CARTS

A Tool for Compositional Analysis of Real-Time Systems

Real-Time Systems Group
PRECISE Center
Department of Computer and Information Science
University of Pennsylvania
August 2009

TABLE OF CONTENTS

INTRODUCTION

As real-time embedded systems are continually increasing complexity, integration becomes a great challenge in their design and development. Managing complexity of the system design is therefore essential for high-assurance and cost-effective development. Component-based design and analysis methodology has consequently been developed and gained its importance over the years as a powerful technique for complexity management, which in turn necessitates compositional analysis frameworks. To facilitate compositional analysis, given a component, one needs to be able to compute the component interface - an appropriate abstraction of the component's timing requirement - that can be used in the system analysis. Further, to enable effective compositional analysis, accurate and efficient interface generation becomes crucial.

What is CARTS ?

To meet the growing needs, we have developed CARTS (Compositional Analysis of Real-Time Systems) as a platform-independent tool that automatically generates resource interfaces needed for the compositional analysis of real-time systems. CARTS is built on top of several interface generation algorithms that were developed by Real-Time Systems Group at the PRECISE center. The tool has a GUI that provides users with easy ways to specify and analyze the system - by using the GUI options or by editing XML files. It is apt for visualizing the generated component interfaces in a tree-like structure, as well as charting the demand- and supply-bound functions of the generated interfaces. At the same time, it is also accompanied by a lightweight command-line option that enables our tool to be integrated with other existing toolchains. In essence, CARTS can be conceived as a handy companion to system designers for analyzing and designing hardware-software architectures of real-time systems in a compositional manner.

Development of CARTS

CARTS is a joined effort of several members in the Real-Time Systems Group at the University of Pennsylvania, under the supervision of Prof. Insup Lee and Prof. Oleg Sokolsky. The algorithm engines were written by Dr. Arvind Easwaran and Jaewoo Lee. The GUI was written by Vinay Ramaswamy. Linh Thi Xuan Phan was involved in the design and management of the GUI development and Sanjian Chen helped with testing the tool.

Acknowledgement

This research was supported in part by AFOSR FA9550-07-1-0216 and NSF CNS-0720703.

PART I

CARTS 1.0 User Guide

1. Getting Started with CARTS

CARTS binary and source code distributions are located at the CARTS website (http://rtg.cis.upenn.edu/carts/). The tool requires Java version 1.6 and above to compile and run, which can be found at http://java.sun.com/javase/downloads/. First, download CARTS to your computer and unzip it.

Running CARTS Binary Distribution. Double-click on the "**carts.jar**" file in the unzipped folder to invoke the application. Alternatively,

- Open a command prompt (Windows) or a shell (Mac OS / Linux).
- At the prompt/shell, type in: **java -jar Carts.jar**

Compiling CARTS Source Code. To compile the source codes, follow the steps below:

- Open a command prompt (Windows) or a shell (Mac OS / Linux).
- At the prompt/shell, go to the source code directory: **cd source**
- Compile the source using ant: **ant compile**
- To run the executable file, type in: **java -cp build Carts**

2. CARTS Input/Output Specification in XML Format

2.1 Input XML file description

The input XML specification to the tool follows the Document Type Description format below.

```
<!ELEMENT system ( component | task )* >
<!ATTLIST system os_scheduler ( DM | EDF ) #REQUIRED > : the whole system's scheduling algorithm
<!ATTLIST system min_period NMTOKEN #IMPLIED > : system's minimum period
<!ATTLIST system max_period NMTOKEN #IMPLIED > : system's maximum period

<!ELEMENT component ( component | task )* >
<!ATTLIST component name NMTOKENS #REQUIRED > : component's name
<!ATTLIST component criticality CDATA #REQUIRED > : used in ARINC-653
<!ATTLIST component vmips NMTOKEN #REQUIRED > : used in ARINC-653
<!ATTLIST component scheduler ( DM | EDF ) #REQUIRED > : component's scheduling algorithm
<!ATTLIST component subtype NMTOKEN #REQUIRED > : used in ARINC-653
<!ATTLIST component min_period NMTOKEN #REQUIRED > : component's minimum period
<!ATTLIST component max_period NMTOKEN #REQUIRED > : component's maximum period

<!ELEMENT task EMPTY >
<!ATTLIST task name NMTOKENS #REQUIRED > : task's name
<!ATTLIST task p NMTOKEN #REQUIRED > : task's period
<!ATTLIST task d NMTOKEN #REQUIRED > : task's deadline
<!ATTLIST task e NMTOKEN #REQUIRED > : task's execution time
<!ATTLIST task jitter NMTOKEN #IMPLIED > : used in ARINC-653
<!ATTLIST task noninterrupt_fraction NMTOKEN #IMPLIED > : used in ARINC-653
```

Consider a sample system represented in the following graphical form.

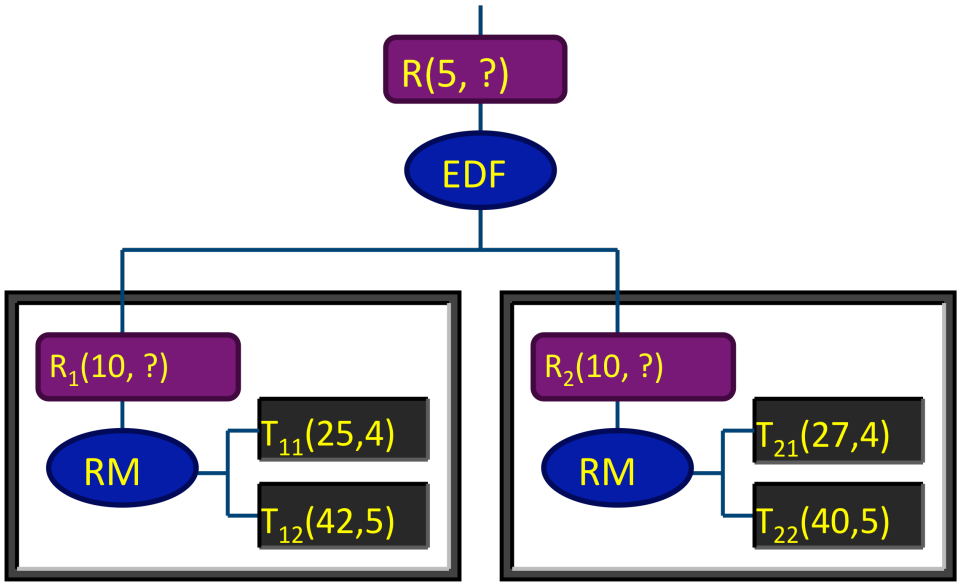


Figure 1. *A sample hierarchy of components*

The input XML description corresponding to the above sample system is given below.

```
<system os_scheduler="EDF" min_period="5" max_period="5">
<component name="Comp1" criticality="?" vmips="0.8" scheduler="EDF"
                    subtype="tasks" min_period="10" max_period="10">
   <task name="T11" p="25" d="25" e="4" jitter="0" noninterrupt_fraction="0" />
   <task name="T12" p="42" d="42" e="5" jitter="0" noninterrupt_fraction="0" />
</component>
<component name="Comp2" criticality="?" vmips="0.8" scheduler="RM"
                subtype="tasks" min_period="10" max_period="10">
   <task name="T21" p="27" d="27" e="4" jitter="0" noninterrupt_fraction="0" />
   <task name="T22" p="40" d="40" e="5" jitter="0" noninterrupt_fraction="0" />
</component>
</system>
```

Figure 2. *The XML input description of the system in* **Figure 1**

2.2 Output XML file description

Resource Model (Resource supply model which is needed to schedule given task set)

1) ***Period***: given period between minimum period and maximum period
2) ***Bandwidth***: calculated optimal(minimal) bandwidth which can schedule given task set under optimal deadline
3) ***Deadline***: calculated optimal deadline which can produce optimal (minimal) bandwidth (Only EDP resource model can change deadline. Otherwise, deadline is same as period)

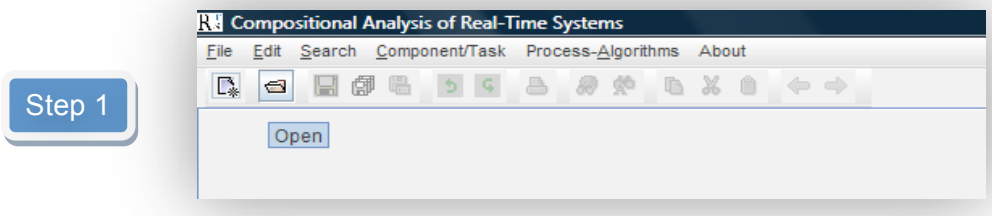
Processed Task Model (A task model which are composed with task set in given component and its sub components)

1) ***Period***: given period between minimum period and maximum period
2) ***Execution Time***: calculated optimal(minimal) execution time of the task which is composed with given component and its subcomponent under optimal deadline
3) ***Deadline***: calculated optimal deadline which can produce optimal (minimal) execution time (Only EDP resource model can change deadline. Otherwise, deadline is same as period)

