GSOC2020

Release 0.0.1

Aug 30, 2020

Contents:

Google Summer of Code

CHAPTER 1

Motivation

I have been working on RTA related field for the past 1.5 years during my master program, and this time GSoC provides me an opportunity to extend that knowledge into more detail and complicated with a great mentor/supervisor. Prior to this GSoC, I only know 1 method to calculate response time of a task in unicore using recurrence relation.

However, that method is even older than me, and not very optimal in my opinion. I want to find a faster, more optimal way to calculate response time where all task would meet it deadline by configure its preemptive type and its environment. Furthermore, APP4MC with Amalthea model already have those preemption properties needed for the implementation, this reduced greatly the orientation time needed to get familiar with new platform/software since I have already used it since 2019

This led me to my decision choosing this topic: **Non-Preemptive / Limited preemptive in Response Time Analysis**

With all the software related material sorted out, all I needed to do left is to find how to calculate response time in 2 different environments: non-preemptive, and limited preemptive.

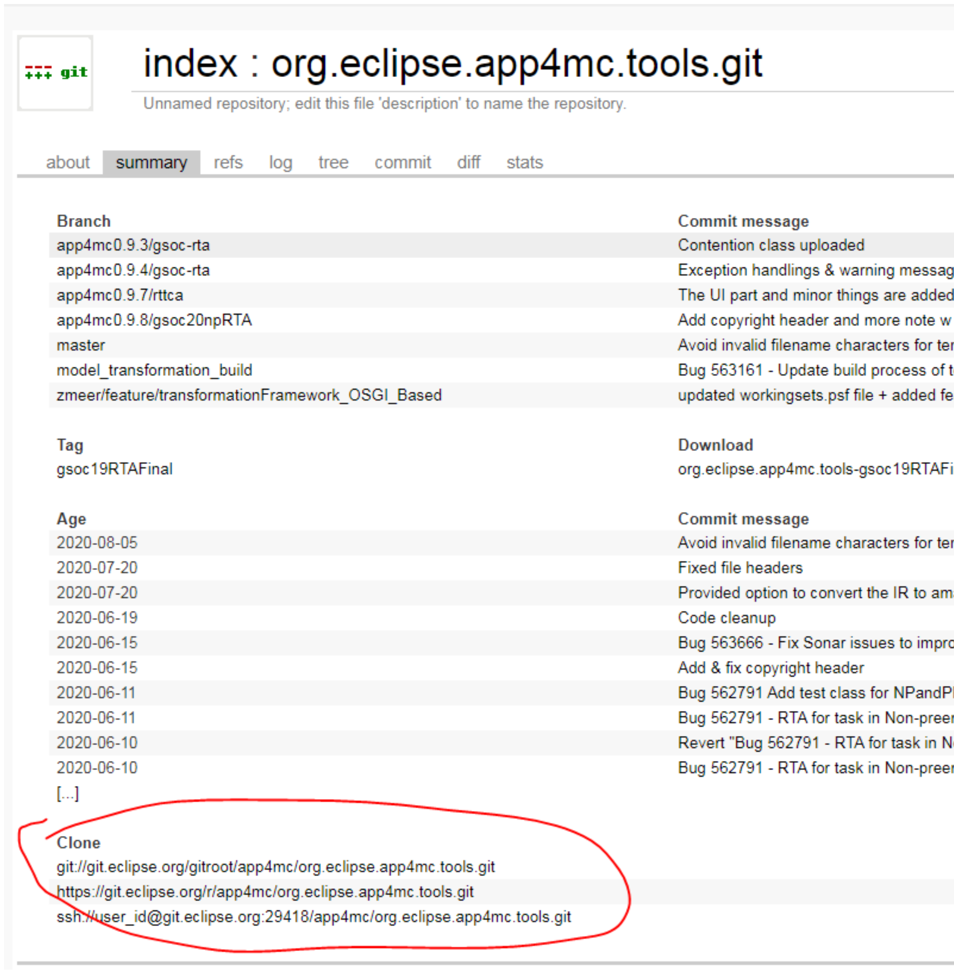
CHAPTER 2

Content

2.1 How to use

First of all, you will need to clone the big repo from this link

—> https://git.eclipse.org/c/app4mc/org.eclipse.app4mc.tools.git/ <—



index : org.eclipse.app4mc.tools.git

Unnamed repository; edit this file 'description' to name the repository.

