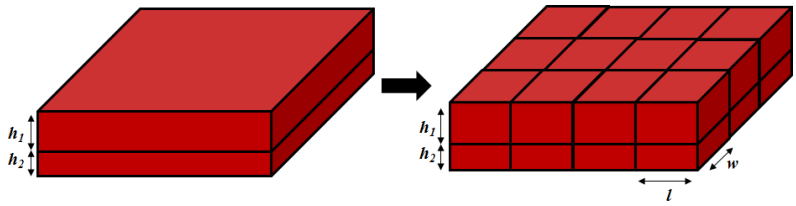


Figure 16: Discretization of a single layer split into 2 layers

Example

```
dimension :
        chip length 10000, width 10000;
        cell length 50, width 50;
```

If the microchannel 4-resistor model is used, the cell length is determined solely based on the cross sectional dimensions of channel and wall in the Channel section of the Stack Description File. This is because the 4RM-CTTM modeling used in 3D-ICE requires that the entire cross section of the microchannel be a part of the thermal cell. This means that the cell length at some position along the *x*

direction in the model is dependent upon the channel or the wall cross sectional width at that point. Note that given different values of `channel length`, `wall length`, `first wall length` and `last wall length`, the discretization along this direction will be non-uniform in the simulator. For all other liquid-cooled cavity types, the cell length and cell width are necessary and sufficient to create a uniform discretization of the entire structure.

For the purpose of illustration, the discretized domain corresponding to the stacked structure built in Fig. 15 is shown in Fig. 18. Note that the `cell length` must still be declared in all cases, in spite of the fact that it will be ignored during the parsing when microchannel 4-resistor model cavity is used.

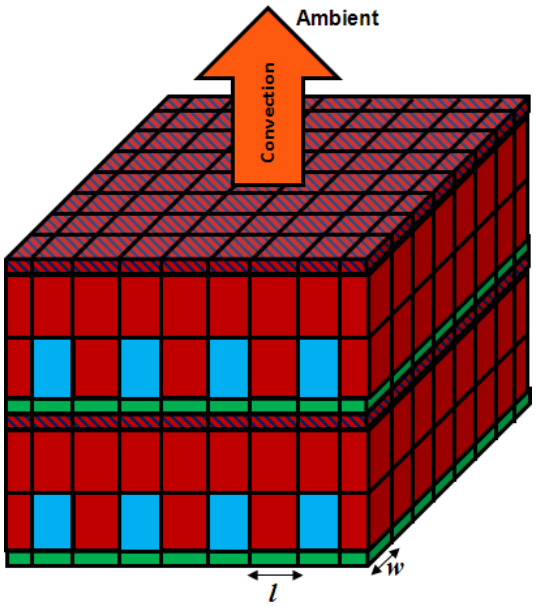


Figure 17: Discretized computational domain for the stack shown in Fig. 15

Non-uniform mode

When non-uniform mode is enabled (`non-uniform 1;`), 3D-ICE will discretize the chip based on the cell length (along *x* direction) and width (along *y* direction) values. For instance, 200x200 in the following example. The default discretization granularity can be overwritten later in the stack and floorplan's definition.

Example

```
dimension :
        chip length 10000, width 10000;
        cell length 50, width 50;
        non-uniform 1;
```

Contrary to the uniform mode, the discretization level of each layer strictly follows its own discretization setting and will not be overwritten by the microchannel's channel length setting. Note that due to the complexity of liquid cooling, the microchannel layer still follows the same discretization method as the uniform mode.

Problem Complexity

Remember that the dimensions of the chip, together with the dimensions of the thermal cells will directly affect the performance of the simulation. Indeed, the number of cells (nodes) influences both the amount of memory used by the library and the time needed to solve the linear system[1]. The main computational effort of the simulator is incurred during the execution of SuperLU and `blas` libraries. Specifically, the LU factorization of the system matrix is the most time/memory intensive. Hence, for large problem sizes, the availability of memory must be ensured to prevent this step from failing during the simulation.

vii. Stack

This section builds the vertical structure of the stack. The stack is composed of Dies (as previously declared) and layers and/or channels.

```
stack :

       [  layer   LL_IDLAYER_ID [discretization DVALUE DVALUE]? ;     |

       channel CC_ID ;                          |

       die     DD_ID  DIE_ID floorplan "PATH" [discretization DVALUE DVALUE]?; ]
       +
```

where

- `LL_ID`, `CC_ID` and `DD_ID` are identifiers used to name the stack elements and they can be used in the simulator code to refer to the corresponding element. They must be unique for each element,
- `LAYER_ID` is the identifier of a layer (as previously declared).
- `DIE_ID` is the identifier of a die (as previously declared) and `PATH` is the path to the floorplan file. This floorplan will be placed on the declared source layer in the definition of the die. The floorplan files contain information of the location and power dissipation activity of various floorplan components for the given die (see the description of Floorplan Files for more details). The same `DIE_ID` can be used multiple times (with different identifiers `DD_ID`) in a stack with the same or different floorplans, if identical/similar dies exist in a single IC.
- `[discretization DVALUE DVALUE]?` is the optional discretization level for layers or die. This argument only takes effect in the non-uniform mode. In the non-uniform mode, If the discretization level is not specified, the layer or die will take default values from the dimension setting. Otherwise, the defined discretization values will overwrite the default one derived from the dimension setting. In this way, the user can have a fully custom discretization design for different layers.

Note that the keyword “channel” is used to specify the cavity in the stack- irrespective of which type of interlayer cavity (microchannels or pin fins) was declared in the stack descriptor.

The above grammar for the stack section is not the representative of the real syntax that is supported and is only given for simplicity. For instance, the three main elements (layer, dieand channel) that compose the stack can be declared in any order butthe final stack sequence must satisfy the following constrains:

- there must be at least one die
- it cannot begin or end with a cavity (i.e., liquid-cooled cavities can't be the bottommost or the topmost layers in a stack)
- there cannot be two consecutive cavities
- cavities can be used only if previously declared
- layers are optional

Declaring a layer in a stack is not mandatory but we left this option to support stacks with irregular patterns of dies and channels or to build auxiliary layers (such as a bonding layer).

As in the case of defining dies, the final sequence of stack elements reflects their vertical disposition. The first stack element corresponds to the topmost element while the last element declared is closest to the PCB.

Example: uniform mode

```
material SILICON :
       thermal conductivity     1.30e-4;
       volumetric heat capacity 1.628e-12;
material BEOL :
       thermal conductivity     2.25e-6;
       volumetric heat capacity 2.175e-12;

top heat sink:
       heat transfer coefficient 1e-07 ;
       temperature 300 ;

microchannel4rm:
       height 100 ;
       channel length 50;
       wall length50;
       first wall length 25;
       last  wall length 25;
       wall material SILICON;
       coolant flow rate 42;
       coolant heat transfer coefficient
                    top 5.7132e-8,
                    bottom 4.7132e-8;
       coolant volumetric heat capacity
                    4.172e-12;
       coolant incoming temperature 300;

layer PCB:
       height 10 ;
       material BEOL ;

die TOP_IC:
       source 2 SILICON;
       layer 50 SILICON;
```

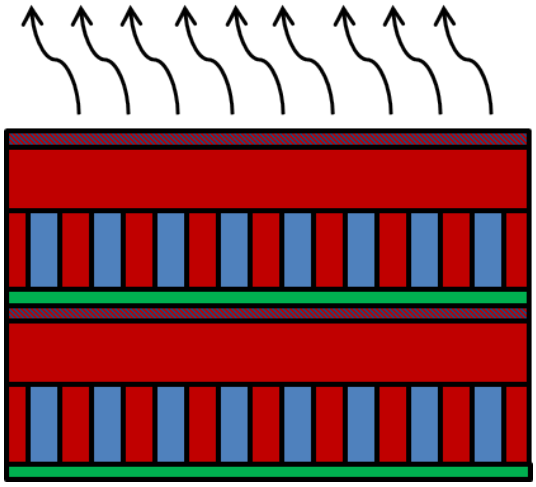


Figure 18 (a): Complete stack

```
die BOTTOM_IC:
       layer 10 BEOL;
       source 2 SILICON;
       layer 50 SILICON;

dimension :
       chip length 10000, width 10000;
       cell length 50, width 50;

stack:
       die     MEMORY_DIE     TOP_IC    floorplan "./mem.flp";
       channel TOP_CHANNEL;
       die     CORE_DIE       BOTTOM_IC floorplan "./core.flp";
       channel BOTTOM_CHANNEL;
       layer   BOTTOM_MOST    PCB ;
```