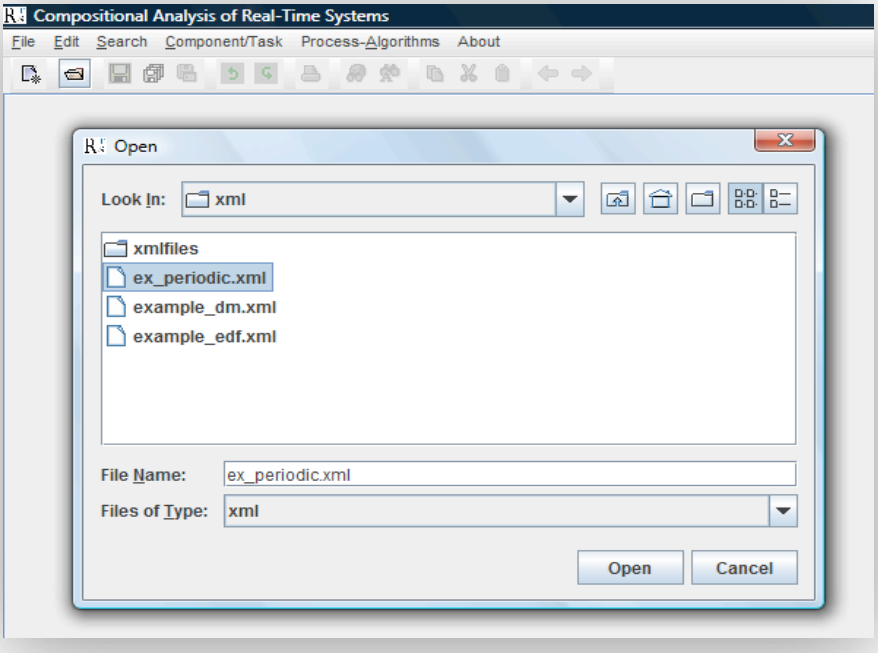
Example: Below is the output result of a system that uses EDP resource model

```
Resource Model
Period : 5.0 , bandwidth : 0.65, deadline : 3.25
Processed Task Model
Period : 5.0, Execution Time : 3.25, Deadline : 5.0
```