Branch	Commit message
app4mc0.9.3/gsoc-rta	Contention class uploaded
app4mc0.9.4/gsoc-rta	Exception handlings & warning messag
app4mc0.9.7/rttca	The UI part and minor things are added
app4mc0.9.8/gsoc20npRTA	Add copyright header and more note w
master	Avoid invalid filename characters for ter
model_transformation_build	Bug 563161 - Update build process of t
zmeer/feature/transformationFramework_OSGI_Based	updated workingsets.psf file + added fe

Tag	Download
gsoc19RTAFinal	org.eclipse.app4mc.tools-gsoc19RTAFi

Age	Commit message
2020-08-05	Avoid invalid filename characters for ter
2020-07-20	Fixed file headers
2020-07-20	Provided option to convert the IR to am
2020-06-19	Code cleanup
2020-06-15	Bug 563666 - Fix Sonar issues to impro
2020-06-15	Add & fix copyright header
2020-06-11	Bug 562791 Add test class for NPandPI
2020-06-11	Bug 562791 - RTA for task in Non-preer
2020-06-10	Revert "Bug 562791 - RTA for task in N
2020-06-10	Bug 562791 - RTA for task in Non-preer
[...]	

Clone
git://git.eclipse.org/gitroot/app4mc/org.eclipse.app4mc.tools.git
https://git.eclipse.org/r/app4mc/org.eclipse.app4mc.tools.git
ssh://user_id@git.eclipse.org:29418/app4mc/org.eclipse.app4mc.tools.git

Here is the ssh html location if you can't find it

You need to sign up an eclipse account to be able to clone it though.

Check out the tag gsoc20MPRTAFinal.

This is the tag that contains the GSoC2020 implementation by the time the Project is submitted. It may be merged to master soon, too.

The content of this tag maybe updated after GSoC submission, I will keep this document updated if more work is done regard this method.

1. Under responseTime-analyzer > plugins > src >. . . > gsoc_rta folder

You will find `NPandPRta` class. This is the implementation source file. One can calculate task's response time in different environment using this.

1. Under responseTime-analyzer > plugins > src >. . . > gsoc_rta >'test' folder, there is `NPandPNumerical` class.

This is a numerical example on how functions/equations used in 'NPandPRta' work.

1. Under responseTime-analyzer > plugins > src >. . . > gsoc_rta folder

`Blocking` is also located here, using this will allow you to calculate local and global blocking time of task

Check out the **Implementation functions** part if you want to know what functions is used for

2.1.1 NPandPRTA

Important remarks:

- In this implementation, the function allow all higher priority task in the same core to be able to preemp lower priority task at runnable bound. No preemption threshold is implemented here. The implementation for cooperative part used the equation 12 in this ref, but with the change of j:P_j > P_i, not j:P_j > \theta_i Preemption threshold did prove to be superior in several research in response time analysis. However the main focus of this project is the cooperative preemption, not threshold. This is one of the main future work if I ever return to this topic.

There is something need to be mentioned before you try using this class, it is created based on WATERS2019 model, the functions are tested using that model. But you should be able to ultilize this class without many problem as long as you provided 3 input parameters:

- Amalthea model - The model that you will use to ultilize this method, this should be given when you create the class object
- Integer array (ia) - This is a representation of how task is allocated to cores, the location of each element represent task, and the value represent core. I.e. {0,2,3,1,1,2} : first task is assigned to first core of the model, 2nd task is assigned to 3rd core, 3rd task is assigned to 4th core and so on
- Task - the task you want to calculate its response time

Also additionally there are 3 other parameters:

- executionCase : TimeType.WCET - just put this like this, since you would want to calculate worst case response time most of the time, change to BCET if you want something different.
- pTypeArray: a customized array that you can use to define tasks' preemption type. Leave null if don't use. Check javadoc for more info.
- usePtypeArray: boolean variable to announce whether you want to use pTypeArray or not. Leave false if don't use.