Example: non-uniform mode

```
material SILICON :
       thermal conductivity     1.30e-4;
       volumetric heat capacity 1.628e-12;
material BEOL :
       thermal conductivity     2.25e-6;
       volumetric heat capacity 2.175e-12;

top heat sink:
       heat transfer coefficient 1e-07 ;
       temperature 300 ;

layer PCB:
       height 10 ;
       material BEOL ;

die TOP_IC:
       source 2 SILICON;
       layer 50 SILICON;

die BOTTOM_IC:
       layer 10 BEOL;
       source 2 SILICON;
       layer 50 SILICON;

dimension :
       chip length 10000, width 10000;
       cell length 50, width 50;
       non-uniform 1;

stack:
       die     MEMORY_DIE     TOP_IC    floorplan "./mem.flp" discretization 3 3;
       channel TOP_CHANNEL;
       die     CORE_DIE       BOTTOM_IC floorplan "./core.flp"discretization 5 5;
       channel BOTTOM_CHANNEL;
       layer   BOTTOM_MOST    PCB ;
```

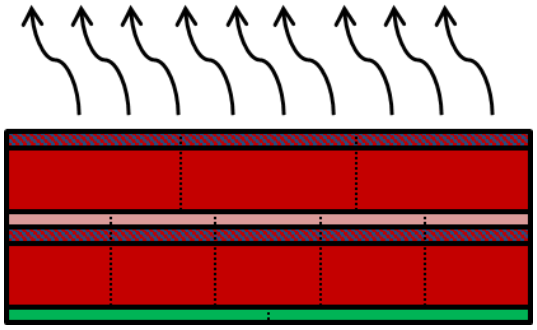


Figure 18 (b): Complete stack and discretization

viii. Analysis options

In 3D-ICE either transient or steady state simulations can be performed.

```
       solver:
```

```
[steady | transient step DVALUE, slot DVALUE] ;

initial temperature DVALUE ;
```

where

- `steady` and `transient` indicate steady state and transient analysis respectively,
- `step` corresponds to the internal stepping time (in sec) to be used in the transient simulation in 3D-ICE (note that while performing co-simulation this is also the step at which the heat sink model will be simulated),
- `slot` is the slot time (in sec) for which each power value in the floorplan file lasts for in the transient simulation (see Section 6.B). This value must be greater or equal to the stepping time value,
- `initial temperature` denotes the initial temperature (in K) for the simulation.

Example (steady state)

```
solver :
        steady;
        initial temperature 300.0;
```

Example (transient)

```
solver :
        transient step 0.02, slot 0.2;
        initial temperature 300.0;
```

ix. Output Instructions

A variety of outputs can be obtained from 3D-ICE simulations, a temperature and power. In this last section of the stack descriptor, various output instructions can be provided for this purpose. The syntax for these instructions follow the model of conventional circuit simulators.

```
output:

        [ T( VER_ID, DVALUE, DVALUE, "PATH" [, INSTANCE_ID]? ) ; |

        Tflp( DD_ID, "PATH", OUTTYPE_ID [, INSTANCE_ID]? ) ; |

        Tflpel( DD_ID.FLPEL_ID, "PATH", OUTTYPE_ID [, INSTANCE_ID]? ) ; |

        Tmap( VER_ID, "PATH" [, INSTANCE_ID]? ) ; |

        Pmap(DD_ID, "PATH" [, INSTANCE_ID]? ) ;                  |

        Tcoolant( CC_ID, "PATH", OUTTYPE_ID [, INSTANCE_ID]? ) ; ]+
```

where

- `T` is an instruction to print the temperature of a particular thermal cell, identified by its three-dimensional coordinates in the thermal grid, in a text file. The output format consists of two columns- the time instance (in sec) against the corresponding temperature value (in K).

```
T( VER_ID, DVALUE, DVALUE, "PATH" [, INSTANCE_ID]? ) ;
```

- VER_ID specifies the "z" location of the thermal cell. This can assume a value of LL_ID, DD_ID or a CC_ID (see Section 6.A.v). If a die (DD_ID) is specified, then the thermal cell would be located in the source layer of that die,
- D_VALUE, D_VALUE (in µm) specify the "x" and "y" locations of the thermal cell. An error message is printed if the location specified is outside of the computational domain,
- PATH is the path of the text file in which the output must be written,
- INSTANCE_ID is an optional parameter that can assume a value of step, slot or final and specifies the frequency in which the output must be reported- at the end of each internal time step, at the end of each time slot, or only at the end of the simulation respectively. Note that in the case of steady state simulation, only those instructions that have the option final are executed. When no parameter is specified, final is assumed,

- Tflp is an instruction to print the temperature of a particular floorplan (the source layer of a particular die), identified by its die identifier (DD_ID). The output format consists of two columns- the time instance (in sec) against the corresponding temperature value (in K).

```
Tflp( DD_ID, "PATH", OUTTYPE_ID [, INSTANCE_ID]? ) ;
```

- DD_ID is the identifier of the die whose floorplan temperature must be printed.
- PATH is the path of the text file in which the output must be written,
- OUTTYPE_ID specifies the exact nature of the temperature to be reported, and can assume the values maximum, minimum or average, corresponding maximum, minimum and average temperature in the floorplan respectively,
- INSTANCE_ID is an optional parameter that can assume a value of step, slot or final and specifies the frequency in which the output must be reported- at the end of each internal time step, at the end of each time slot, or only at the end of the simulation respectively. Note that in the case of steady state simulation, only those instructions that have the option final are executed. When no parameter is specified, final is assumed.

- Tflpel is an instruction to print the temperature of a particular floorplan element (the power block of the source layer of a particular die), uniquely identified by its die identifier (DD_ID) and floorplan element identifier (FLPEL_ID). The output format consists of two columns- the time instance (in sec) against the corresponding temperature value (in K).

```
Tflpel( DD_ID.FLPEL_ID, "PATH", OUTTYPE_ID [, INSTANCE_ID]? ) ;
```

- DD_ID.FLPEL_ID contains the die and the floorplan element identifier (see Section 6.B) which can uniquely identify a floorplan element in the entire stack,
- PATH is the path of the text file in which the output must be written,

- OUTTYPE_ID specifies the exact nature of the temperature to be reported, and can assume the values maximum, minimum or average, corresponding maximum, minimum and average temperature in the floorplan element area respectively,
- INSTANCE_ID is an optional parameter that can assume a value of step, slot or final and specifies the frequency in which the output must be reported- at the end of each internal time step, at the end of each time slot, or only at the end of the simulation respectively. Note that in the case of steady state simulation, only those instructions that have the option final are executed. When no parameter is specified, final is assumed.

- Tmap is an instruction to print the temperature map of a particular layer in the stack, identified by its identifier. The output is printed in a matrix format in a text file, with each line representing a row of thermal cells in the thermal grid (counted in "y" direction), and every temperature value (in K) in each line corresponding to a thermal cell (columns, counted in the "x" direction). Once the temperature map at a given time point is printed in this format, the temperature map at the next time point requested is printed from the next line in the same format. Hence, the total number of lines in the file is the product of the number of rows in the thermal grid and the number of instances the output is printed. In addition, whenever the instruction Tmap is used in a project, two other files, named "x_axis.txt" and "y_axis.txt" are printed which consist of the indices of the columns and rows (in µm), respectively, for these temperature maps. They can be used to visualize the data in the text files with the help of applications supporting graphical outputs such as Matlab.

```
Tmap( VER_ID, "PATH" [, INSTANCE_ID]? ) ;
```

- VER_ID identifies the layer for which the temperature map must be printed. This can assume a value of LL_ID, DD_ID or a CC_ID (see Section 6.A.v). If a die (DD_ID) is specified, then the source layer of that die is used,
- PATH is the path of the text file in which the output must be written,
- INSTANCE_ID is an optional parameter that can assume a value of step, slot or final and specifies the frequency in which the output must be reported- at the end of each internal time step, at the end of each time slot, or only at the end of the simulation respectively. Note that in the case of steady state simulation, only those instructions that have the option final are executed. When no parameter is specified, final is assumed.

- Pmap is an instruction to print the power map of a particular die in the stack, identified by its identifier. The output is printed with the same criteria implemented to print a temperature map.

```
Pmap(DD_ID, "PATH" [, INSTANCE_ID]? ) ;
```

- DD_ID identifies the die for which the power map must be printed.
- PATH is the path of the text file in which the output must be written,
- INSTANCE_ID is an optional parameter that can assume a value of step, slot or final and specifies the frequency in which the output must be reported- at the end of each internal time step, at the end of each time slot, or only at the end of the

simulation respectively. Note that in the case of steady state simulation, only those instructions that have the option `final` are executed. When no parameter is specified, `final` is assumed.

- `Tcoolant` is an instruction to print the temperature of the coolant at the outlet of a cavity identified by its channel identifier (`CC_ID`). The output format consists of two columns- the time instance (in sec) against the corresponding temperature value (in K).