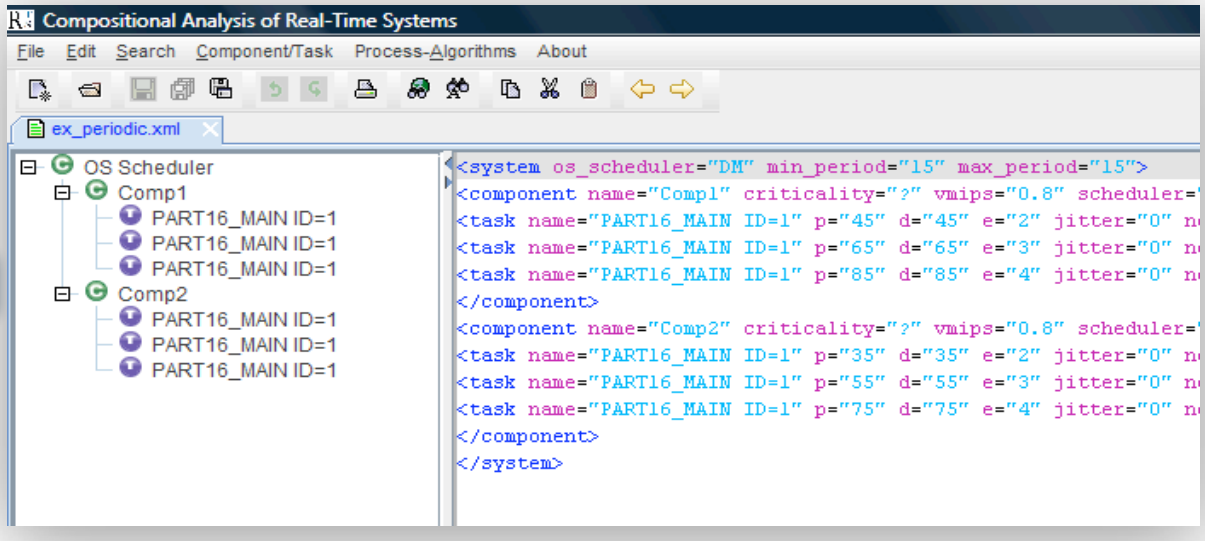
3. Open An Existing CARTS XML File



Step 2



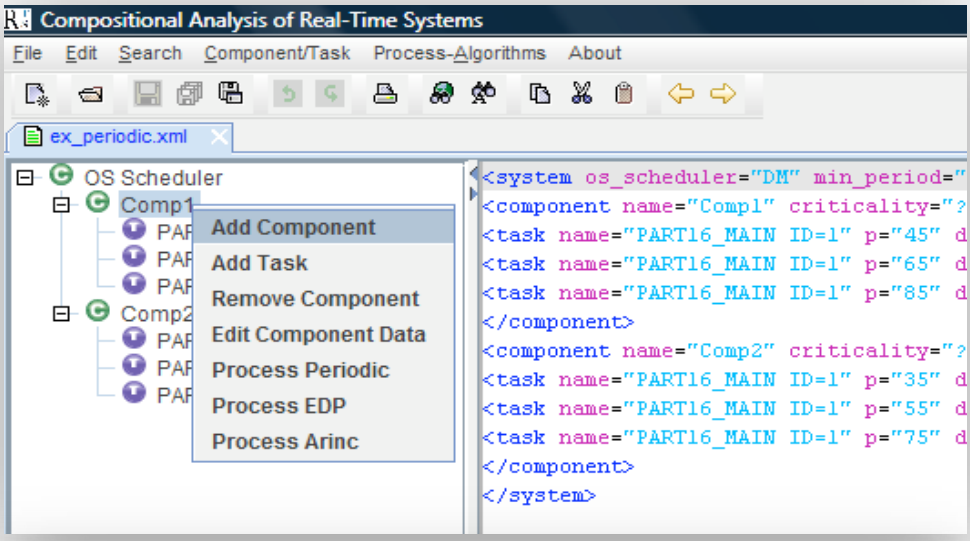
Step 3

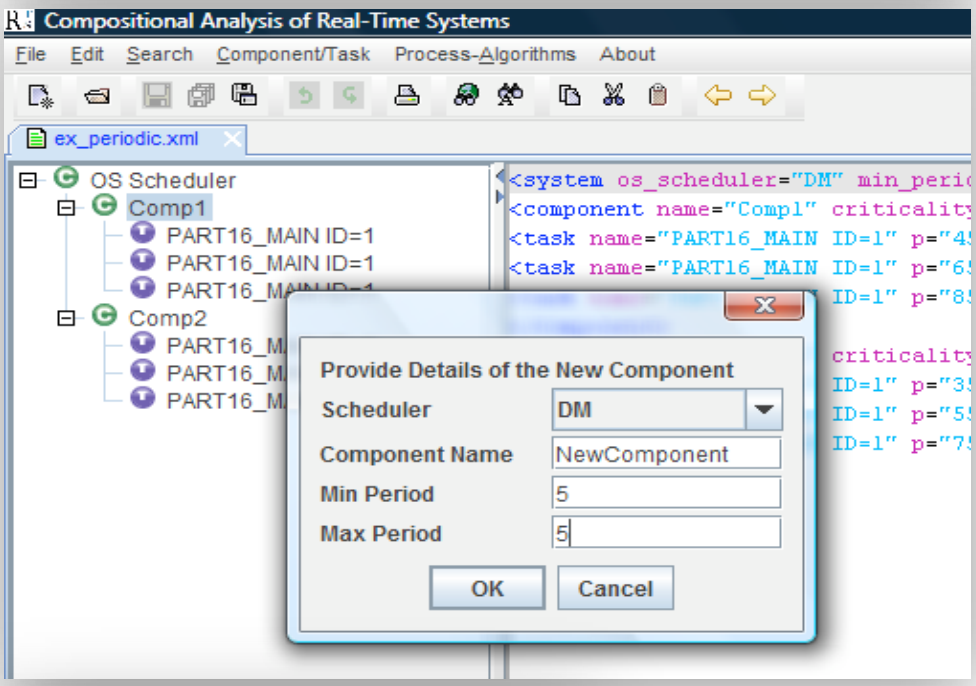


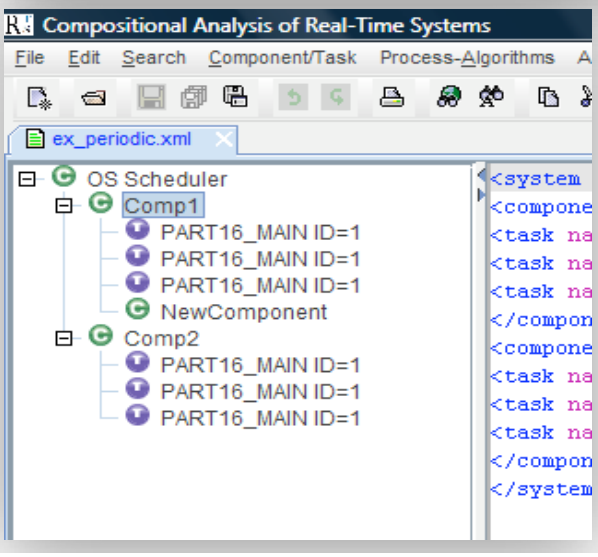
Tree View

CARTS XML File

4. Add A New Component

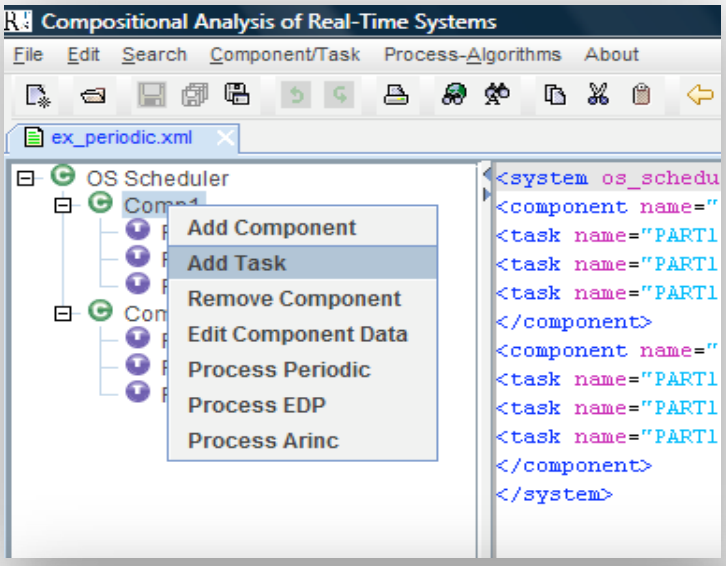




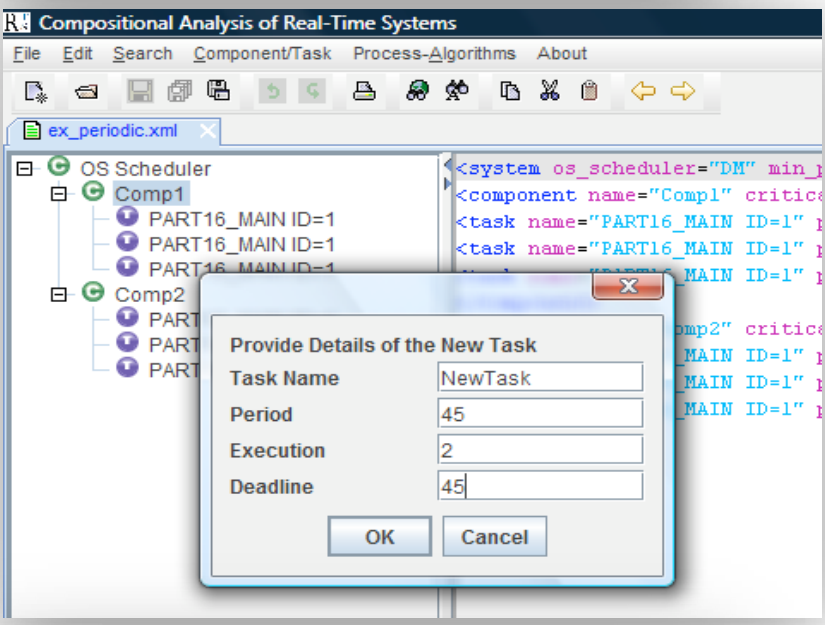


5. Add A New Task

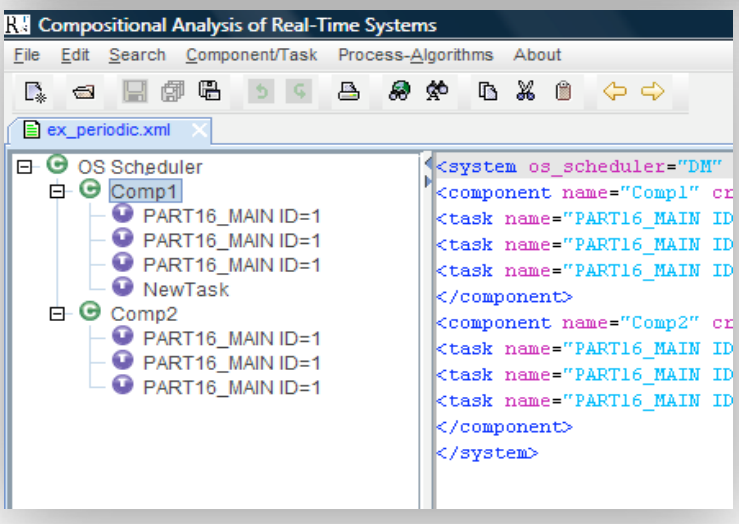
Step 1



Step 2

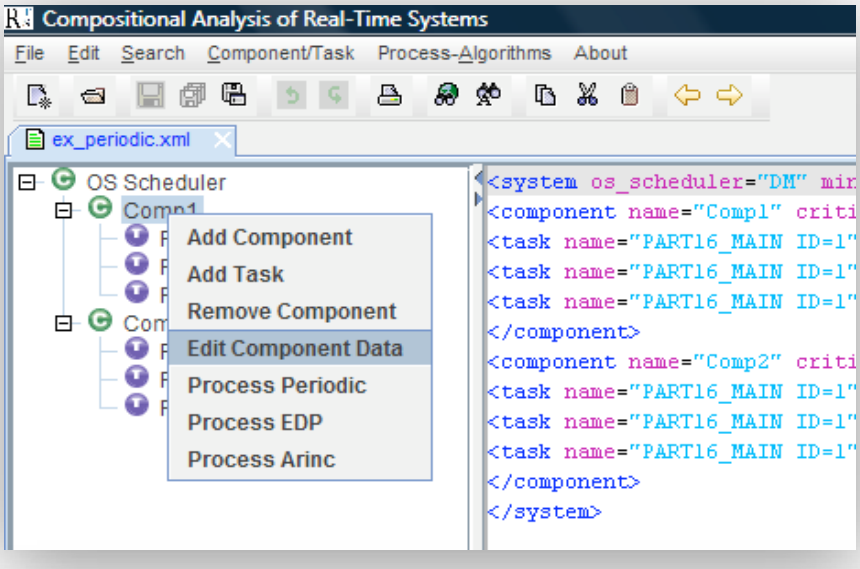


Step 3

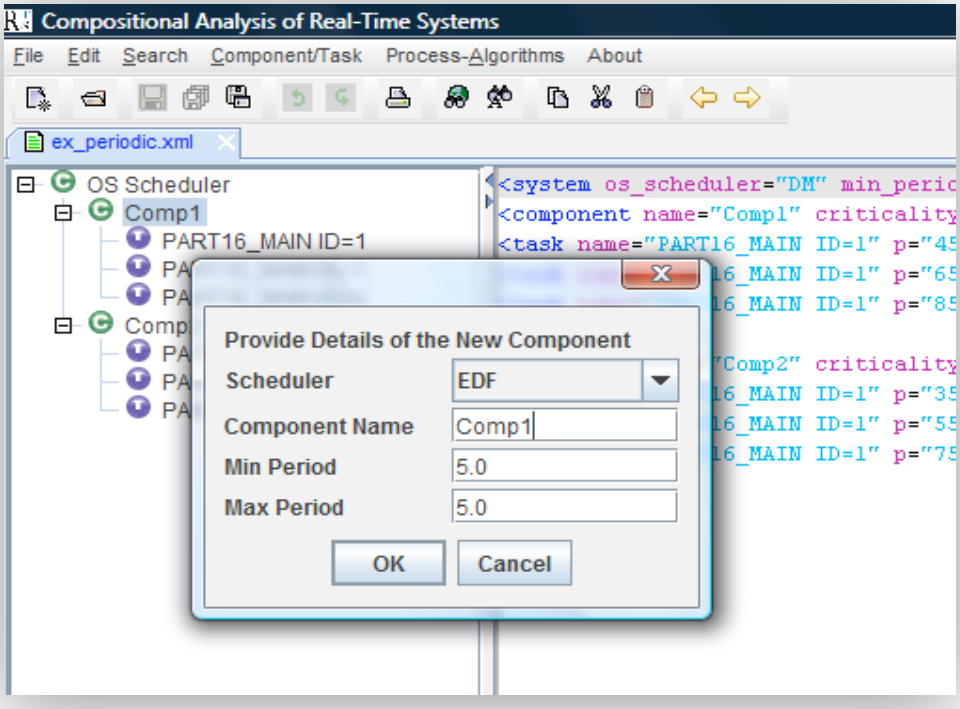


6. Update An Existing Component

Step 1

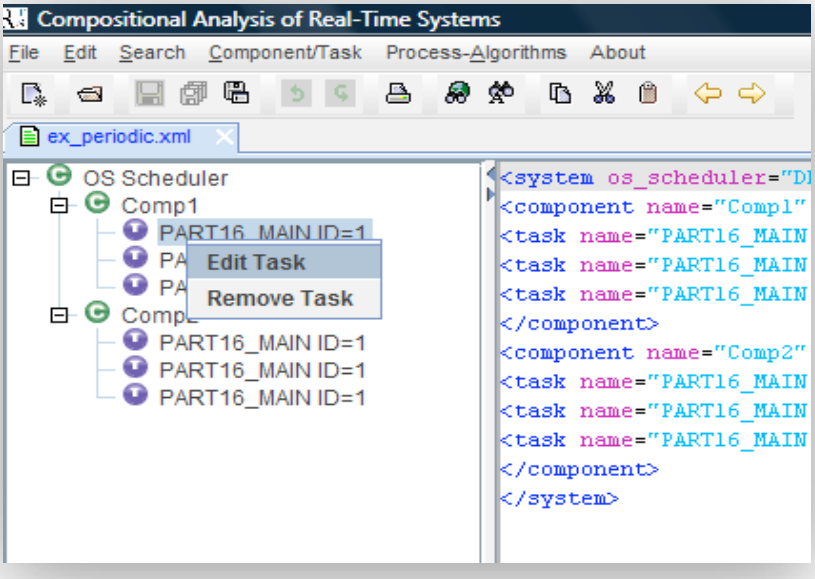


Step 2