Below are the important functions that you will probably use most of the time. For the list of all function, refer to the javadoc, Implementation functions or open the class, I left lots of comment there

getRTAinMixedPreemptiveEnvironment

Calculate response time of task in mixed environment. Drop the task, the integer mapping array, and the model (again this class is made mainly for WATERS2019 derived model, but it should work on other as well) and you get your response time. I also opt in an option where you can input your own preemptive type array. Where you can change task's preemptive type without changing it in the model.

setSchedubilityCheck

This function set a boolean variable where you can enable/disable schedubility check. Which means if you set to false. Every RTA functions will return the value without checking whether that response time bigger than task's period or not.

getResponseTimeViaLevelI

Calculate resposne time of task in preemptive environment via level-i busy window technique. Pretty vanilla/basic implementation.

getResponseTimeViaRecurrenceRelation

Calculate resposne time of task in preemptive environment via recurrence relation. Again very basic execution of how response time is calculated Should give the same result as response time level-i.

getPureExecutionTime

Using the well-known semantics, where task is run as follow: READ -> EXECUTION -> WRITE Calculate all of the element from each step, sum all of them and we have task's execution time.

2.1.2 Blocking

`<Blocking.java>`

Blocking analysis, calculate blocking time of semaphore(critical section) when they are exist, or else calculate time other tasks have to wait due to global resource occupancy (task had to wait because other task is reading/writing label)) Same as the `NPandPRta.java`, this class also created based on WATER 2019 model.

Again I only listed important/useful function. For more info, please refer functions' javadoc and the implementation functions part for more details

getGlobalBlockingTime

Calculate task's global blocking time (time blocked by task from other cores) due to semaphore lock. If there is no semaphore, the function will calculate blocking time due to resource being read/write by other task. The blocking policy is Priority Ceiling Protocol FYI

getLocalBlockingTime

Same with getGlobalBlockingTime, but this time we calculate blocking time due to local task (task within same core)

2.2 Function introduction

2.2.1 Function inside the class `NPandPRTA` and its usage:

```
setSchedubilityCheck(boolean schedubilityCheck)
```

set to false will return RTA result without checking its schedubility (only check core ultilization). This is used to avoid return 0 for task's RT that exceed its period/deadline. Mainly use for test class. Recommend to set it to true when partitioning

```
getRTAinMixedPreemptiveEnvironment(Task thisTask, int[] ia, TimeType executionCase,␣
↪int[] pTypeArray, boolean usePtypeArray):
```

calculate task's response time in different environments: fully preemptive, non-preemptive, deffered preemptive, and a mixed of all (core have tasks with different preemptive type). `pTypeArray` is preemptive type array for task, used when developer want to do shuffle preemptive type via code rather than edit the model. -Array same length with `int[]ia` array. index value meaning : 0 = preemp, 1 = non-preemptive, 2 = cooperative type -Set to null when use model's configurated preemtive type `boolean usePtypeArray` set to true if use pTypeArray Mechanism within this class

```
  Step 1: determine what preemptive type thisTask is
  Step 2: find out whether there is blocking from lp task (cuz np/c type task)
  Step 3: choose the method by these criteria
                   - if (taskType == preemp/cooperative) && (blockingTime == 0)
                              => use normal level-i (getResponseTimeViaLvI)           ␣
↪                    (1)
                   - if (taskType == preemp/cooperative) && (blockingTime !=0)
                       => use level-i with preemptive mixed environment␣
↪(getPreempCoopMixedTaskRTA)    (2)
                              finishing time here is different than the above
                        finishingTime = startTime + execution time + preemption time
```

(continues on next page)