  `Tcoolant( CC_ID, "PATH", OUTTYPE_ID [, INTANCE_ID]? ) ;`
  - `CC_ID` is the identifier of the cavity whose outlet temperature must be printed,
  - `PATH` is the path of the text file in which the output must be written,
  - `OUTTYPE_ID` specifies the exact nature of the temperature to be reported, and can assume the values `maximum`, `minimum` or `average`, corresponding maximum, minimum and average outlet temperature respectively,
  - `INSTANCE_ID` is an optional parameter that can assume a value of `step`, `slot` or `final` and specifies the frequency in which the output must be reported- at the end of each internal time step, at the end of each time slot, or only at the end of the simulation respectively. Note that in the case of steady state simulation, only those instructions that have the option `final` are executed. When no parameter is specified, `final` is assumed.

Examples

Two example stack descriptor files, one steady state and one transient, based on the above discussions are shown below.

Example (steady state)

```
material SILICON :
       thermal conductivity     1.30e-4;
       volumetric heat capacity 1.628e-12;
material BEOL :
       thermal conductivity     2.25e-6;
       volumetric heat capacity 2.175e-12;

top heat sink:
       heat transfer coefficient 1.0e-7;
       temperature 300;

microchannel4rm:
       height 100 ;
       channel length 50;
       wall length50;
       first wall length 25;
       last  wall length 25;
       wall material SILICON;
       coolant flow rate 42;
       coolant heat transfer coefficient top 5.7132e-8,
```

```
                    bottom 4.7132e-8;
      coolant volumetric heat capacity4.172e-12;
      coolant incoming temperature300;
layer PCB:
      height 10 ;
      material BEOL ;
die TOP_IC:
      source 2 SILICON;
      layer 50 SILICON;
die BOTTOM_IC:
      layer 10 BEOL;
      source 2 SILICON;
      layer 50 SILICON;
dimension :
      chip length 10000, width 10000;
      cell length 50, width 50;

stack:
      die     MEMORY_DIE     TOP_IC    floorplan"./mem.flp";
      channel TOP_CHANNEL;
      die     CORE_DIE       BOTTOM_IC floorplan"./core.flp";
      channel BOTTOM_CHANNEL;
      layer   BOTTOM_MOST    PCB ;

solver:
      steady ;
      initial temperature 300.0 ;

output:
      T( MEMORY_DIE, 5000, 3000, "output1.txt",final);
      Tmap( CORE_DIE, "output2.txt",  final );
      Pmap( CORE_DIE, "output3.txt",  final );
      Tflpel(CORE_DIE.core1, "output4.txt", average);
      Tcoolant(TOP_CHANNEL, "output5.txt", maximum);
```

Example (transient)

```
material SILICON :
      thermal conductivity     1.30e-4;
      volumetric heat capacity 1.628e-12;
material BEOL :
      thermal conductivity     2.25e-6;
      volumetric heat capacity 2.175e-12;

top heat sink:
      heat transfer coefficient 1.0e-7;
      temperature 300;

microchannel4rm:
      height 100 ;
      channel length 50;
      wall length 50;
      first wall length 25;
      last  wall length 25;
      wall material SILICON;
      coolant flow rate 42;
      coolant heat transfer coefficient top 5.7132e-8,
                  bottom 4.7132e-8;
      coolant volumetric heat capacity 4.172e-12;
```

```
       coolant incoming temperature 300;

layer PCB:
       height 10 ;
       material BEOL ;
die TOP_IC:
       source 2 SILICON;
       layer 50 SILICON;
die BOTTOM_IC:
       layer 10 BEOL;
       source 2 SILICON;
       layer 50 SILICON;

dimension :
       chip length 10000, width 10000;
       cell length 50, width 50;

stack:
       die     MEMORY_DIE     TOP_IC    floorplan "./mem.flp";
       channel TOP_CHANNEL;
       die     CORE_DIE       BOTTOM_IC floorplan "./core.flp";
       channel BOTTOM_CHANNEL;
       layer   BOTTOM_MOST    PCB ;


solver:
       transient step 0.02, slot 0.2 ;
       initial temperature 300.0 ;

output:
       T( MEMORY_DIE, 5000, 3000, "output1.txt",step);
       Tmap( CORE_DIE, "output2.txt",  slot );
       Pmap( CORE_DIE, "output3.txt",  slot );
       Tflp( MEMORY_DIE, "output4.txt", minimum, step);
       Tflpel(CORE_DIE.core1, "output5.txt", average, final);
       Tcoolant(TOP_CHANNEL, "output6.txt", maximum);
```

B. Floorplan File

Every die in the stack must be related to a "Floorplan File" (*.flp), whichessentially provides the power dissipation profile (or heat sources) for the simulation. Each Floorplan file must contain the list of functional blocks (cores, caches, memories, etc), their positions, and the power dissipation as a function of time.

Every functional block, here called *floorplan element*, is an area inside the die, laid out in the source layer. Each floorplan element has a unique identifier- the name it is assigned. In addition, the position and the dimensions of each floorplan element are given (in µm) based on the same Cartesian coordinates that was used for building the stack, with the origin at the SOUTH-WEST corner of the source layer. An example floorplan of a 1cmX1cm die with the reference coordinates is shown in Fig. 19. All the distances shown here are in µm.

A floorplan element in the Floorplan File is declared using the following syntax.

```
       IDENTIFIER :
```

```
        position  DVALUE , DVALUE ;

        dimension DVALUE , DVALUE ;

        [discretization DVALUE, DVALUE ;]?

        [ power values DVALUE [ , DVALUE ]* ]? ;
```

Or, if its surface cannot be represented as a simple rectangle, the following syntax is also supported(only in the uniform mode):

```
IDENTIFIER :

        [ rectangle (DVALUE , DVALUE , DVALUE , DVALUE ) ; ]+

        [ power values DVALUE [ , DVALUE ]* ]? ;
```

where

- `IDENTIFIER` is the unique identifier used to name the floorplan element. This string must be unique within the floorplan file it belongs to but it can be used on a different file,
- `position`, is the (x,y) coordinate of the SOUTH-WEST corner of the floorplan element (in µm),
- `dimension` is the (length, width) dimensions of the floorplan element (in µm),
- `rectangle` is the (x,y, length, width) description of a part of the floorplan element (in µm)
- The `DVALUE`(s) against the keyword `power values` are the list of power dissipation values (expressed in W) of the floorplan element for each *time slot* (scroll down for the explanation of time slots in 3D-ICE) separated by commas.
- `[discretization DVALUE, DVALUE ;]?` is the discretization level the floorplan element has. This argument only takes effect in non-uniform mode. If this argument is not specified, it inherits the default disretization level the die has.

As an example, the declaration for the Core 0 and the L2 Chache 0 in Fig. 19 with 5 time slots would be as follows:

Example

```
L2_Cache_0 :

        position 0, 1750;
        dimension 1250, 7500;
        power values 0.3, 0.3, 0.4, 0.2, 0.3;

Core_0 :
        rectangle (   0,    0, 1250, 1750) ;
        rectangle (1250,    0, 3750, 2000) ;
        rectangle (1250, 2000, 2500, 1500) ;
        power values 0.5, 0.5, 0.1, 0.1, 0.7;
```

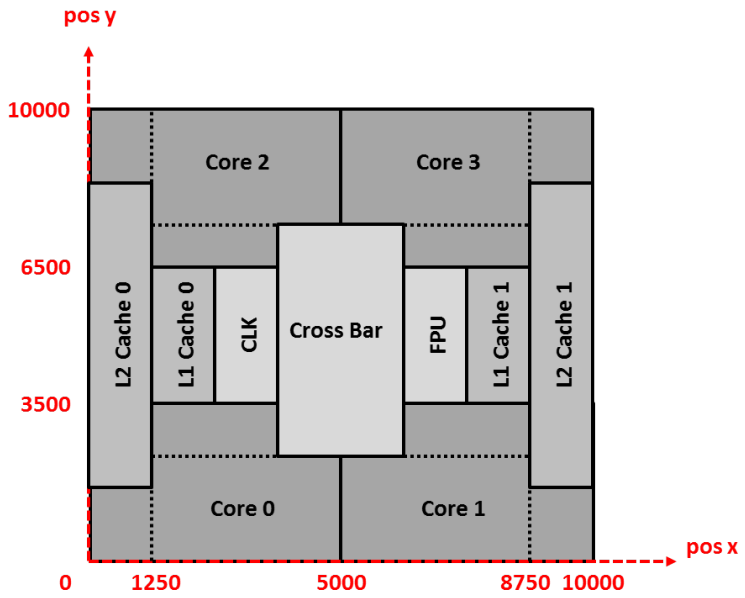


Figure 19: An example floorplan for a 1cmX1cm IC die. Notice the irregular floorplans Core 0, Core 1, Core 2, Core 3.

The following points must be kept in mind while writing the floorplan files:

- Floorplan elements must not overlap. During the parsing of the Floorplan File, the values describing the elements are checked to verify that all the elements are inside the chip and that they donot overlap.

- In the non-uniform mode, each floorplan element is discretized into a number of cells following the discretization setting. There will be no intersect area between different floorplan elements.

