Advanced Computer Architectures
Automatic Design Space Exploration for System on Chip

Davide Cremona

May 7, 2016

Abstract

This work is focused on the cration of an automatic tool to explore the design space given a general description of a System on Chip and prune infeasible configurations. In the following chapters we will be given a general description of the idea behind the tool and a detailed description of the source code. It will be also described the results obtained with some screenshots of the tool running.

1 Introduction

In this section will be given a small introduction on this work and his purpose.

1.1 Scope

The scope of this work is to automatize the design space exploration for mixed critical System on Chip. This is made by creating a tool that reads an XML file that describes all the design decisions that are not yet taken and produces multiple XML that represents all the possible decisions. The second step of this project consists in taking all the generated files and verify the feasibility of every single configuration with the use of a software called iNARK, as described in the paper *Early Stage Interference Checking for Automatic Design Space Exploration of Mixed Critical Systems*[1]

1.2 Implementation Decisions

The project is divided in two main parts: **XML files generation** and **verification**. The two parts are implemented in two different scripting languages:

- **XML File Generation**: this script ("fileGenerator.py") is implemented in Python because it's a very simple language and can handle large amount of data with high performance.

- **Verification**: this script ("d1.sh") is implemented using the bash shell scripting language because it has to iterate through all the generated files and has to call (./ call) iNARK to verify every single generated file.

2 The File Generation Script

In this section, the "fileGenerator.py" script will be described and analyzed. The script is available for the download at `http://www.mediafire.com/download/97zkno8562e983b/fileGenerator.py`

2.1 The Idea

The idea is to parse the input XML file using the module ElementTree[2] to:

1. Create a Task instance for each task described in the XML file
2. Add to each Task the edges that starts from that Task
3. For each possible configuration, output a new XML file in the user-specified directory.

To manage the generation of each XML file, an array of xml "edge" tags is stored in each Task object.
To continue this description, it's necessary to give a definition to the concept of "configuration" of a Task.
A **configuration** is a subset of the elements of the edges array stored in a single Task.
The current configuration of a Task is represented as an integer variable that can have values from 1 to $2^n - 1$ where 'n' is the number of edges in the array (-1 because the configuration #0 is not considered). This is because each configuration is seen as a binary number: the configuration '1' of a task with 2 edges is translated in the binary number '01'. Using this binary number as a mask, it is possible to select only the edges of the current configuration from the edges array and to put only these edges in the output XML file.

2.2 Source Code

The Source code of the File Generation script is divided in two files:

- fileGenerator.py that is the main script
- Task.py that is the class representing the tasks

fileGenerator.py

This script is in charge to generate an XML file for each decision in the design space.
After some controls about the arguments and the existance of the input file, it will use the ElementTree[2] module to parse the XML input file:

```python
    #get all the xml in one tree
    xmlTree = ET.parse(xmlPath)
    #gets the root
    xmlRoot = xmlTree.getroot()
```

Once the file is parsed, a dictionary[4] (hash map) of components names is populated:

```python
    # getting all the names, divided by layer.
    #(needed for the last stage -> file generation)
    xmlNames = []
    textNames = {}
    for i in [0,1,2,3,4]:
      textNames[i] = []
      namesLayerXQuery = "./"+TAG_COMPONENTS+"/"+TAG_COMPONENT+"["+TAG_LAYER+"='"+str(i+1)+"']/"+TAG_NAME
      xmlNames.append(xmlRoot.findall(namesLayerXQuery))
      for xmlName in xmlNames[i]:
        textNames[i].append(xmlName.text)
```

Then the dictionary[4] of tasks is created by reading the names binded to the second layer; Edges tags are stored in a list of Element[2] objects and added to the corresponding task. Finally, every task in the tasks dictionary[4] is linked to the next task to create a structure like a linked list.

```python
    #puts all the tasks in a dictionary
    tasks = {}
    for name in xmlNames[int(layer)-1] :
      tasks[name.text] = Task(name.text)

    # gets all the edges
    edgesXQuery = "./"+TAG_EDGES+"/"+TAG_EDGE
    xmlEdges = xmlRoot.findall(edgesXQuery)

    # adds edges to the corresponding task
    for edge in xmlEdges :
      taskName = edge.find(TAG_FROM).find(TAG_NAME).text
      if tasks.has_key(taskName) :
        tasks[taskName].addEdge(edge)

    #adjusting tasks list to have a pointer to the next task
    for prev, item, nxt in previous_and_next(tasks.values()) :
        item.next = nxt
```