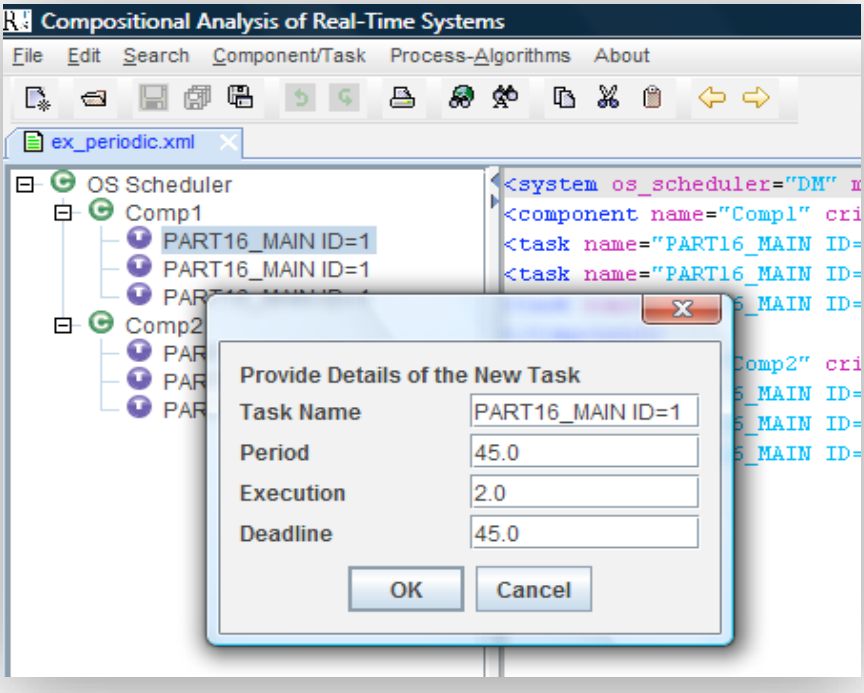


7. Update An Existing Task

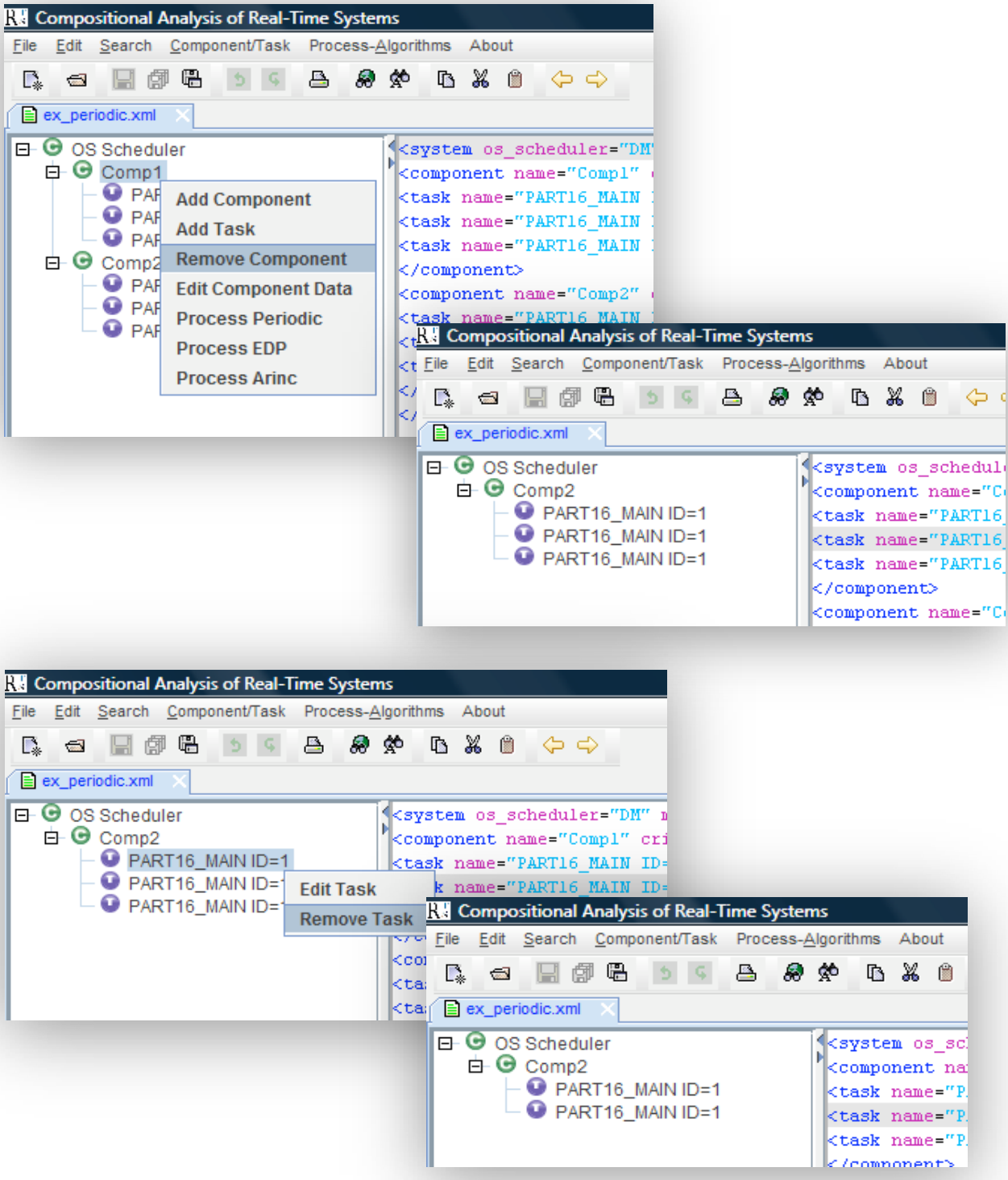
Step 1



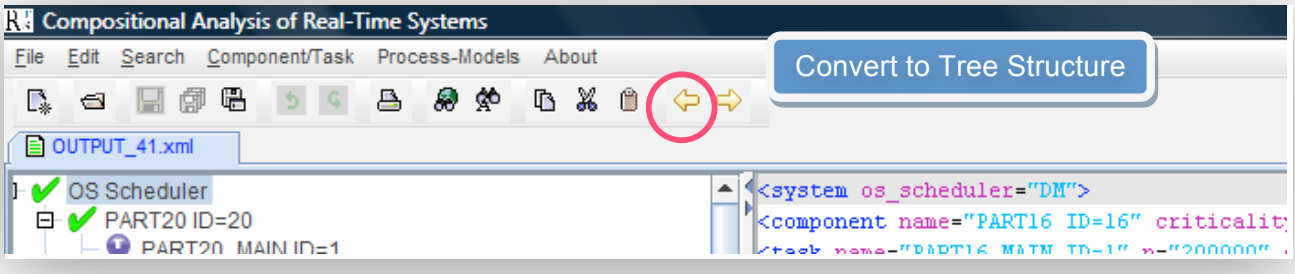
Step 2

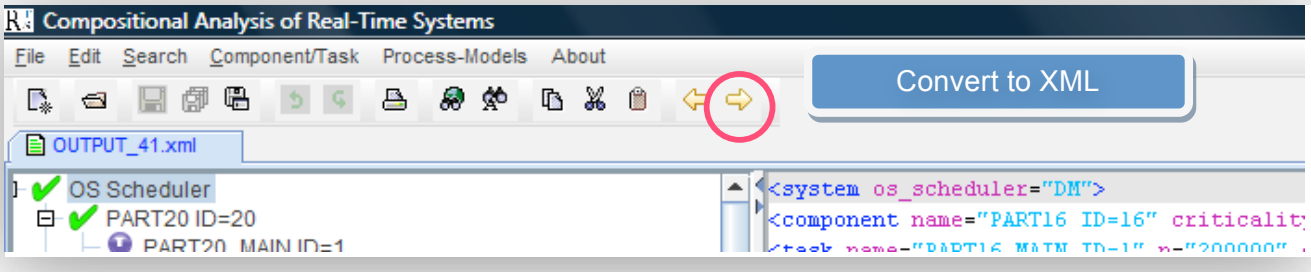


8. Remove An Existing Component/Task

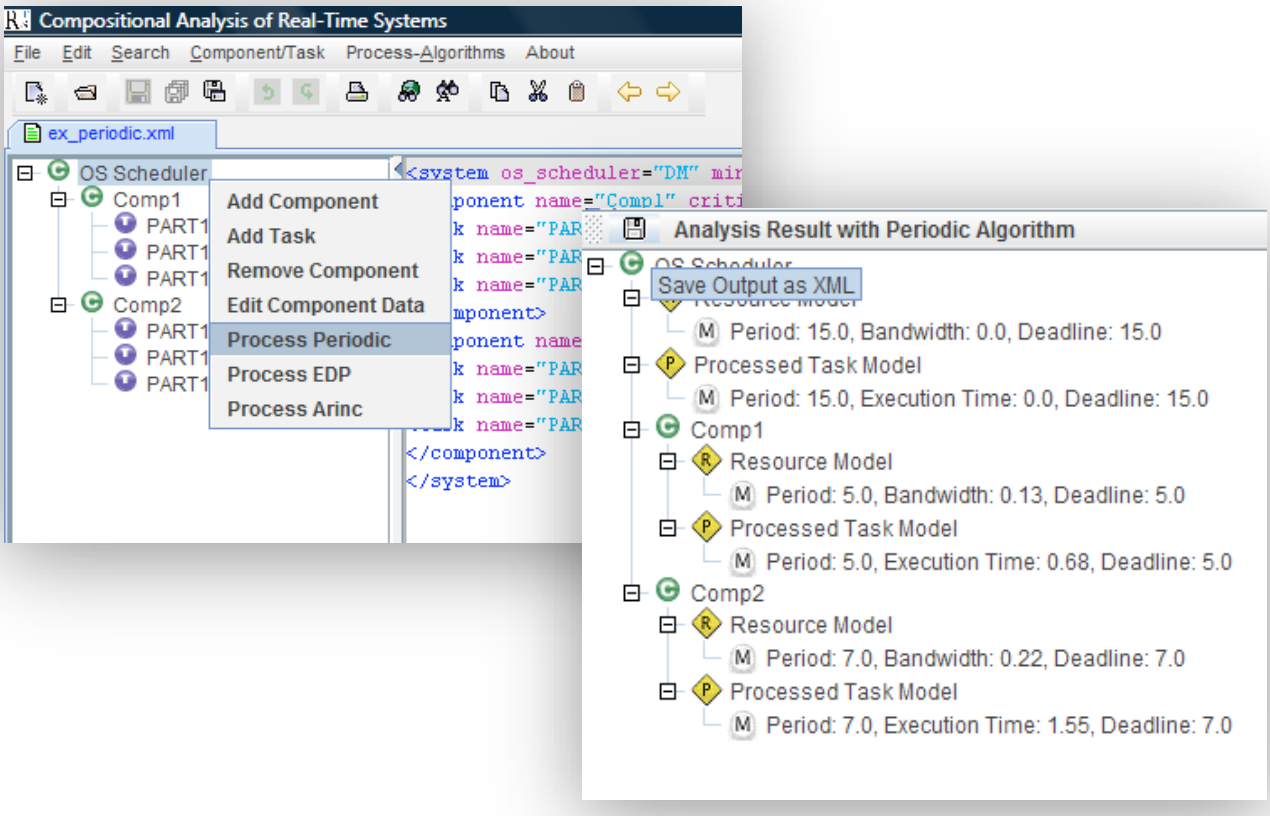


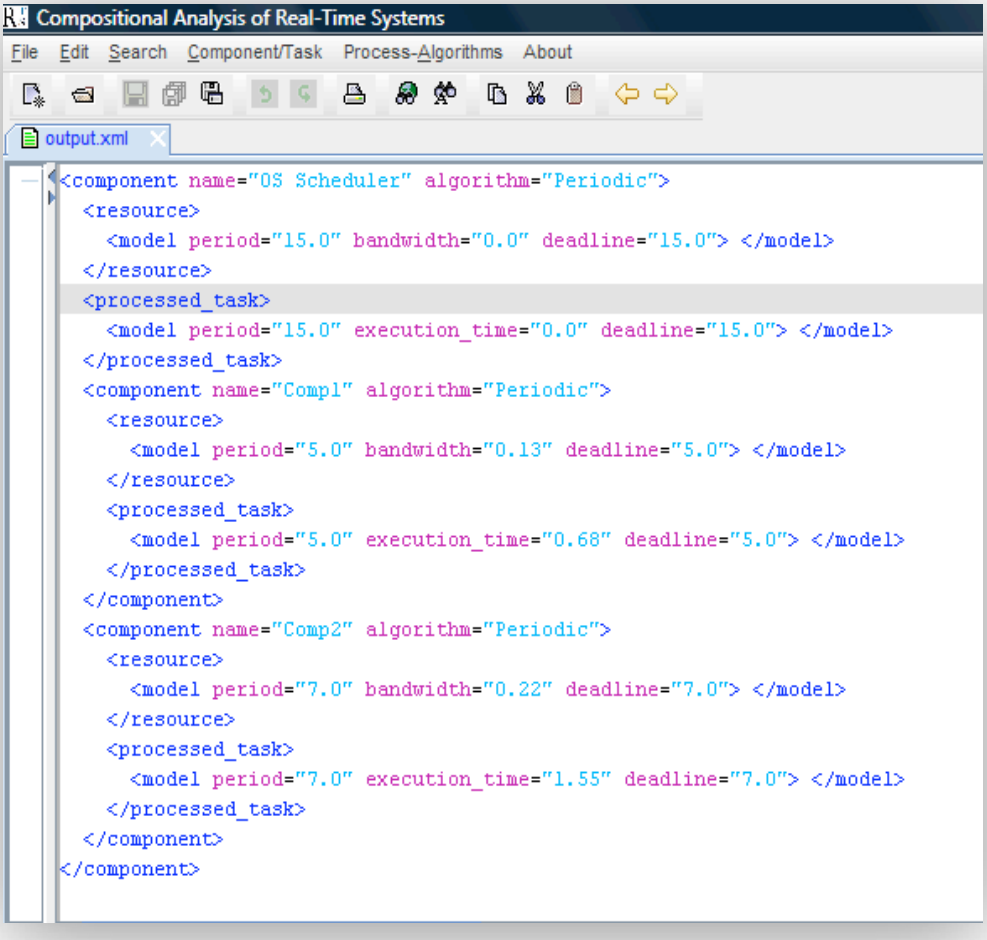
9. Convert between XML and Tree View



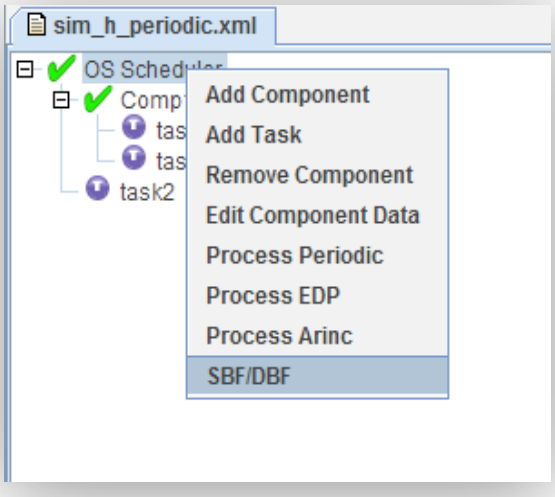


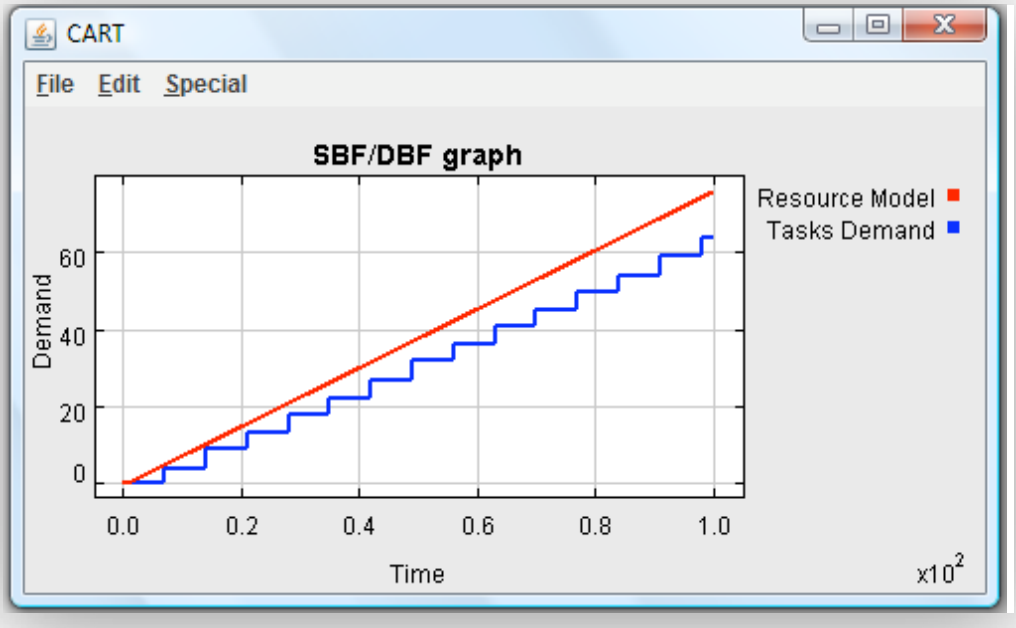
10. Apply Processing Models





11. View Demand/Supply Bound (DBF/SBF) Functions





In the SBF/DBF graph, the SBF of the resource model is represented by a red line. This function captures the resource supplied by the interface at a given time. For example, the amount of resource units supplied by the interface at any time interval of length $0.4^* 10^2$ is approximately 30.

The DBF is represented by a blue line, which represents the amount of resource needed by the tasks in the component for them to be schedulable. In the above figure, the number of resource units required in any interval of length $0.4^* 10^2$ in order to feasibly schedule the tasks in the selected component is approximately 25.
In this example, the DBF locates below the SBF, therefore the tasks in the component are schedulable under the computed interface.