- In the uniform mode, when the stack structure is discretized based on the given thermal cell dimensions, the power dissipated by a floorplan element is **uniformly** divided among the thermal cells that *intersect its surface*. **It is not required that the position and dimensions of the floorplan elements are an exact multiple of the dimensions of the thermal cells.** Therefore, depending on the layout of the IC, **a thermal cell can receive the power from different floorplan elements**. For the same reason, **a floorplan element can** also **be smaller than the thermal cell** itself. The assignment of thermal cells to different floorplan elements is illustrated in Fig. 20.The thermal cell highlighted in the center of the image receives as input the power coming from the `FPU` unit, the `L1Cache1` and the `Core1` at the same time. The power value consumed by each component will be scaled according to the fraction of the surface of the thermal cell covered by the corresponding IC unit:

$$Input = Power_{FPU}\frac{a_1}{Area_{FPU}} + Power_{CacheL1}\frac{a_2}{Area_{CacheL1}} + Power_{Core1}\frac{a_3}{Area_{Core1}}$$

- The same Floorplan File can be assigned to 2 or more dies in the Stack Description File if they happen to have identical structure and behavior in the design. Each floorplan element in the

entire design, in that case, is uniquely identified by the `DD_ID` identifier of the corresponding die element and the `IDENTIFIER` of the floorplan element.

- If two dies in the stack have the same floorplan but during the simulation they have a different power dissipation activity, then 2 different Floorplan Files must be created for each die and assigned to the corresponding die element declarations in the Stack Description File. This is because the power dissipations are directly linked to each floorplan element in the Floorplan File.
- In the uniform mode, it is possible to have gaps in the floorplan- regions where there is no floorplan element and hence, no power dissipation. The thermal cells in these regions will simply not be assigned any source value during the solving of system equations. However, it is not allowed to have gaps in the floorplan regions in the non-unifrom mode because of its unique modeling method.

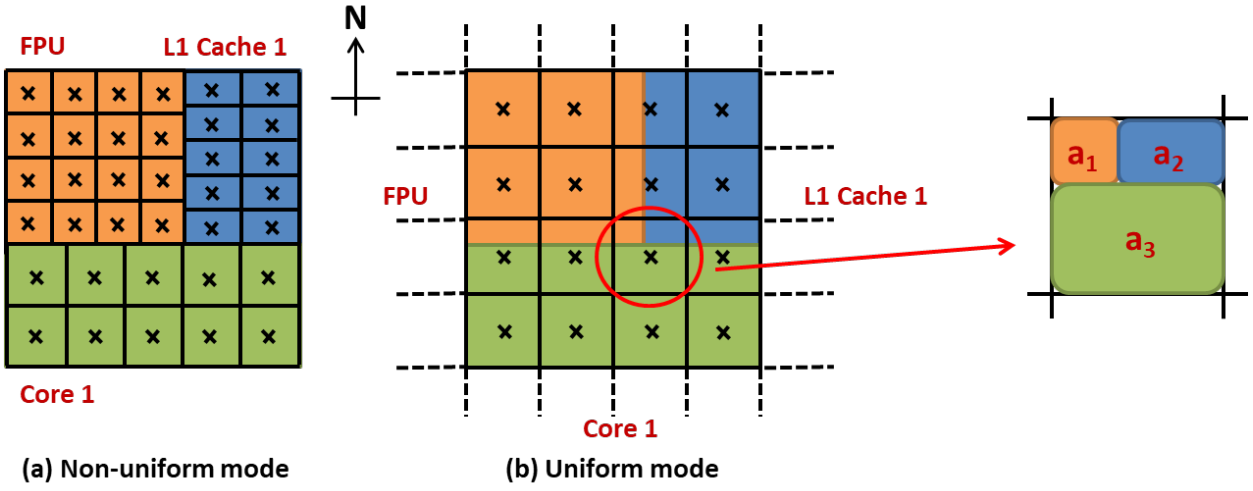


Figure20: Assignment of thermal cells to floorplan elements

Time Slots

The entire time-interval of simulation (ToS) in 3D-ICE is divided into *time slots*- the minimum time duration for which the switching activity of the floorplan elements has been resolved. For example, if for a given design the switching activity (a measure of how much a floorplan component is active, directly related to its power dissipation) is sampled every 200ms during a 1 second ToS, then there are 5 time slots for the 3D-ICE simulation. And hence, there must be 5 values of power dissipation for each floorplan element declaration in the Floorplan File. Conversely, the number of power values for the floorplan element is interpreted as the number of time slots (NoTS) for the simulation. In all 3D-ICE simulations, it is assumed that the power dissipation in a particular thermal cell is CONSTANT during the period of a time slot- calculated based on the power value of the corresponding floorplan element,in which the thermal cell exists, for that time slot (see Fig. 21).

Note that time slot is different from *time step*, which is the discretization length of the time-domain for the numerical integration of the system of differential equations representing the entire thermal circuit created inside 3D-ICE. The exact durations of both time slot and time step are given when running the simulator. For further details, see Chapter 5.

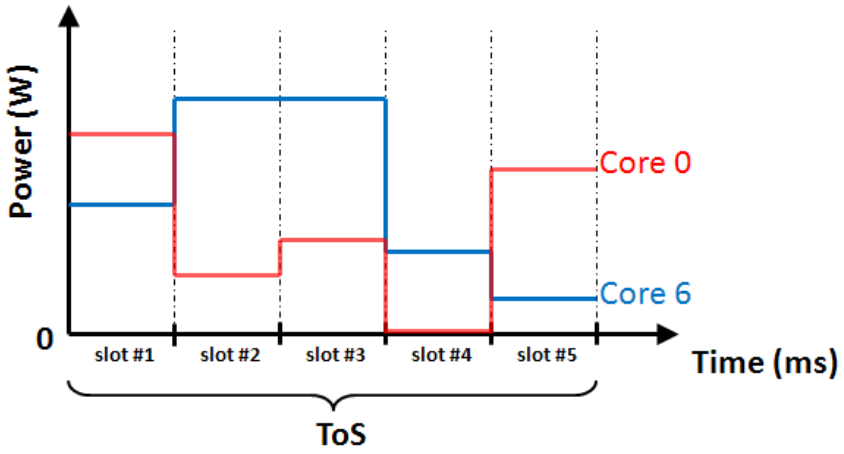


Figure 21: Power dissipation profile of Core 0 and Core 6 in Fig. 19

Since uniform sampling of the switching activity for all the components in an IC is assumed, the number of power values given for each floorplan element in the Floorplan File shouldbe the same. Even when there is zero power dissipation for a particular floorplan element during some time slot (Core 0 at slot #4 in Fig. 21) or if a particular floorplan element has constant power dissipation during some or all time slots (Core 6 during slot #2 and slot #3 in Fig. 21), the corresponding power values must be mentioned explicitly for each time slot. The function `emulate_slot` returns a value indiacting the end of the simulation whenever if finds a floorplan element with an empty list of power values. Therefore, the ToS will be determined by the length of the shortest list of power values within all the floorplan elements that are declared in the dies.

8. Co-simulation and plugin interface

3D-ICE starting from version 3.0.0 supports a pluggable heat sink interface that allow it to perform co-simulation with a separate heat sink model. 3D-ICE pluggable heat sinks can be found in the `heatsink_plugin/heatsinks` directory. User-supplied heat sinks can also be simulated. In this configuration, 3D-ICE simulates the 3D IC stack and an heat spreader connected to the top of the 3D IC stack, whose dimensions can be larger than the stack itself. If the heat dissipation stack to be simulated does not include a heat spreader, the last layer of the IC can take the place of the heat spreader.

The simulation of the heat sink is instead performed by a heat sink plugin, in the form of a dynamic library loaded by 3D-ICE.

Plugin-related files are in the `heatsink_plugin` directory. This directory contains the following subdirectories:

- `loaders`: This directory contains the loader plugins. The 3D-ICE plugin interface exposes a C API that is directly accessible by C/C++ programs without a loader. Loaders are used to support co-simulation with higher level programming languages.

  Two loaders are currently supported:

  - FMI: Allows to load plugins that conform to the FMI, or Functional Mockup Interface (https://fmi-standard.org/). This loader is targeted mainly at Modelica models. This is the only loader that performs grid pitch mapping between the finite volume grid used by 3D-ICE and the plugin, as doing this part in C++ is much faster than doing it in Modelica.

  - Python: Allows to load a python file as a plugin.

- `templates`: This directory contains example code intended for users who want to learn how to write a plugin.

- `heatsinks`: This directory contains heat sink models.

- `common`: Contains libraries for building heat sinks.