(continued from previous page)

```
                              preemption time = time thisTask get preempted by other␣
↪higher priority task in the same core.
                  - if (taskType == non-preemp))
                      => use level-i with blocking (getNonPreemptaskRA)                ␣
↪             (3)
                      blockingTime doesn't matter here since even if there is no␣
↪blocking time, the equation still hold true
                      important part is taskType is non-preemp which make␣
↪finishingTime = startTime + execution time;
```

```
getResponseTimeViaRecurrenceRelation(Task thisTask, int[] ia, final TimeType␣
↪executionCase):
```

calculate response time of task using classic recurrence relation method.

```
getResponseTimeViaLvI(Task thisTask, int[] ia, TimeType executionCase):
```

calculate response time of task using level-i method (window technique).

```
getNonPreemptaskRTA(Task thisTask, int[] ia, TimeType executionCase, int[] pTypeArray,
↪ boolean usePtypeArray):
```

when task is non-preemptive type. Response time will be calculated via level-i technique Since task is non-preemptive -> once it start, it doesn't stop. Our goal here is to calculate task's `start time` .Cuz task's `finishing time = start time + execution time` . We can derive response time by `finishing time - kPeriod`

```
getLowerPriorityBlockingTime(Task thisTask, int[] ia, TimeType executionCase, int[]␣
↪pTypeArray, boolean usePtypeArray):
```

time `thisTask` would be blocked by lower priority task. Mechanism here is to scan all lower priority tasks (`lpTask`) that are allocated in the same core with `thisTask`. Then check their preemptive type, `lpTask` only block`thisTask` if they are non-preemptive or cooperative. `thisTask` only get blocked once, so we take the longest time that task could be block by `lpTask` (i.e lpTask_1 block 10ms, lpTask_2 block 20ms then return 20ms as blocking time)

```
getPreempCoopMixedTaskRTA(Task thisTask, int[] ia, TimeType executionCase, int[]␣
↪pTypeArray, boolean usePtypeArray)
```

When task is not non-preemptive type and under a mixed or cooperative environment . Response time will be calculated via level-i method. This time our calculation is not only focus on start time but also finishing time. Finishing time = start time + execution time + preemption by higher priority task. And after this response time = finishing time - kPeriod.

```
getPureExecutionTime(Task thisTask, int[] ia, final TimeType executionCase):
```

Label access Read + ticks/executions need + Label access Write. No blocking here.

```
getExecutionTime (Task thisTask, int[] ia, final TimeType executionCase):
```

getPureExecutionTime + blocking. Contention, Copyengine, global blocking etc… can be added here.

```
getUltilizationCheck(List<Task> groupTask, int[] ia, TimeType executionCase):
```

check whether it is possible to shove a group of task together in one core. return true when core ultilization value is less than 100%.

2.2.2 Function inside the class `Blocking` and its usage:

```
private void setUpFlagArrayAndHashMap(final int[] iap):
```

This function is used to to figure out whether any task assigned to CPU trigger GPU task or not. And if it does, then the shooting task (cpu task) will have more labels (comes from shot task, or GPU task) Rather complicated for this part. If there aren't any of those behavior then it will return nothing. So don't worry here

```
private List<Task> groupTask(final Task thisTask)

private List<Task> groupTaskWithIA(final Task thisTask, int[] ia)

private List<Task> groupOtherTask(final Task thisTask) {

private List<Task> groupTaskFromOtherCore(final Task thisTask, int[] ia)
```

Functions work as its name, return a list of task that located inside the same core or other cores. Reason behind 2 function groupTask and groupTaskWithIa is because groupTaskWithIA will derive from an input integer array, the other use input integer array when we initialize the class. Result is same, but I will use the groupTaskWithIA when I want to multi thread it.

```
private List<Task> taskSorting(final List<Task> taskList)

private List<Label> labelSortingBySize(List<Label> labelList)
```

task sorting will sort tasks by Rate Monotonic Scheduling (RMS) method, highest priority go first.

labelSortingBySize will rearrange label in the list by its size, biggest go first.

```
private HashMap<Task, List<Label>[]> getReadWriteLabelHash(Task thisTask)
```

put a task in, return a hashmap contain task as key and 2 list as value. Should look like this

Hash = task, [readList,WriteList]

```
private List<Label> getAccessedLabelListWithDuplicate(Task thisTask)
```

Same with getReadWriteLabelHash, but instead of a hash map with 2 list value. I put everything in a list, don't care whether there is duplicate values or not (read and write the same label)

```
private List<Label> getDuplicateBetweenListAndSet(List<Label> thisList, Set<Label>␣
↪thisSet)
```