PART II

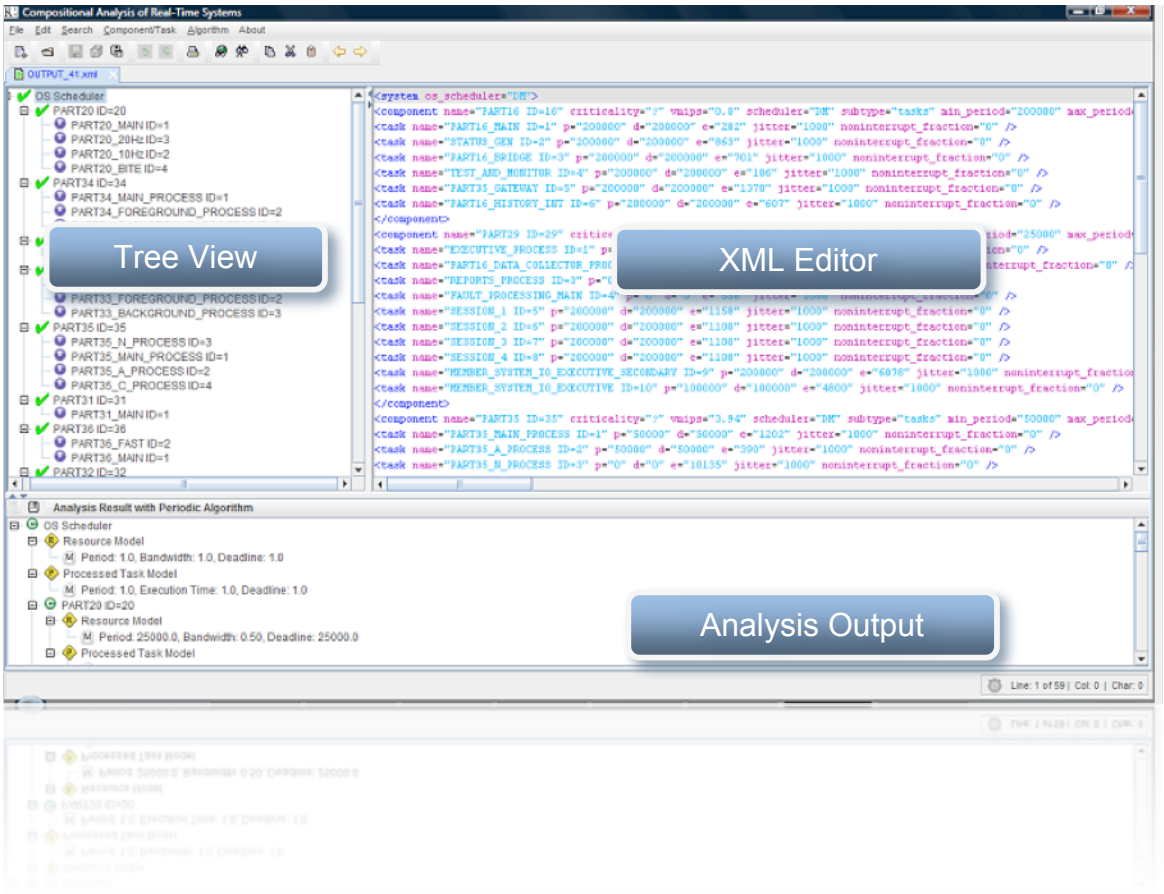
CARTS Design & Implementation

1. Design of CARTS

1.1 Open Source Editor

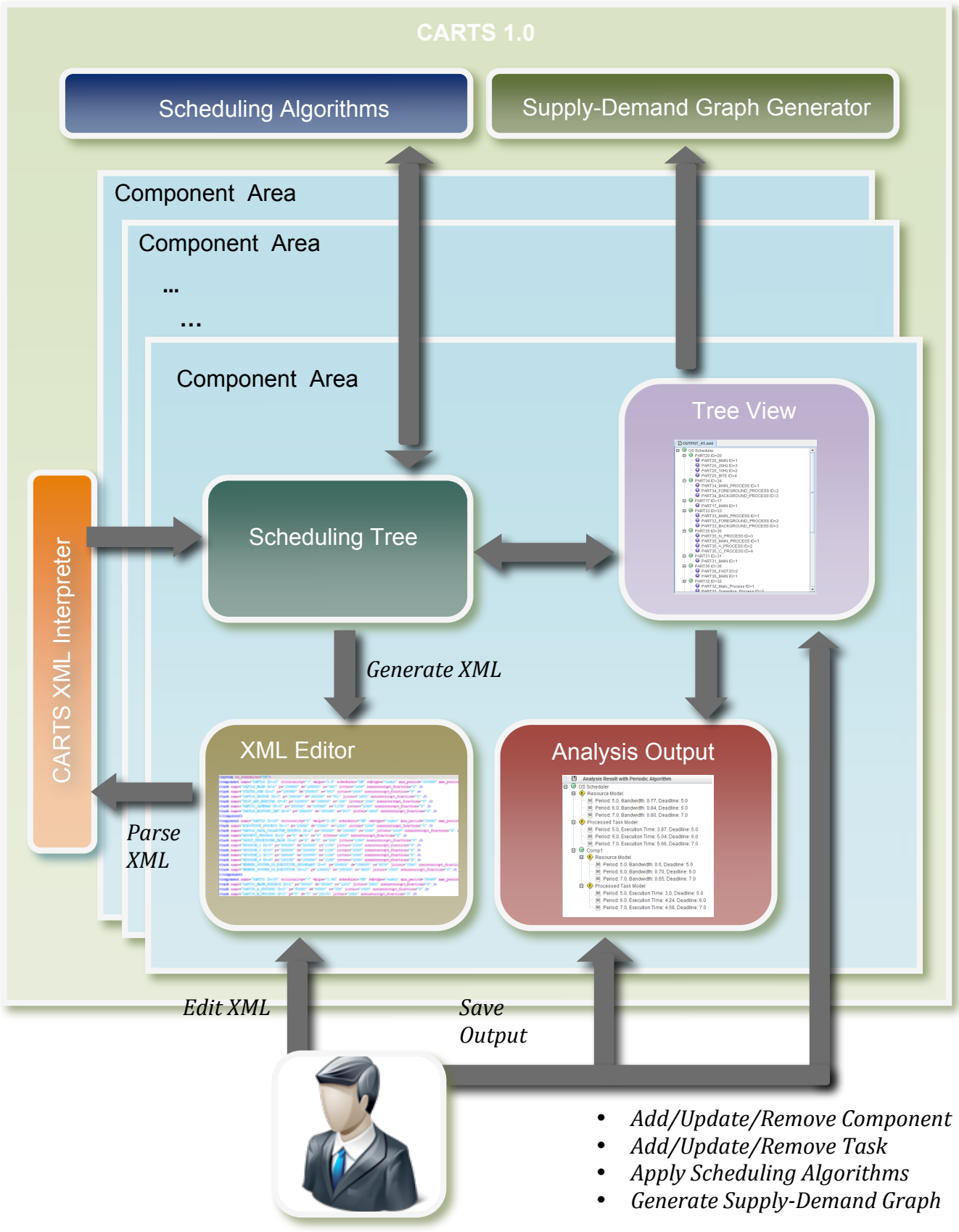
An XML editor is needed to support the editor functionalities for the XML file. Instead of writing a new one from scratch, we are using the Open Source Editor *JPE* (*http://www.jpedit.co.uk)* written in Java. *JPE* is a simple editor supporting the basic functions with semantic support for XML.

1.2 CARTS Graphical User Interface



1.3 Architecture

The JPE Editor is used as a framework for CARTS. The *XML Editor* component of UI is completely managed by the *JPE* software itself. The *Tree View* component of the UI has been added by to display the system as a tree structure. The following diagram depicts the components and their interaction.



- *Add/Update/Remove Component*
- *Add/Update/Remove Task*
- *Apply Scheduling Algorithms*
- *Generate Supply-Demand Graph*

2. Implementation of CARTS

2.1 Classes

The following provides a short description of the major classes in CARTS.

2.1.1 SchedulingComponent

SchedulingComponent class represents the component in the system. It contains the name given to the component, child components and tasks. The class also contains fields which contain the maximum and minimum period values required, algorithm which the process models apply on the component.

This also supports methods to apply Periodic, EDP and ARINC models on the *SchedulingComponent* and generate values. It also supports writing the component field values to a given XML file.

2.1.2 **Task**

Task class represents the task that is present under a *SchedulingComponent*. The class consists of fields for task's period, deadline, and execution time. It also supports writing the task field values to a given XML file.

2.1.3 SchedulingTree

SchedulingTree class contains reference to the root component of the entire system. It provides methods to write the entire system to an XML file.

2.1.4 XMLInterpreter

XMLInterpreter class contains methods to interpret the given XML file and create SchedulingComponents and Tasks, and eventually building a *SchedulingTree*.

2.1.5 ComponentUI

ComponentUI class represents the UI for *SchedulingTree*. It contains a reference to the *SchedulingTree* it is representing. It contains a *JTree* object which is a shadow tree of the referred *SchedulingTree*. This *JTree* object results in a Tree View of the *SchedulingTree*.

It provides the user with options to add/update/remove component/task. It also provides methods to apply process models on any SchedulingComponent.

2.1.6 Output

Output class provides methods to display the values generated by applying process models as a tree and also write them in an XML file.

2.1.7 ComponentArea

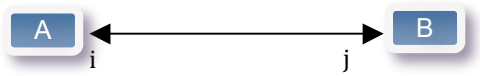
ComponentArea class is a combination of *SchedulingTree*, XML Editor, *ComponentUI* and Output objects. A *ComponentArea* object can be created from scratch or using an existing CARTS XML file. When created from an existing one, the *XMLInterpreter* is used to build the *SchedulingTree* object. The *SchedulingTree* is given to the *ComponentUI* to build the Tree View of it.

This class also provides methods to convert an edited XML to a Tree View and vice-versa.

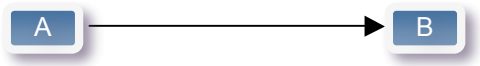
2.1.8 JPE

The JPE class contains a collection of ComponentArea objects. It contains a JTabbedPane where each tab contains a ComponentArea object. The JPE class also provides handlers for the UI menus and buttons.

