Advanced Computer Architectures
-
Automatic Design Space Exploration for System on Chip

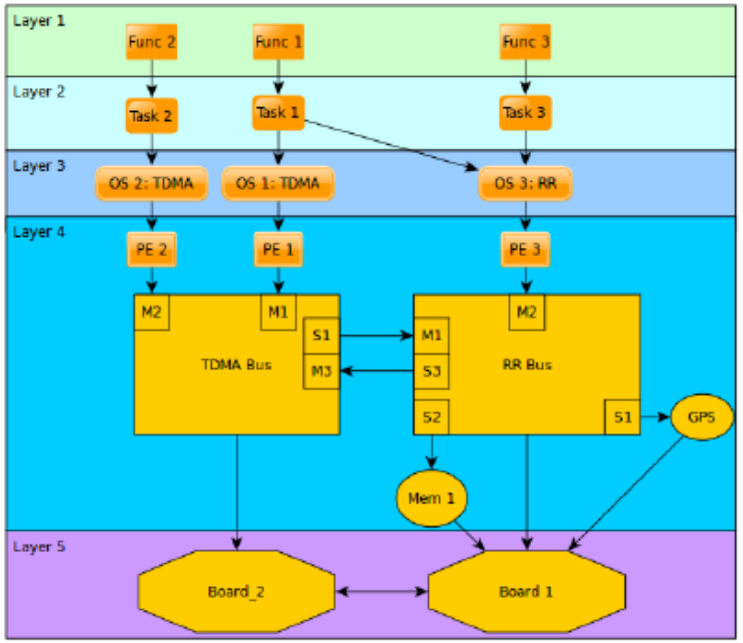
Outline

- Problem Description
- Idea of resolution
- Implementation
- Conclusions and What I learned

Problem Description [1]

"Automatize the design space exploration for mixed critical System on Chip."

To make this, we first take an XML file that describes the graphic representation of all the possible design decisions to make.

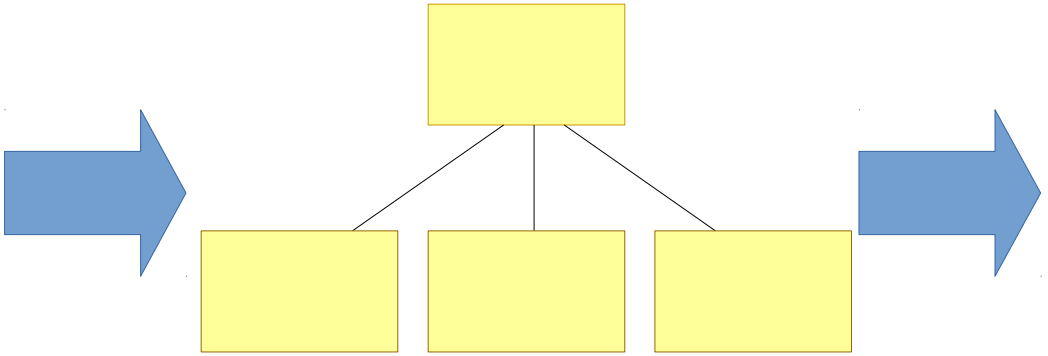


```
<root>
    <components>
    ...
    </components>
    <edges>
    ...
    </edges>
</root>
```

Problem Description [2]

From this XML file, we explore the decisions tree and validate each decision with a tool called iNARK.

```
<root>
    <components>
    ...
    </components>
    <edges>
    ...
    </edges>
</root>
```



The problem is:
How to explore this tree?
How can we generate all the possible descriptions for each decision taken?

The Idea [1]

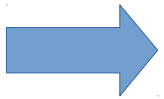
· Since the work is focused only on the mapping phase of SoC design, we consider only design decisions between Layer 2 and Layer 3 (Tasks to OSs).

· Every Task is composed by a Name and an array of edges that starts from a Task and ends at an OS.

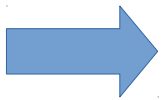
· We have to generate a design description file for every decision taken.

·That file is an XML file (like the one given in input) that describes the connections between components.

· A definition: “configuration of a Task”: is a subarray of the edges array that starts from a common Task.



We **can have 2^n configurations** for each Task where n is the nuber of edges in the edges array.