Simple tool for me to get whatever a list and a set have in common, similar with list.retainAll()

```
private List<Task> getLabelCalledTaskSet(Label thisLabel)
```

Get a list of task that called this specific label, use when we need to identify which task is accessing the same label when finding global/local blocking time.

```
private Time getLocalCrossingLabelAccessTime(Task thisTask, List<Task>␣
↪sameCoreTaskList)
```

get local label access by lower priority tasks in the same core. `CrossingLabel` means several task access same label, visually crossing each other. Just my intepretation here. Not a technical term

```
private Time getGlobalCrossingLabelsAccessTime(Task thisTask, List<Task>␣
↪diffCoreTaskList)
```

Same with getLocalCrossingLabelAccessTime, only this time we concern about task located in other cores

```
private Time getLabelAccessTimeWithTask(Task thisTask, final Label thisLabel,␣
↪TimeType executionCase)
```

Mainly use for localCrossingLabelAccessTime Check label access time for thisLabel with thisTask Automatically check whether it is read/write access from thisTask to thisLabel, take the longer one if both occured

```
private Time getLabelAccessTime(final Task thisTask, final Label thisLabel, final␣
↪LabelAccessEnum thisType, final TimeType executionCase)
```

calculate read/write label access of thisTask on thisLabel Use for globalCrossing and getLabelAccessWithTask above. Pretty basic function in RTA

```
public List<Semaphore> getSemaphore(final Task thisTask)
```

Use to locate semaphore in task, and put them in a list to cross checking later when calculate global/local blocking

```
public Time getSemaphoreTime(final Task thisTask, final Semaphore thisSema, final␣
↪TimeType executionCase)
```

get critical section length that guarded by Semaphore thisSema for thisTask Calculated by get labelAccess time + ticks (if avaiable) between request and release semaphore.

```
private List<Task> groupSemaphoreLinkedTask(final Task thisTask)
```

Get a list of task that have same semaphore(s) with thisTask, use for global blocking since we may have hundreds of cores

```
public Time maxPriorTaskBlockingTime(final Task thisTask, final TimeType␣
↪executionCase)
public Time otherPriorTaskBlockingTime(final Task thisTask, final TimeType␣
↪executionCase)
public Time otherPriorTaskBlockingTime(final Task thisTask, final TimeType␣
↪executionCase)
private Time normalTaskBlockingTime(final Task thisTask, final TimeType executionCase)
```

normal prior task PCP calculation, for reference and testing. Didn't use for any of the implementation. Not important.

```
public Time getLocalBlockingTime(final Task thisTask, final TimeType executionCase) {
```

Calculate task's global blocking time (time blocked by task from other cores) due to semaphore lock. If there is no semaphore, the function will calculate blocking time due to resource being read/write by other task. The blocking policy is Priority Ceiling Protocol FYI

```
public Time getGlobalBlockingTime(final Task thisTask, final TimeType executionCase) {
```

Same with getGlobalBlockingTime, but this time we calculate blocking time due to local task (task within same core)

```
private int getGPUindex(Amalthea modelp) {
```

This is a generic approach on how we locate GPU task in the integer array used for mapping. Need to be modify if task failed when executed.

Done

2.3 Theory and equation for response time analysis in different environment

- Table of Notation for **Basic RTA**

Description	Symbol
Task i	τ_i
WC Response time	R_i^+
WC Execution time	C_i^+
Period	T_i
Priority	P_i

2.3.1 Fully preemptive environment

- Response time analysis using recurrence relation

Equations of this part comes from [4]. Check Reference for the paper/research

Equation 1: $R_i^+ = C_i^+ + \sum_{j:P_j \geq P_i} \left\lceil \frac{R_{i-1}^+}{T_j} \right\rceil C_j^+$ |

Numerical example: Assume a task (C_i^+, T_i) set {(1,3), (1,5), (1,6), (2,10)}

R_1^0 = R_1^1 = $C1$ = 1 (task with highest priority, hence response time = execution time)

R_2^0 = C_2 + $C1$ = 2

R_2^1 = C_2 + $\left\lceil \frac{R_2^0}{T_1} \right\rceil C_1^+$ = $1 + \lceil \frac{2}{3} \rceil 1 = 2$

R_2^1 = R_2^0 = 2. No further exploration can be done here, τ_2 response time R_2 = 2. This mean, we stop when previous iteration value equal the current iteration value, this value will be the response time of task.