2.2 Class Diagram

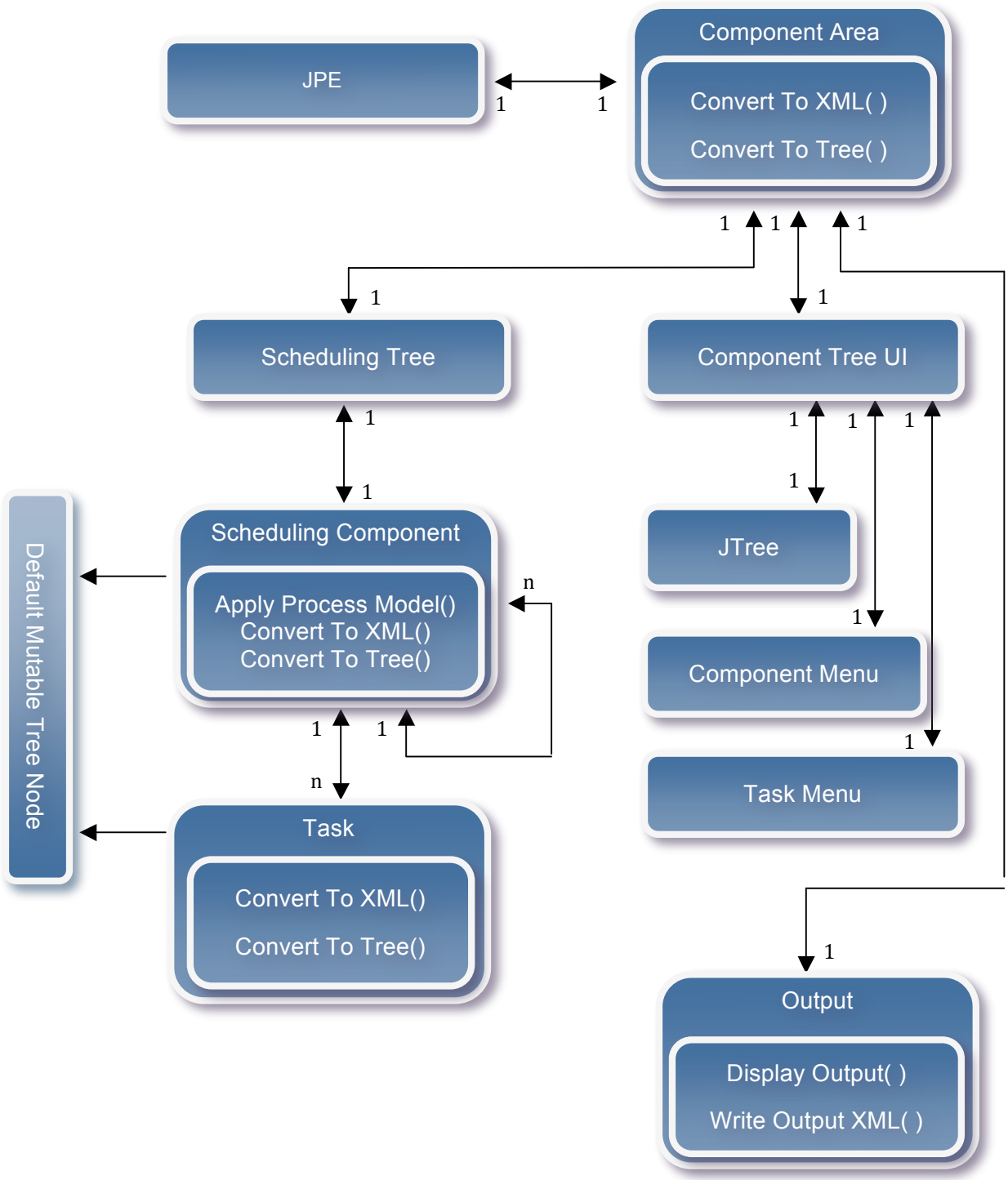


i instances Class A contain *j* instances of Class B



Class A extends Class B

The following captures a high level view of the main class diagram implemented in CARTS 1.0.


JPE
Component Area
Convert To XML( )
Convert To Tree( )
Scheduling Tree
Component Tree UI
Scheduling Component
Apply Process Model()
Convert To XML()
Convert To Tree()
Default Mutable Tree Node
JTree
Component Menu
Task Menu
Task
Convert To XML()
Convert To Tree()
Output
Display Output( )
Write Output XML( )

2.3 Scheduling Algorithms

The CARTS tool presently implements three analysis techniques for generating component interfaces; namely, Periodic resource model based approach [3] (Periodic.java, EDFSchedulability periodic.java, DMSchedulability periodic.java), EDP resource model based approach [1] and ARINC-653 (avionics real-time operating system standard) specific approach [2] (ARINC.java). This section does not present details of any of these techniques, but only focuses on how these techniques have been implemented in the CARTS tool. It is expected that the reader of this document is familiar with the aforementioned scientific publications.

Each approach implements three main functions; **abstractionProcedure**, **generateInterface** and **transformInterface**. Since these functions are similar in each of the three approaches, we will only focus on the Periodic resource model based approach in this documentation.

2.3.1 abstractionProcedure

Input: A *SchedulingComponent* object *c*.

Output: None.

For each child of *c* that is also a *SchedulingComponent* object, the function iteratively calls itself using the child object as input. When all these iterative calls return, interfaces have been generated for all the *SchedulingComponent* objects which are present in the subtree rooted at c. In particular, for all the
children of c that are *SchedulingComponent* objects, their **ProcessedTaskList** data structure contains the interface tasks. That is, it contains the list of tasks generated from Periodic resource models (one for each period value in the range [**MinimumPeriod**, **MaximumPeriod**]) that will be presented to c.

The tasks in the workload of c comprises of all the tasks in the **TaskList** data structure as well as specific tasks from the **ProcessedTaskList** data structure of its children. If **MinimumPeriod** = **MaximumPeriod** for each *SchedulingComponent* object, then each **ProcessedTaskList** data structure contains only one task and that will be in the workload of c. On the other hand, if **MinimumPeriod** is not equal to **MaximumPeriod**, then the tool enforces the condition that **MinimumPeriod** and **MaximumPeriod** be identical for each *SchedulingComponent* object in the entire system. In this case, we match the interface period of c with the period of the tasks from the **ProcessedTaskList** data structure of its children. That is, for each period value i in the range [**MinimumPeriod**, **MaximumPeriod**], the workload of c comprises of all the tasks in the **TaskList** data structure and one task with period i from the **ProcessedTaskList** data structure of each of its children.

Function **abstractionProcedure** first invokes function **generateInterface** and then invokes function **transformInterface** for each period value i. **generateInterface**

returns a periodic resource model with period i, which is then transformed into a periodic task with period i by function **transformInterface**. The resource model is stored in the **ResourceModelList** data structure and the transformed task is stored in the **ProcessedTaskList** data structure of object c. Further, if c has a parent *SchedulingComponent* object, then the tasks in **ProcessedTaskList** are also copied in the **ChildrenToTaskTable** data structure of the parent. Finally, the **Processed** boolean variable of c is set to true so that appropriate visual cues can be given in the GUI indicating that c has been processed.

2.3.2 generateInterface and transformInterface

2.3.2.1 generateInterface

Input:
1. Period value i for periodic resource model.
2. A *SchedulingComponent* object c.

Output: A *ResourceModel* object characterizing a periodic resource model with period i. This resource model is guaranteed to satisfy schedulability conditions for c as specified in this scientific publication [3].

Depending on the scheduling algorithm employed by object c (edf or dm), function **generateInterface** invokes function **getBandwidth** of either class *EDFSchedulability_periodic* or class *DMSchedulability_periodic*. Function **getBandwidth** takes as input c and i, and generates as output the bandwidth required from a periodic resource model with period i to schedule object c. In other words, **getBandwidth** implements the schedulability tests from [3] (Theorem 1 by *EDFSchedulability_periodic* and Theorem 2 by *DMSchedulability_periodic*). This resource model bandwidth is then used to generate the appropriate periodic resource model using one of the constructors of class **ResourceModel**.

2.3.2.2 transformInterface

Input:
1. A *ResourceModel* object r characterizing a periodic resource model.
2. Period value i for the periodic task that will be generated.
3. A *SchedulingComponent* object c.

Output: A Task object characterizing a periodic task with period i. This task is obtained from resource model r using techniques specified in [3].

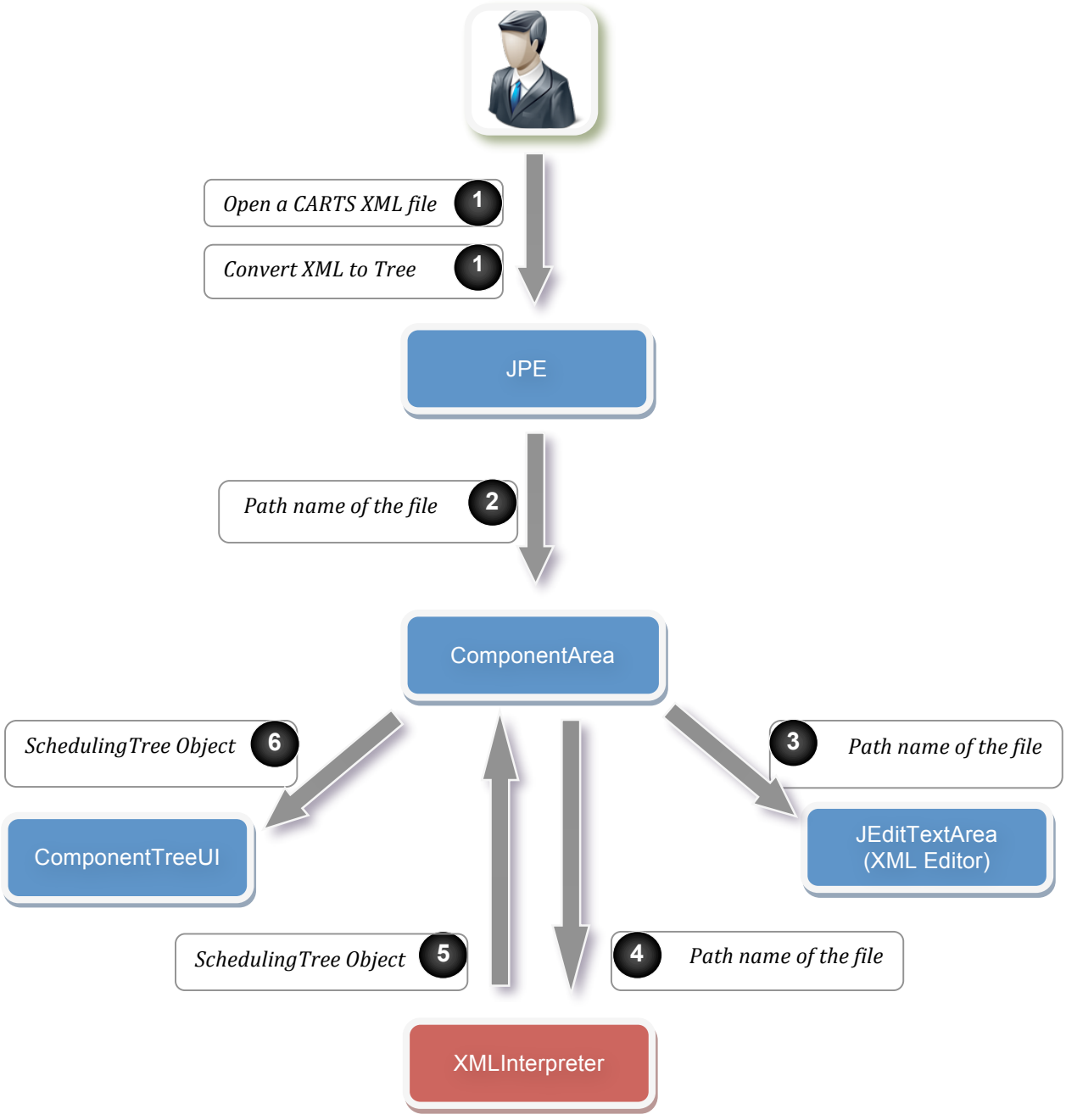
Depending on the scheduling algorithm employed by object c (edf or dm), function **transformInterface** invokes either function **T_EDF** of class *EDF-Schedulability-periodic* or function **T_DM** of class *DMSchedulability_periodic*. These functions take as input c and i, and generate as output a periodic task with period i such that the amount of resource required by the task is at least as much as the amount of