Writing a heat sink in Modelica

Modelica is a declarative object-oriented modeling language. The Modelica syntax allows to write equations as opposed to assignment statements in ordinary imperative programming languages, and supports linear and nonlinear differential equations natively through the `der()` operator. A Modelica translator uses symbolic equation manipulation to automatically perform the steps needed to produce imperative code that performs the numerical integration of differential equations. OpenModelica is an open source Modelica translator that translates Modelica to C code. Modelica is thus a very convenient

choice for the modeling of heat sinks, as it is only necessary to write the heat transfer equations instead of writing the code to solve them. An general introduction to the Modelica language can be found at https://mbe.modelica.university/front/intro

At the lowest level, modeling an heat sink in Modelica for 3D-ICE co-simulation requires to implement a model with the following interface:

```
model Interface3DICE
  parameter String args;
  parameter Modelica.SIunits.Temperature initialTemperature;

  parameter Integer sinkRows;
  parameter Integer sinkColumns;
  parameter Modelica.SIunits.ThermalConductance sinkCellBottomConductance;
  parameter Modelica.SIunits.Length sinkLength;
  parameter Modelica.SIunits.Length sinkWidth;
  parameter Modelica.SIunits.Length spreaderX0;
  parameter Modelica.SIunits.Length spreaderY0;

  Modelica.Blocks.Interfaces.RealOutput T[sinkRows, sinkColumns];
  Modelica.Blocks.Interfaces.RealInput Q_flow[sinkRows, sinkColumns];
end Interface3DICE;
```

The first two parameters, `args` and `initialTemperature` are set by 3D-ICE at the beginning of the simulation and can be read from the Modelica code.

The other parameters define the sink meshing, thermal conductance towards the spreader, and heat sink geometry. These parameters must be set by the plugin as they are read by 3D-ICE to connect the sink to the spreader. The variables `sinkRows`, `sinkColumns`, `sinkCellBottomConductance`, `sinkLength`, `sinkWidth` depend on the heat sink geometry and how it has been modeled, therefore they are expected to be constants set in the Modelica code. The last two variables, `spreaderX0` and `spreaderY0` depend on the placement of the sink relative to the spreader, and it is common practice to make them configurable from the 3D-ICE stack file by parsing the `args` variable and assigning them from Modelica code.

A graphical representation of the `sinkLength`, `sinkWidth`, `spreaderX0`, `spreaderY0` parameters is shown below. Note that Modelica encourages the use of SI units, thus all lengths and widths are in meters, and not in micrometers unlike in 3D-ICE.

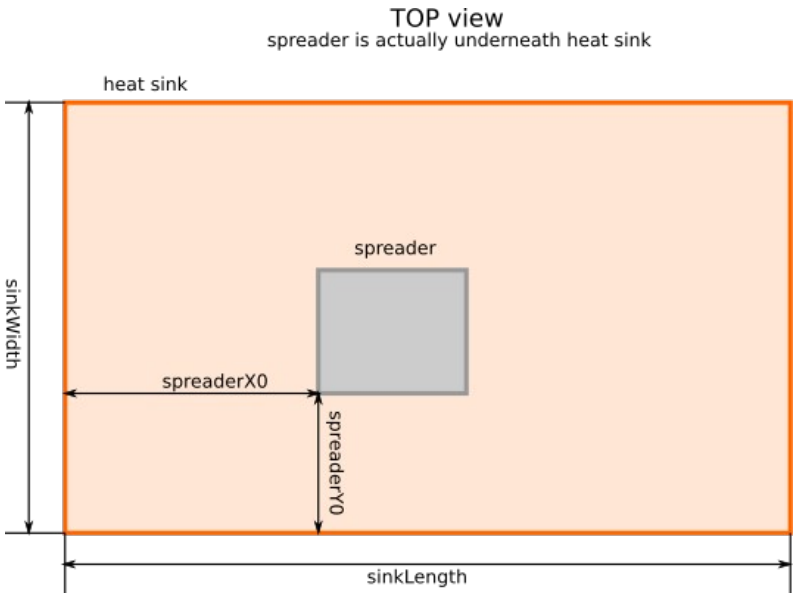


Figure 22: Mapping between a generic sized heat sink and the heat spreader

Finally, the `T` variable is the heat sink bottom temperature, that 3D-ICE reads at every simulation step, and `Q_flow` is the heat flow towards the heat sink, that 3D-ICE sets at each simulation step.

Convenience libraries are provided in the `common` directory to abstract this interface using Modelica object-oriented features.

- The HeatsinkBlocks.PartialModels.Interface3DICE is a base class that implements the `Interface3DICE` interface and includes the boilerplate code required to parse `args` and set `spreaderX0` and `spreaderY0`, as well as code for forwarding of all other parameters from a redeclarable heat sink model. Further parameter parsing is possible in extended classes, providing a generic way to pass arbitrary parameters from the 3D-ICE stack description to heat sinks. This class also maps the causal interface of the the FMI into a-causal heat ports as commonly used to model heat exchange in Modelica.

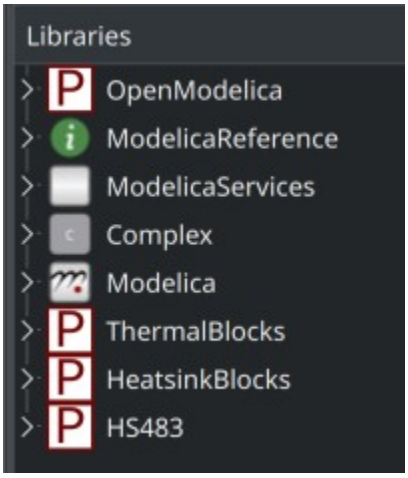
- The HeatsinkBlocks.PartialModels.Heatsink is a generic heat sink base class that can be extended to add the differential equations modeling the desired heat sink.

The example `templates/Modelica` can be used as a reference to understand how to write a new heat sink in Modelica. Additionally, looking at the complete heat sinks in the `heatsinks` directory can be useful as well.

Editing a Modelica Heatsink using the OMEdit editor

OMEdit is a Modelica IDE, part of the OpenModelica package to simplify writing Modelica code. Although a complete guide would stray from the purpose of this document, it is important, when

loading heat sink model using this editor to first load the required libraries in the correct order. First load the ThermalBlocks library, then the HeatsinkBlocks one and finally the model you intend to edit.



Using a heat sink written in Modelica

Modelica heat sinks require the FMI loader plugin to be interfaced to 3D-ICE, thus the plugin to be loaded is always `fmi_loader.so`. This is an example taken from `templates/Modelica`

```
top pluggable heat sink :
        spreader length 20000 , width 20000 , height 1000 ;
        material COPPER ;
        plugin "../../loaders/FMI/fmi_loader.so", "Heatsink.TestHeatsink_Interface3DICE 0.0 0.0";
```

The FMI loader interprets the first argument in the args string as the name of the top level model to load, in this case `Heatsink.TestHeatsink_Interface3DICE`. The name must be fully qualified including the package, and must be a model that implements the `Interface3DICE` interface. When the model to be loaded has been obtained by extending the the HeatsinkBlocks.PartialModels.Interface3DICE interface, there will be two additional parameters that are the to determine how the heat sink is placed on top of the spreader. The first parameter is `spreaderX0`, while the second one is `spreaderY0`. These two parameter are in meters, not micrometers. Additional heat-sink specific parameters may be required depending on the heat sink model.

Writing a heat sink in C++ or Python

These languages do not provide a native way of solving differential equations. You need to choose appropriate libraries for that. Taking as example the C++ plugin template, modeling an heat sink requires to implement the following class

```
class HeatSink
{
public:
    HeatSink(unsigned int nRows, unsigned int nCols,
```

```
            double cellWidth,   double cellLength,
            double initialTemperature,
            double spreaderConductance,
            double timeStep
            const std::string& args);

    void simulateStep(const CellMatrix spreaderTemperatures,
                            CellMatrix heatFlow);
};
```

The class constructor is called by 3D-ICE to inform the heat sink plugin of the simulation parameters, which are

- `nRows, nCols` the number of rows and columns of thermal cells of the heat spreader,
- `cellWidth, cellLength` the size of a heat spreader thermal cell, in µm,
- `initialTemperature` the temperature of the heat spreader at the beginning of the simulation, in K,
- `spreaderConductance` the thermal conductance from the center of the heat spreader thermal cells to their top face, in W/K,
- `timeStep` the simulation time step in seconds.

Then, the `simulateStep` member function is called at each simulation time step to update the heat sink model. Its parameters are

- `spreaderTemperatures` a matrix provided by 3D-ICE with the current temperatures of the heat spreader cells. This is an input parameter.
- `heatFlow` the heat flow from the spreader to the sink. This is an output parameter, that has to be computed by the plugin.

The HS483 heat sink model

HS483 is a Modelica model for a COTS HS483-ND heat sink, connected to a P14752-ND fan. Fan speed should be limited in the range from 1500 to 6000RPM (or 0, for natural convection). Power to be dissipated should be limited to less than 40W. More information is detailed in the model source code.
Two 3D-ICE interfaces are available for this model:
`HS483_P14752_ConstantFanSpeed_Interface3DICE`: this model has the following parameters

- `spreaderX0, spreaderY0`: see chapter Writing a heat sink in Modelica
- `airTemperature`: air temperature in K
- `fanSpeed`: fan speed in RPM

`HS483_P14752_VariableFanSpeed_Interface3DICE`: this model has the following parameters

- `spreaderX0, spreaderY0`: see chapter Writing a heat sink in Modelica
- `airTemperature`: air temperature in K
- `fanSpeedFilename`: filename of a table with the fan speed as a function of the simulation time. The format of this table is the one of the Modelica CombiTimeTable, an example is provided in the heat sink directory.