- Response time analysis using level-i busy window,

In preemptive environment, beside recurrence relation method, there is another way to calculate task response time (RT) by calculate response time of several instance of task during its busy period, and pick the longest one among them. Equation from this part comes from [2].

Equation 2: Level-i active period (busy window)

$L_i = \sum_{j:P_j \geq P_i} \left\lceil \frac{L_i}{T_j} \right\rceil C_j$

Equation 3: The number of τ_2's instances to check are:

$K_i = \left\lceil \frac{L_i}{T_i} \right\rceil$

Equation 4: The finishing time of *k*-instance

$f_i^k = \sum_{j:P_j > P_i} \lceil \frac{f_i^k}{T_j} \rceil C_j + (k-1)C_i$

Equation 5: From finishing time, we can calculate response time simply since: Response time = finishing time - arrival time. Where arrival time = k-instance's period.

$R_i^k = f_i^k - (k-1)T_i$

Response time of τ_i is the maximum among all k-instances.

For numerical example, I recommend you check "tutorial on real-time scheduling" by Emmanual Grolleau, his papered is where I check and back test the code flow. First 4 pages is enough to understand how to use level-i busy window.

2.3.2 Non-preemptive environment

For non-preemptive environment, recurrence relation method is not doable, only busy window is usable as far as I know.

We reuse equation 2, 3 to get total number of k instance, only different is we will calculate arrival time, then find finishing time by arrival time + execution time, and use equation 5 for response time:

Equation 6: release time s_i^k of τ_i^k

$s_i^k = \sum_{j:P_j>P_i}(\lfloor\frac{s_i^k}{T_j}\rfloor + 1)C_j + (k-1)C_i$

Then simply finishing time $f_i^k = s_i^k + C_i$ since it is non-preemptive, response time can be calculate again by equation 5.

2.3.3 Limitted preemptive (or cooperative) environment

Window technique again proved to be useful in this scenario, this time we need to introduce blocking time, which is the largest execution time of lower priority runnable (since cooperative is runnable bound). All equation in from this part is from paper [1], check out references tab for more info.

Blocking time

$B_i = max_{j:P_j<P_i}(C_j, r)$

Reuse equation 2,3 again, but include blocking time B_i into equation 2 to find out proper busy period

$L_i = B_i + \sum_{j:P_j\geq P_i}\left\lceil\frac{L_i}{T_j}\right\rceil C_j$

Task τ_i finished time can be derived from equation 4, with a touch of blocking time

$f_i^k = B_i + \sum_{j:P_j>P_i}\lceil\frac{f_i^k}{T_j}\rceil C_j + (k-1)C_i$

And here we can apply equation 5 for response time, pretty simple right?

Notice here: This is true iff we assume all higher priority task can preempt lower priority task as soon as its arrival time come. If you want to introduce some preemption threshold θ_i, follow the below equations.

Equation 7: get task's release time:

$s_i^k = B_i + \sum_{j:P_j>P_i}(\lfloor\frac{s_i^k}{T_j}\rfloor + 1)C_j + (k-1)C_i$

Equation 8: get task's finshing time:

$f_i^k = s_i^k + \sum_{j:P_j>\theta_i}(\lceil\frac{f_i^k}{T_j}\rceil - (\lfloor\frac{s_i^k}{T_j}\rfloor + 1))C_j$

And then applied the same equation 5 for response time.

developer note: sorry I didn't include any numerical example in these case, it's rather lengthy to do so, but if you understand how level-i work, these equation wouldn't be difficult at all

2.4 Plan and milestone

2.4.1 Phase 1

May 4 - June 23, 2020 (Community Bonding Period, Coding part 1)