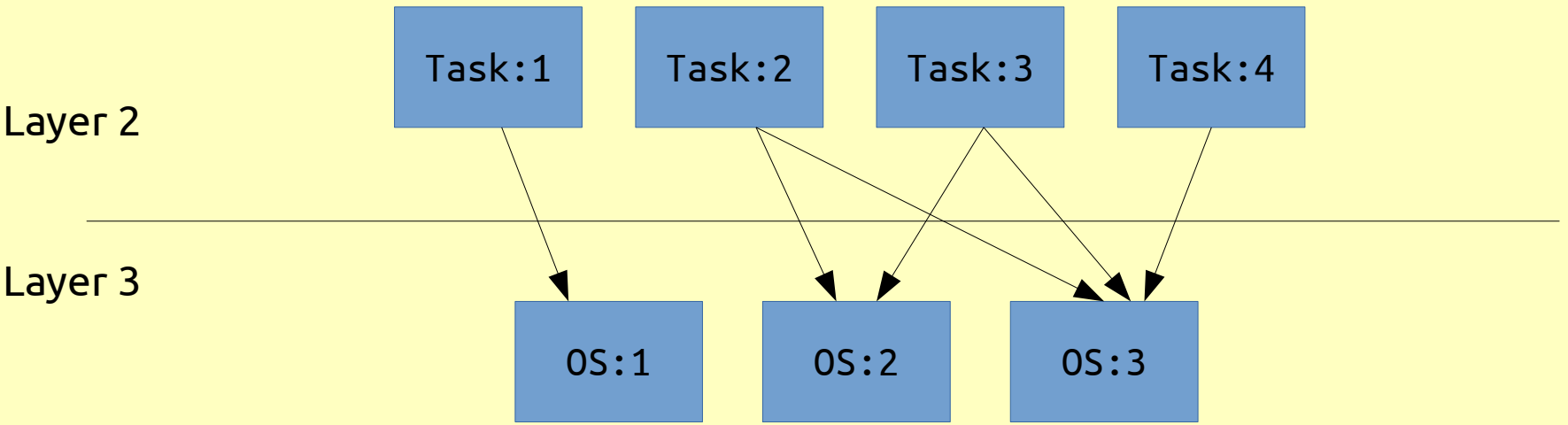


A **configuration** can be **seen as a binary number**, used as a mask to select or not every single edge of a Task.

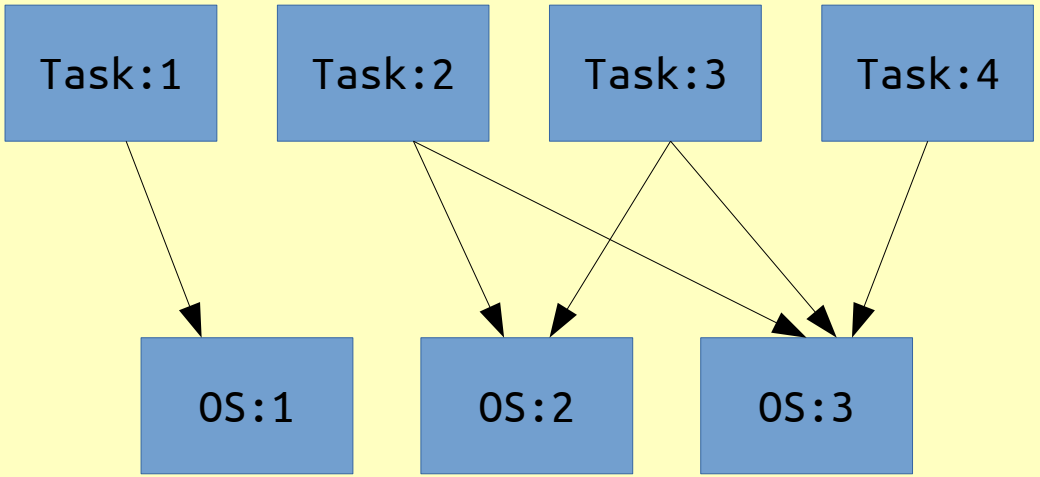
The Idea [2]

We can generate all the XML files of the tree, by incrementing recursively the configurations of the given Tasks.

An Example:



The Idea [3]



· it can have 2^| E | = 4 configurations
#0 #1 #2 #3 but we can consider only 3 of them (#0 is dropped.)

Considering only Task:2, we can say that:

· His array of edges is composed like that:
E = { <e1 - from: Task:2 – to: OS:2>,
<e2 – from: Task:2 – to: OS:3> }

So | E | = 2.

· the remaining configurations are represented as a binary number:
#1 → 01 select edge 1
#2 → 10 select edge 2
#3 → 11 select edge 1 and 2

When we have to generate an XML file, we look at the configuration mask of each Task and for each bit we print/don't print the XML tag of the corresponding edge.