The cuplex_kryos_21606 heat sink model

cuplex_kryos_21606 is a Modelica model for a COTS uplex kryos NEXT water block from aqua computer, part number 21606. The water block has been fitted in the range 0.06 to 0.12 l/min. Power to be dissipated should be limited to less than:

- 20W, for a 0.06l/min water flow
- 70W, for a 0.08l/min water flow
- 80W, for higher water flow rates.

More information is detailed in the model source code.

Two 3D-ICE interfaces are available for this model:

`Cuplex21606_ConstantFlowRate_Interface3DICE`: this model has the following parameters

- `spreaderX0`, `spreaderY0`: see chapter Writing a heat sink in Modelica
- `waterTemperature`: air temperature in K
- `waterFlowRate`: water flow rate in l/min

`Cuplex21606_VariableFlowRate_Interface3DICE`: this model has the following parameters

- `spreaderX0`, `spreaderY0`: see chapter Writing a heat sink in Modelica
- `airTemperature`: air temperature in K
- `flowRateFilename`: filename of a table with the water flow rate as a function of the simulation time. The format of this table is the one of the Modelica CombiTimeTable, an example is provided in the heat sink directory.

9. Network interface for remote simulations

The 3D-ICE software library includes software object that can be used to implement simulations where two distinct processes, a client and a server, communicate through a network socket. In this interface, 3D-ICE running on a host machine, acts as the server and the some device or external source that generates the power trace inputs acts as a client. Such set of data structures and functions can be used whenever it is necessary to decouple the generation of power traces and the control of the simulation from its execution, which might requires more computational resources. The network interface can be used in the following scenarios:

- Hardware/software co-simulation: for example, obtaining real-time temperature data from an architecture being designed and prototyped on an FPGA, testing run-time thermal management techniques that depend on data from temperature sensors in a real device. For more details, see [6]
- Creating graphical interfaces that involve communication with other processes running parallel on a computer.
- Integration in other tools or libraries: for example, floorplanning software, software emulators for power traces on the same computer that is running 3D-ICE.

In particular, it will be useful whenever power traces are not known a priori and cannot be written as input for the simulation directly in the floorplan file, and for cases where the power traces are too long and must be read from large data files. The primary project files for a simulation, i.e. the .stk and the flp files, describing the structure of the IC and the floorplan architecture must still be written and stored in the server. The client would simply send the power traces that are otherwise entered in the .flp files. The server performs the thermal simulation and provides the real-time values of the outputs requested in the .stk file. These values will be sent over a socket back to the client, instead of being printed in a text file. The client can also send command to the server to request any kind of change of the state of the simulation (reset temperatures, change flow rate, run simulation steps, etc).

For the usage of the interface, two data types have been added as well as functions to handle them as done for the other components of the library. These data types are `Socket` and`NetworkMessage` and they will be described in Section 9.

10. Running 3D-ICE

In the `3d-ice/bin/` folder, the main simulator file to be compiled is`3D-ICE-Emulator.c`. It has been written to parse and analyze any Stack Description File and the corresponding Floorplan Files placed in the same folder. Once compiled with the make command, the corresponding executable is acts as the thermal simulator application. To simulate a new thermal project, you must:

- create the Stack Description File and the corresponding Floorplan Files according to the instructions in Chapter 6
- run the following command in `3d-ice/bin/`
  `$./3D-ICE-EmulatorPATH`

where `PATH` is the path to the Stack Descriptor File containing the description of the 3D-IC project.

The project is then simulated and the results, as requested in the .stk file, are written in the text files specified in those instructions by the user.

3D-ICE uses backward Euler method with constant time-stepping to solve the system equations. Hence, the solution is always numerically unconditionally stable. However, accuracy can be increased by reducing the time step value. The local truncation error of backward Euler method behaves as $\boldsymbol{O(h^2)}$, where h is the time step. However, given a ToS, the number of time steps in the entire simulation is a $\boldsymbol{O(1/h)}$ function. Hence, the upper bound of the total accumulated error at the final time point in the simulation behaves as

$$\boldsymbol{O(h^2)\cdot O(1/h)=O(h)}.$$

In other words, the total final error is approximately a linear function of the time step. For RC circuits, such as the thermal circuit that 3D-ICE solves, it is common practice to have at least 5 time steps for the

duration of a rise time of the output temperature (defined as the time duration for the rise of temperature from 10% to 90% of its steady state value) to resolve the transients accurately.

Numerical stability when using the co-simulation interface

The numerical stability of performing co-simulation of differential equation models is a complex matter [8]. To not stray from the purpose of this manual, it can be broadly stated that it depends on the numerical stability of the integration algorithm used by 3D-ICE, on that used by the plugin and, for extreme values of the integration step, also on the co-simulation interface itself [9].

OpenModelica, the open source Modelica compiler we used for 3D-ICE plugins supports multiple integration methods, including BDF solvers, Runge Kutta, and forward and backward Euler. However, this is only true for all-Modelica simulations. When Modelica models are exported as FMI for co-simulation, OpenModelica currently only supports the forward Euler algorithm, that is not unconditionally stable and is currently the limiting factor as for lengthening the integration step. In the future, we expect OpenModelica to implement implicit integration algorithms for FMIs.

It should also be noted that the Modelica environment, not to unduly limit the potential for optimization given by the choice of solvers, expects users to be responsible of making appropriate choices of solver algorithms and integration step, and that the fact that an unwise choice could lead to numerical instability, as well as the capability of correcting such an unwise choice, is common culture and practice in the Modelica community.

From a practical standpoint, we can thus borrow from the Modelica operational experience, in that numerical instability in thermal models is easy to detect, as it shows up as negative temperature values (values in Kelvin cannot be negative), or values exponentially diverging towards infinity. In all cases, with the present state of FMI support in OpenModelica, the solution to this issue is invariantly to reduce the numerical integration step.

As in the co-simulation interface the integration step is dictated by the 3D-ICE side, this can be done by reducing the `step` parameter in the stack description file.

11. Usage of the 3D-ICE as Software Thermal Library

3D-ICE, in addition to functioning as a stand-alone thermal simulator, can also serve as a software thermal library that can be used to build customized applications. Such requires knowledge about the organization and the use of various functions and data structures that are built into 3D-ICE. With the release of 3D-ICE 2.0, we are happy to announce that a new, useful tool for the visualization of the software library, called Doxygen [5], was used to build an online documentation of the 3D-ICE library. It contains convenient access all information about the functions and the files in the library, including a hierarchical visualization of the various dependencies between them. This new documentation can be accessed by running in the `make doc` command inside the 3D-ICE folder, and then opening `3d-ice/doc/html/index.html` on your web browser. We urge users who are interested in employing 3D-ICE to build custom applications to refer to this documentation.