The output directory is created if it does not exists and the header for each XML file is created:

```
    #preparing the output directory
    if not os.path.isdir(dirname):
    #check if output directory exists
      os.makedirs(dirname)                                  #
   if not, create a new one

    #creating the output XML
    outputRoot = ET.Element(TAG_ROOT)
    #create root
    outputRoot.append((xmlRoot.find(TAG_COMPONENTS)))
    #copy components
    outputEdges = SubElement(outputRoot, TAG_EDGES)
    #create edges
```

Now, some utilities variables are initialized (file counter, files names prefix, array of tasks) and the first file is generated:

```
    #basic filename
    filename = "out"
    #counts the generated files
    filecont = 1
    tasksArray = tasks.values()

    #generating the first xml file that is the first
   configuration
    #(all the tasks has been initialized with the
   first configuration
    # when they are created [Task()])
    filecont = fileGeneration(filecont, filename,
   dirname, tasksArray, xmlEdges, outputRoot,
   outputEdges)
```

At the end of the script there is the file generation loop. For each iteration the following instructions are executed:

1. Take the head of the tasks list and search for the first task that can increment his configuration.
2. If the task that has been returned is not null (None, in Python), then change his configuration and generate a new configuration file.
3. After the file generation, the list of edges is cleared to prepare a new file.

The loop will end when there are no more tasks that can change. Here it is the loop code:

```
#getting the task list head
result = tasksArray[0]
while result!=None:
  result = tasksArray[0] #needed due to looping problems

  #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()
```

There are two main functions in this script:

- **previous_and_next** That is in charge to return a list of elements composed by three tasks: prev, item and next. This function is used to convert the tasks dictionary[4] in a lined-list like structure. This is the code:

```
def previous_and_next(some_iterable) :
  prevs, items, nexts = tee(some_iterable, 3)
  prevs = chain([None], prevs)
  nexts = chain(islice(nexts, 1, None), [None])
  return izip(prevs, items, nexts)
```

- **fileGeneration** That is in charge to write out every single XML file. It puts the edges in order of layer and then write the file. This is the code:

```
def fileGeneration(filecont, filename, dirname, tasksArray, xmlEdges, outputRoot, outputEdges):
  #start file generation
  # copy layer 1 edges
  for edge in xmlEdges:
    if edge.find(TAG_FROM).find(TAG_NAME).text in textNames[0]:
      outputEdges.append(edge)
  # generate layer 2 edges
  for task in tasksArray:
```

```
  for edge in task.getConfiguration():
    outputEdges.append(edge)
# copy other layers
for edge in xmlEdges:
  name = edge.find(TAG_FROM).find(TAG_NAME).text
  if name not in textNames[0] and name not in
 textNames[1]:
    outputEdges.append(edge)
physicFile = open(dirname+"/"+filename+str(
 filecont)+".xml",’w+’)      #open file
filecont = filecont+1 #increment file counter
physicFile.write(tostring(outputRoot))
    #write file
outputEdges.clear()
return filecont
#end file generation
```

Task.py

This file contains a class that describes a generic Task. The main methods of this class are:

- **getConfiguration()**: Returns an array of xml edges corresponding to the current configuration, using a binary number as a mask to select the right edges.

```
# returns a list of edges wrt the current
 configuration number.
def getConfiguration(self) :
  configurationMask = list(bin(self.
 currentConfiguration)[2:].zfill(self.
 getNumberOfBits()))
  returnConfig = []
  i=0
  for edge in self.edges :
    if configurationMask[i]==’1’ :
      returnConfig.append(edge)
    i=i+1
  return returnConfig
```

- **setNextConfiguration()**: This methods simply checks if the task has reached the maximum configuration and, if not, it will increase the self.currentConfiguration variable.

- **getNextAvailable()**: This is the main method of the class. It allows the main script to generate every time a different XML file. This is the flow:

1. If the current task can increment his configuration, then the method will return the current task.
2. If he cannot increment, then he has to reset the configuration (return to configuration #1); Check if he is not the last task in the list. If the check returns true, then the next task available is null (None), otherwise the next task available is the one returned by calling getNextAvailable() from the next task in the list.
   The code:

```
def getNextAvailable(self):
  if self.canIncrement():
    return self
  else:
    self.resetConfiguration()
    if self.next != None:
      return self.next.getNextAvailable()
    else:
      return None
```

3 The Final Tool

References

[1] Emanuele Vitali, Gianluca Palermo. *Early Stage Interference Checking for Automatic Design Space Exploration of Mixed Critical Systems*, Politecnico di Milano - Dipartimento di Elettronica, Informazione e Bioingegneria.

[2] The Python Standard Library. *The ElementTree XML API*, `https://docs.python.org/2/library/xml.etree.elementtree.html`

[3] BASH Programming. *BASH Programming - Introduction HOW-TO* `http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html`

[4] Python Dictionary. `http://www.tutorialspoint.com/python/python_dictionary.htm`