The Idea [4]

The algorithm is described as:

```
number_of_files = GenerateFile()          # generate the first file with the standard
                                            configurations
h ←- taskListHead()

While h is not null do:

    h ←- next_task_that_can_increment()   # if a task cannot increment, this
                                            procedure will reset his configuration.
    If h is not null then

        increment_configuration(h)
        number_of_file = GenerateFile()

    End If

    H ←- taskListHead()

End While
```

The Idea [5]

To explain the behaviour of the procedure “next_task_that_can_increment()” here it is an example of evolution of the tasks configurations:

	T1	T2	T3	T4	
1)	1(1)	1(01)	1(01)	1(1)	→ task 2 is returned, he can increment.
2)	1(1)	2(10)	1(01)	1(1)	→ task 2 is returned, he can increment.
3)	1(1)	3(11)	1(01)	1(1)	→ now task 2 cannot increment, reset conf. and check task 3, task 3 can increment, so he will be returned.
4)	1(1)	1(01)	2(10)	1(1)	→ now task 2 can increment! Return task2!
5)	1(1)	2(10)	2(10)	1(1)	
…					
9)	1(1)	3(11)	3(11)	1(1)	→ next algorithm call will go through all the list until T4 is reached. Task 4 cannot increment so the next task is considered, but the next task is null! So null will be returned.

Implementation [1]

For a complete description, the code is available at:
https://www.mediafire.com/?k54dl9dhpb952kh

The project is divided in two main parts:

· **XML files generation** that is in charge to generate all the xml files, is written in Python because it's a powerful language and can parse/write easily big XML files with high performances.

· **XML files validation** that is in charge to validate each generated file, is written with the Bash shell scripting language because it's more easy to handle files and directories using this language.

Implementation [2]

File generation Loop (python):

```
#getting the task list head
result = tasksArray[0]
while result!=None :

    #take the next task that can change
    #his configuration
    result = result.getNextAvailable()
    if result!=None :

        #change his configuration
        result.setNextConfiguration()

        #generating a new file by passing
        #tasksArray and the other stuffs.
        filecont =
        fileGeneration(filecont, filename, dirname, tasksArray,
                       xmlEdges, outputRoot, outputEdges)

        #clearing edges for the next file
        outputEdges.clear()
        result = tasksArray[0] # taking the head so we can continue.
```

Implementation [3]

This is the main code used to handle Tasks configurations logic (python):

This is the line of code used to generate the configuration mask:

```
configurationMask = list(bin(self.currentConfiguration)
                         [2:].zfill(self.getNumberOfBits()))
```

And this is the getNextAvailable() that returns the next Task that can increment his configuration:

```
    # gets the first Task that can change in the Tasks linked list
    def getNextAvailable(self):
        if self.canIncrement(): # true if currentConfiguration < 2^#edges
            return self
        else:
            self.resetConfiguration() # sets the currentConfiguration to 1
            if self.next != None:
                return self.next.getNextAvailable()
            else:
                return None
```

Implementation [4]

The bash script is a configurable one, configuration file is d1.cfg

The script is callable by typing:

`./d1.sh [input xml file path] [output directory path]`

```
for filename in $dirname/*xml; do
    #for each file, check all the high-low tasks pairs
    allok = 1
    for low in ${low_priority_tasks[*]}; do
       for high in ${high_priority_tasks[*]}; do
            #call iNARK for this combination

            ./$iNARK_directory/iNARK --input $filename --source $low
                                     --target $high    --type $iNARK_type
                                     --depth $iNARK_depth

            if [ "$?" -ne "0" ]
            then
                    allok=0
            fi
        done
    done

    …
done
```

This script first call fileGenerator.py and then, for all generated file it will verify the file by calling iNARK.

If at the end of the verification process the flag “allok” is still 1, the file is moved in the “accepted” directory. Otherwise it's moved in the “rejected” directory.

Conclusions and What I have learned

Some conclusions:

· Considering the configuration of a task as a binary number, I have observed that the total number of generated files is computed as:

$$files\_number = \prod_{i=1}^{|T|} (2^{|E_i|} - 1)$$

Where:
·*i* is an index that indicates the i-th task in the tasks set.
·*|T|* is the tasks set
·*|Ei|* is the cardinality of the edges array for the task i
· "-1" is because we don't consider the configuration #0 that is the empty configuration.

One thing that I have learned is to use the powerfoul programming language Python and I have studied some aspects of SoC design I did not know before starting the project.

Thanks for your attention.
Davide Cremona – davide.cremona@mail.polimi.it