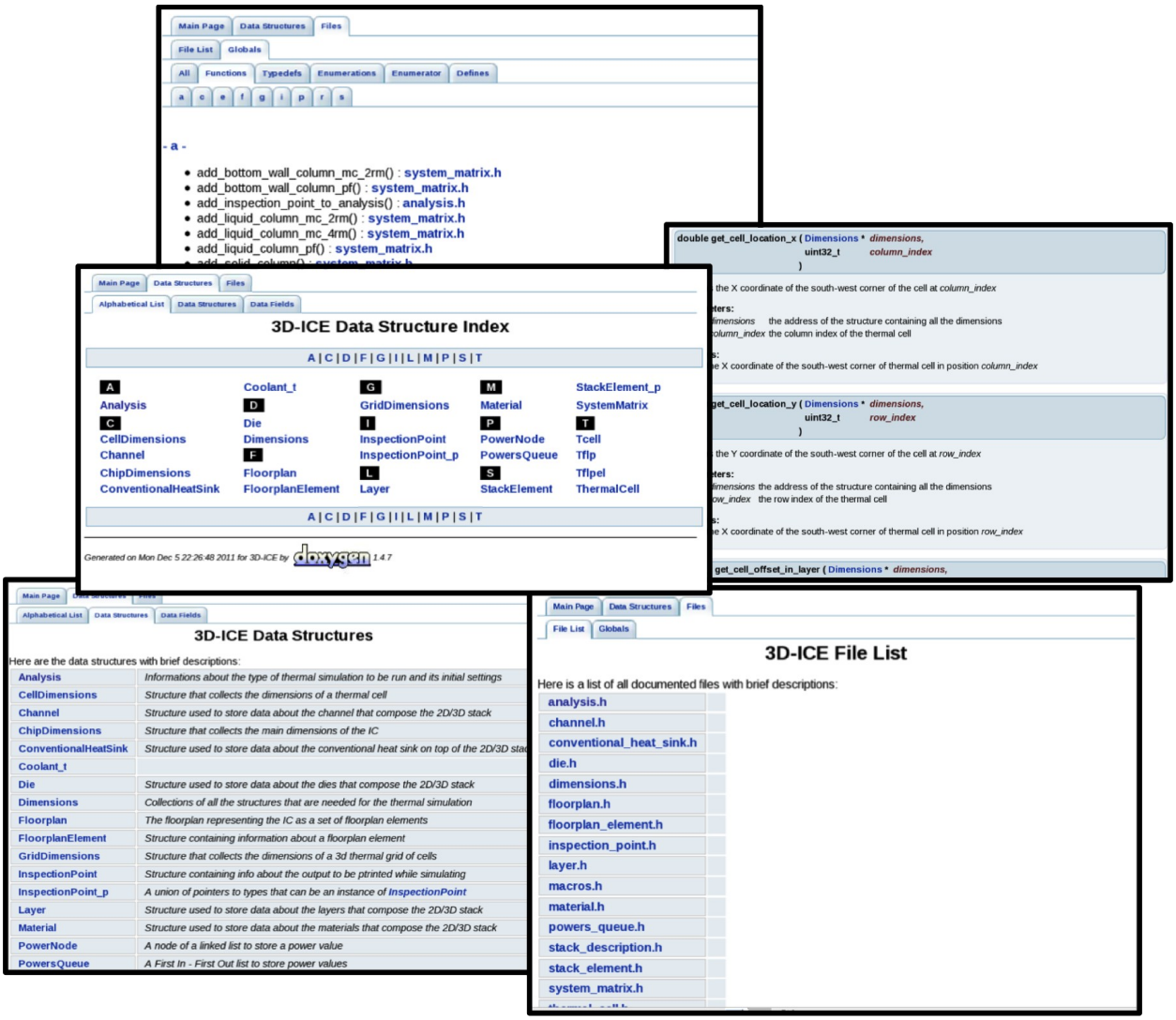


Figure 23: Doxygen documentation for 3D-ICE 2.0

A. StackDescription_t, Analysis_t and Output_t

Using 3D-ICE as a software library involves several steps. First, the problem must be initialized and the necessary data structures containing the 3D IC information must be filled. The three data structures that storethe description of the IC stackand the configuration of the simulation to be run are, respectively,`StackDescription_t`,`Analysys_t` and `Output_t`. To use these types, the header files that must be included in the main file to access the homonym data structures are `stack_description.h`, `analysis.h` and `output.h`.

These data structure types collect all the data pertaining to the 3D IC structure and floorplans. An instance of these data typeswill then be related to theStack Description File. The functions available to initialize, fill and destroyan instance of these two variablesare:

- `stack_description_init (StackDescription_t*)`
- `analysis_init (Analysis_t*)`
- `output_init (Output_t*)`

- `parse_stack_description_file (String_t, StackDescription_t*, Analysis_t*, Output_t*)`

- `stack_description_destroy (StackDescription_t*)`
- `analysis_destroy (Analysis_t*)`
- `output_destroy (Output_t*).`

In particular, the function `parse_stack_description_file`, parses he contet of a stack file and places the information read into the corresponding data structure. This function can be accesses by including the heard file `stack_file_parser.h`. The functions to access the information in this data structure are in the doxygen documentation.

B. ThermalData_t

This data structure type collects all the data needed for the thermal simulation, such astemperatures, power inputs, matrices representing the system equations, etc. Its type is declared in the header `thermal_data.h`. The functions to initialize, build and destroy a `ThermalData_t` variable are:

- `thermal_data_init (ThermalData_t*)`
- `thermal_data_build (ThermalData_t*, StackElementList_t* ,Dimensions_t*, Analysis_t*)`
- `thermal_data_destroy (ThermalData_t*).`

Just as an instance of`StackDescripiton_t, Analysis_t`and `Output_t` are tied to a Stack Description File,an instance of`ThermalData_t` depends upon the sequence of stack elements in `StackElementList_t` (a member of `StackDescription_t)` and `Analysis_t`. Henceit is necessary to initialize and fill the`StackDescription_t` and `Analysis_t` variables before filling the`ThermalData_t` variable. And for the rest of the simulation project, thethree variables must be used in tandem since they refer to the same problem.

C. Emulation and thermal output

Once the data structures mentioned in the previous sectionshave been filled, the next step is to execute the simulation. It can be done using the functions

- `emulate_step (ThermalData_t*, Dimensions_t*, Analysis_t*)`
- `emulate_slot (ThermalData_t*, Dimensions_t*, Analysis_t*)`
- `emulate_steady (ThermalData_t*, Dimensions_t*, Analysis_t*).`

`emulate_step`increments the simulation time by a single time step and then terminates- this function can be called iteratively in a loop until the end of the ToS, controlled using the return variables of these functions which indicating whether or not the simulation has reached certain epochs (see the description in the files containing these functions for more details); while`emulate_slot`increments the simulation time until the simulation time of a single power value finishes.Finally, the `emulate_steady`performs the steady simulation (one step simulation).

Outputs canusually be read from the thermal analysis before, during and at the end of the simulation in the text files mentioned in the output instructions written in the .stk file of the project. The structure `Output_t` stores the list of outputs that should be generated.

If you intend to use 3D-ICE as a software library, there are functions in 3D-ICE that enable the extraction ofthe thermal state of a single floorplan element, an entire floorplan, a single channel outlet or all the coolant outlets in a givenchannel layer at any time during the simulation. It is also possible to readthe temperature of a single thermal cell or print directly the thermal map (a matrix) for a specific layer in the stack. Some of these functions require you to refer to flooplan elements, layers etc. using the corresponding identifiers declared in the Stack Description File. The functions available to generate theoutput are:

- `generate_output_headers (Output_t *, Dimensions_t *, String_t)`
- `generate_output (Output_t*, Dimensions_t*, Temperature_t*, Source_t*, Time_t, OutputInstant_t)`

The first function creates the text files and must be called only once before the simulation. The function `generate_output`, instead, must be called whenever the output (as the current thermal state of the IC) is needed.

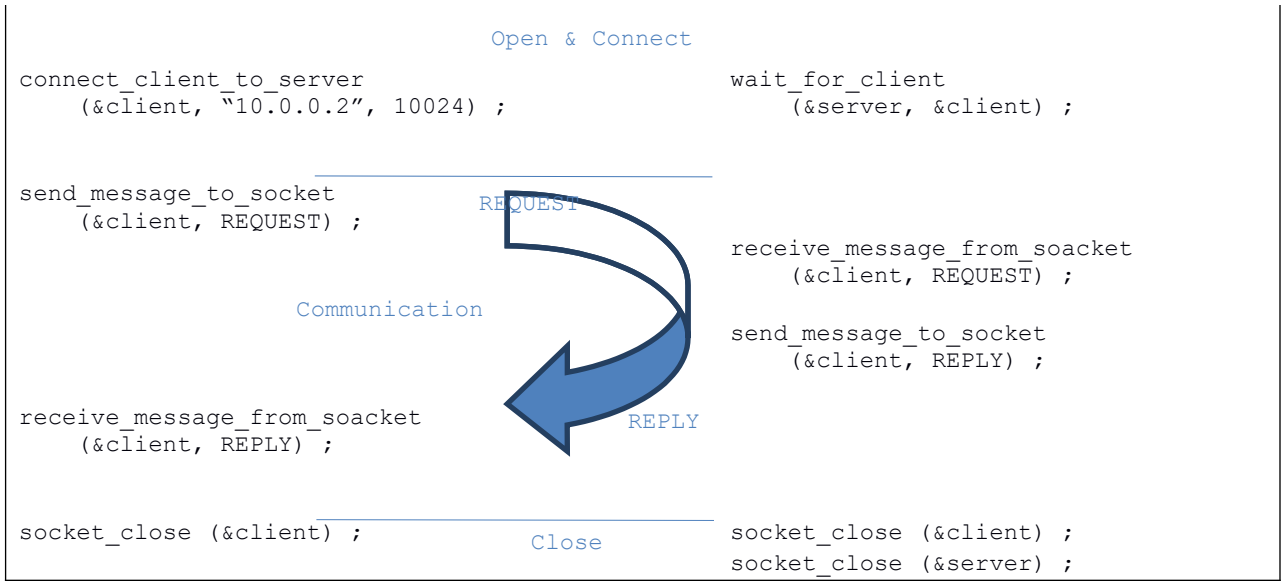
To learn more about all the functions and data structures descried above, please refer to the Doxygen documentation of 3D-ICE in the `3D-ICE/doc/html/` folder.

D. Socket

The client / server communication can be implemented using the data type `Socket_t`. To use it, the header file `network_socket.h` must be included in the source file. The documentation of the functions that can be used to init, open and close the network sockets, as well as to send or receive messages are available through the Doxygen documentation.