- Structuring the implementation process:
    1. Non-preemptive response time analysis (RTA)
    2. limited/cooperative RTA
    3. a mixed of non-preemptive, limited/cooperative and preemptive RTA
- Retesting level-i and recurrence relation method in calculating response time
- Implementing non-preemptive RTA in APP4MC

The commits can be tracked via this bug from bugzilla :

This phase is mainly for getting familiar with level-i busy window technique in RTA which contribute quite a lots in further exploration regard to RTA, at least non-preemptive RTA was completed during this phase

2.4.2 Phase 2

June 24 - July 21 (Coding - Part 2)

- Implement cooperative RTA in APP4MC
- Create test classes for both non-preemptive and cooperative
- Attempt on mixed preemptive environment RTA

The commits can be tracked via this bug from bugzilla :

After this phase, the skeleton or outline of how data flow and how response time analysis in different cases is formed. From getting info from model and mapping array to source out what environment input task's in, and finally decide what equation/function is used to give out the best solution.

2.4.3 Phase 3

July 22 - August 18 (Coding - Part 3)

- Implement mixed preemptive environment RTA
- Adding getting preemptive type from an customized array, similar with mapping array for the class. This will remove unnecessary time for mouse clicking in order to change preemption type in model explorer
- Implement blocking analysis, based on semaphore and Priority Ceiling Protocol
- Adding blocking due to shared resources if no semaphore/critical section is available within model
- Create test classes, adding javaDoc, documentation

The commits can be tracked via this bug from bugzilla :

Implementation done here, tough time with all the additioning stuff but was fun and interesting.

2.5 Contribute to the community

There are several paper about response time analysis but not many open source code that developer can use as reference or implementation can be found in the internet. This project is my contribution to the open source community in this matter. In this project, you can find method to calculate response time in different preemptive environment: non-preemptive, cooperative, or a mixed of the above.

The class `NPandPRta` can be used and modify as developer desire, I left lots of comments inside the code itself for whoever going to ultilize that for him/herself. Also for whoever want to approach level-i but don't have much knowledge about mathematics, `NPandPRtaNumerical` is the class you should dig in. It consist of the same mechanism/algorithm with `NPandPRta` but input value isn't derived from the model but from an array. Drag down to whatever function you want to use to edit that input value, or leave as it is. Whatever help you understand the data flow and the equation :D.

Happy coding, don't push yourself too harsh fellow developers.

2.6 Repositories

2.6.1 RTA Implementation in eclipse

Click here -> eclipse repo <-

2.6.2 Readthedoc repo

Click here -> github repo <-

2.7 Future plan

2.7.1 Implementing preemption threshold

This already mentioned in [1] while researching for cooperative preemption type and proven to be superior than without using preemption threshold. I would like to implement a threshold variable for each task, prefered a generic/dynamic threshold where each task set (several tasks that located in 1 core) element will have different threshold to ensure its response time alway meet its deadline.

2.7.2 Investigate different scheduling method

The current implementation use Rate Monotonic Scheduling to decide task's priority, it would be cool to see what would happen with the same task set but different scheduling method such as EDF, Fixed priority, round robin etc.

2.7.3 Visualize busy window

Resposne time analysis using busy window method is difficult and annoying just by looking at the equation, visualize the busy window and how each of the task's instances look like within it will help immensely to whoever trying to get used to with the method

2.8 References

My implementation couldn't been finished without these research papers and their authors.

[1]: Ignacio Sanudo, Paolo Burgio and Marko Bertogna "Schedulability and Timing Analysis of Mixed Preemptive-Cooperative Tasks on a Partitioned Multi-Core System".

[2]: Emmanuel Grolleau, "Tutorial on real-time scheduling".

[3]: Robert Höttger, Junhyung Ki, The Bao Bui, Burkhard Igel and Olaf Spinczyk "CPU-GPU Response Time and Mapping Analysis for High-Performance Automotive Systems".

[4]: Mathai Joseph and Paritosh Pandya, 1986 "a standardized response time analysis methodology".

[5]: John P. Lehoczky, "Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines"

If you have any question, contact me via : the.bui003@stud.fh-dortmund.de

Happy coding