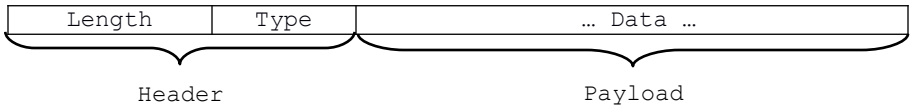
The following example resumes the communication pattern to follow to establish an exchange of messages between the two processes. Please note that the server must use the data structure of type `Socket` twice: one to handle a socket that waits for connections from a client (this is called `server` in the example) and a second one to use to communicate with the client (called `client` in the example) after the connection between the two processes has been established. This mechanism of using two different sockets allows the server to handle many clients at the same time but this functionality is not shown (implemented) in `3D-ICE-Server.c`.

Client 10.0.0.1		**Server 10.0.0.2**
`Socket_t client ;`	Initialization	`Socket_t server ;` `Socket_t client ;`
`Socket_init (&client) ;`		`Socket_init (&server) ;` `Socket_init (&client) ;`
`open_client_socket (&client) ;`		`open_server_socket (&server, 10024) ;`



E. NetworkMessage

The data type `NetworkMessage_t`, available including the header file `network_message.h`, stores the message that can be sent over the network sockets to exchange information between two processes. The structure of the messages that can be built with 3D-ICE is the following:



The header of the message contains two words: `Length` and `Type`. `Length` indicates the total length of the message (header plus payload), expressed as number of words that are going to be sent or received through the socket, while `Type` indicates the type of data that the message contains. The messages can have any length and the functions available in `network_message.h` guarantee that the memory to store the messages is handled transparently: the user must not worry about the content of the field length because its value is computed or updated internally whenever new data is added in the payload. The types of network messages supported so far in 3D-ICE are defined through the enumeration `MessageType_t` defined in the header file `types.h`. The user is free to add new message types and to define the corresponding payload as well as the operations that either the client or the server must do whenever they send or receive a message of such type.

The sequence of functions that must be used to create and send a message is the following:

- `network_message_init (NetworkMessage_t *)`
- `build_message_head (NetworkMessage_t *, MessageType_t)`
- `insert_message_word (NetworkMessage_t *, void *)`
- `send_message_to_socket (Socket_t *, NetworkMessage_t *)`
- `network_message_destroy (NetworkMessage_t *)`

The function called to insert a word of data in the payload must be called as many times as the number of words that must be present in the message according to its type. When the message is built, **the payload behaves as a stack**, meaning that every time that a word is inserted with `insert_message_word`, that word will be put at the right of the actual payload and the length of the message will automatically be increased by one unity. Therefore, **the order of the calls** to `insert_message_word`**must reflect the order of the data** in the payload. Because of this mechanism, the function `network_message_destroy` must be called before creating a new message using the same variable to delete the header and the payload and reset the state of the message.

On the other side, when a message must be received from a socket, the sequence of functions to be used is the following:

- `network_message_init (NetworkMessage_t *)`
- `receive_message_from_socket (Socket_t *, NetworkMessage_t *)`
- `extract_message_word (NetworkMessage_t *, void *, int)`
- `network_message_destroy (NetworkMessage_t *)`

Once the message has been received, the function `extract_message_word` is meant to extract a word from the payload using an index. It means that once the message is received, the **words in the payload can be accessed and read in any order**. Again, the number of words in the payload depends on the type of the message that is received (the type can be accesses using the corresponding field in the data structure) and the message must be reset with `free_network_message` before using the same variable to receive a new message.

More details about the usage of the network interface in 3D-ICE can be found in the sources itself or in their documentation. As main reference, the user can follow the two examplesavailable in the `bin` folder: `3D-ICE-Client.c` and `3D-ICE-Server.c`. These two files shows how to establish the communication between a client and a server and how a client can build messages to send to the server a sequence of power traces and get back from the server some information about the thermal state of the simulation.

F. Debugging of ThermalSimulation

For the purpose of debugging, several pre-processing options can be enabled to directlycheck the values computed during the construction of thermal data (thermal grid/circuit, system matrices, sources etc) before the simulation even starts. These options can be activated uncommenting the correspondingline in the file `3d-ice/sources/Makefile` and running the `make`command again (runthe `make clean` command before building). As a consequence, messages will beredirected to `stderr`.

The debug options that are available are:

- `PRINT_SYSTEM_MATRIX` prints the content of the system matrix while it is filled. For every columnin the system matrix (***G***+***C***/*h*, see [1] for more details),it prints the cell ID of the corresponding cell and the list of the nonzero coefficients indicating representing its neighbors. For every neighbor, it also prints the cell ID.

- `PRINT_SYSTEM_VECTOR`prints the content of the system vector (the Thermal state) at every time step.

G. Binding to SystemC/TLM2.0 Applications

SystemC is a set of C++ classes which can be used to develop event-driven simulations. Transaction-level modeling (TLM) is an approach to modeling digital systems focused in the abstraction of communication between the primary functional blocks within a system. Combined they represent a powerful alternative to create fast and still accurate virtual platforms typically used for performance analysis and architecture exploration.

The 3D-ICE thermal library provides a SystemC/TLM2.0 interface that allows it to be easily integrated into virtual platforms based on those technologies.

Users interested in this feature need to install the SystemC 2.3.1 library (which includes TLM2.0) on their systems. The sources can be downloaded from . Installation instructions can be found in the installation notes file contained in the release package. After fulfilling this installation prerequisite, the correct path to the SystemC library and the appropriate architecture (linux or linux64) must properly set in `makefile.def`. Then a new version version of the 3D-ICE thermal library with enable support to SystemC/TLM2.0 must be compiled.

```
$ make clean

$ make SYSTEMC_WRAPPER=y
```

The new thermal library can be linked to the user program and the `IceWrapper` class implemented in `include/IceWrapper.h` provides the interface between the thermal simulator and the user application.

In addition, an extra executable 3D-ICE-SystemC-Client example will be created inside the `3d-ice/bin/` folder. The source code for this program can be found in the same folder and it serves as a basic example on how to integrate the 3D-ICE thermal library to your SystemC/TLM2.0 based application. To test the program, go to the bin folder execute 3D-ICE-Server and wait until it is ready to receive requests then execute the 3D-ICE-SystemC-Client as presented below:

```
$ ./3D-ICE-Server stack.stk 12345

$ ./3D-ICE-SystemC-Client 127.0.0.1 12345
```

12. References

[1] A Sridhar, A Vincenzi, M Ruggiero, T Brunschwiler, D Atienza, "3D-ICE: Fast compact transient thermal modeling for 3D-ICs with inter-tier liquid cooling", *Proceedings of the 2010 International Conference on Computer-Aided Design (ICCAD 2010)*, San Jose, CA, USA, November 7-11 2010.

[2] A Sridhar, A Vincenzi, M Ruggiero, T Brunschwiler, D Atienza, "Compact transient thermal model for 3D ICs with liquid cooling via enhanced heat transfer cavity geometries", *Proceedings of the 16th International Workshop on Thermal Investigations of ICs and Systems (THERMINIC'10)*, Barcelona, Spain, 6-8 October, 2010.

[3] T Brunschwiler, S Paredes, U Drechsler, B Michel, W Cesar, Y Leblebici, B Wunderle, H Reichl, "Heat-removal performance scaling ofinterlayer cooled chip stacks", *Proceedings of the 2010 IEEE Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems* (ITHERM '10), pp. 1-12, June2010.

[4] T A Davis and E P Natarajan, "Algorithm 8xx: KLU, adirect sparse solver for circuit simulation problems", *ACM Transactions on Mathematical Software*, vol.5, no.1, pp.1–14, 2010.

[5] Doxygen, URL:https://www.doxygen.nl/index.html.

[6] D. Atienza, P. G. Della Valle, G. Paci, F. Poletti and L. Benini, "HW-SW Emulation Framework for Temperature-Aware Design in MPSoCs", *ACM Transactions on Design Automation for Embedded Systems (TODAES)*, vol. 12, no. 3, August, pp. 1 – 26, 2007.

[7] Lee, S., Song, S., Au, V., Moran, K., "Constriction/Spreading Resistance Model for Electronics Packaging", ASME/JSME Thermal Engineering Conf., Vol.4, 1995, pp.199-206.

[8] Cláudio Gomes, Casper Thule, David Broman, Peter Gorm Larsen, and Hans Vangheluwe. 2018. Co-Simulation: A Survey. *ACM Comput. Surv.* 51, 3, Article 49 (July 2018), 33 pages. DOI:https://doi.org/10.1145/3179993

[9] F. E. Cellier and E. Kofman, Continuous system simulation. Springer Science & Business Media, 2006.

[10] F. Terraneo, A. Leva, W. Fornaciari, M. Zapater, D. Atienza, "3D-ICE 3.0: efficient nonlinear MPSoC thermal simulation with pluggable heat sink models", Transactions on Computer-Aided Design of Integrated Circuits and Systems Volume 40 pp. 1-14, 2021-04-19