resource required by r. In other words, these functions implement the transformation from resource model to tasks specified in [3].

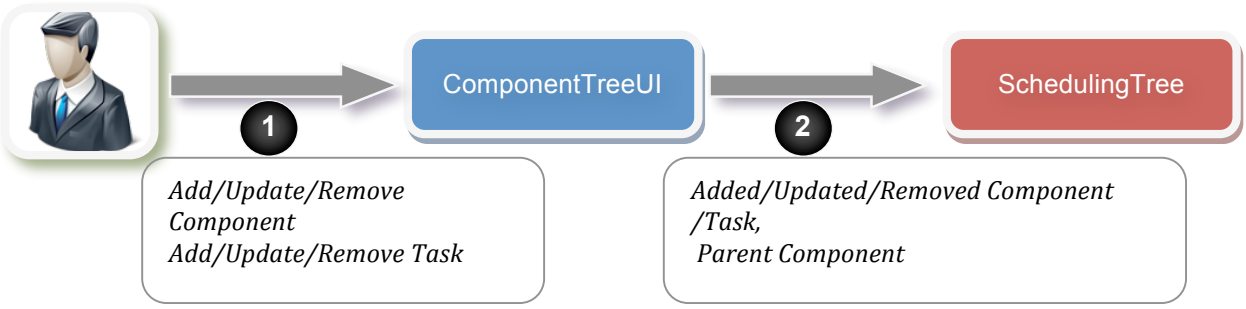
2.4 Data Flow

This section gives a general idea of flow of data among the objects of the classes explained.

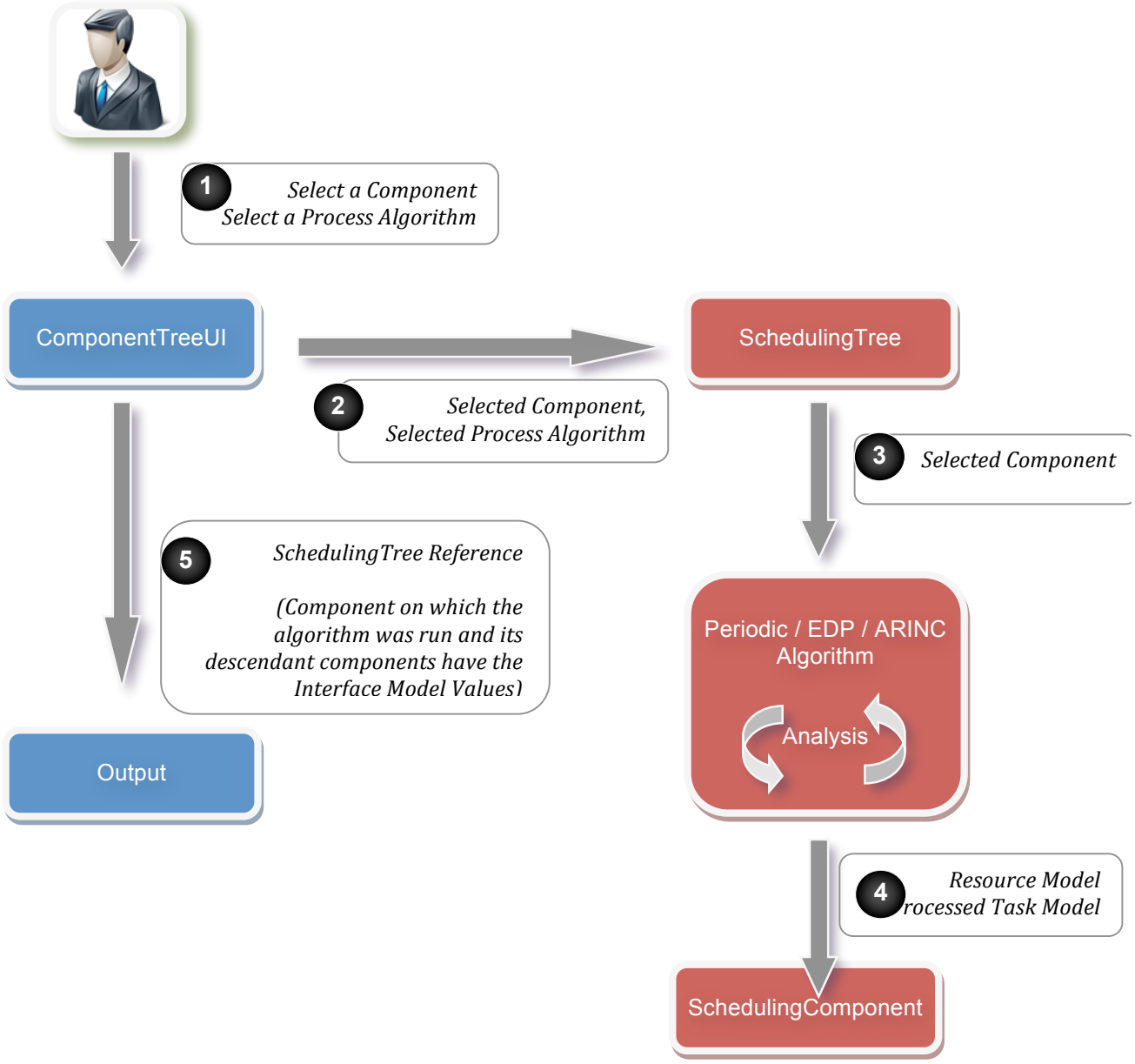
2.4.1 CARTS XML to Tree View



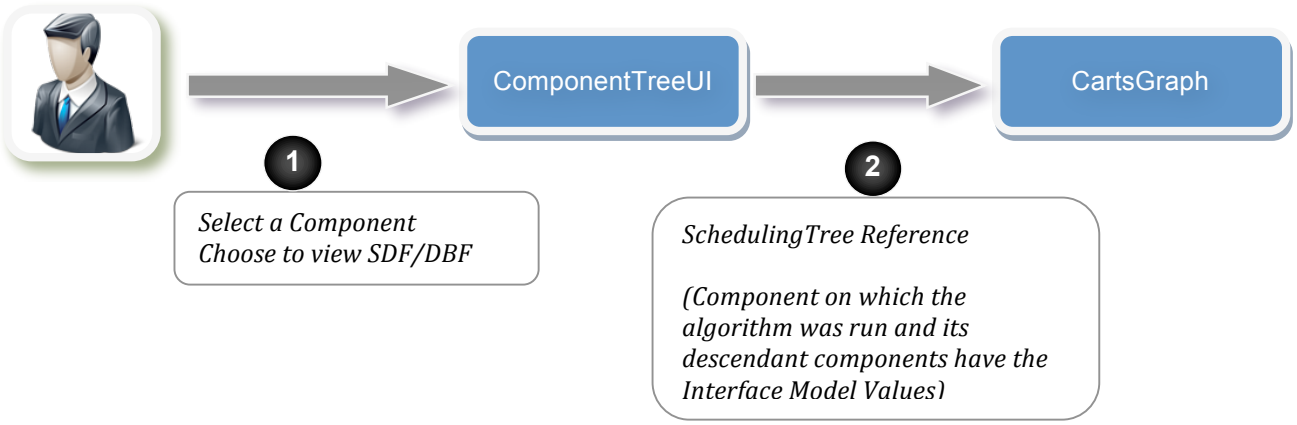
2.4.2 Add/ Update/ Remove operations on Tree View



2.4.3 Apply Algorithms through Tree View



2.4.4 View SDF/DBF Graphs



2.5 Source Code

The source code has been divided into 3 main groups. Packages whose names start with -

- **com** – contains the open source code from JPE. Only the features which are desired in CARTS 2.0 have been retained.
  - **com.jpe** – The JPE class has been modified to accommodate the Scheduling Tree GUI (Tree View) in JPE Editor.

- **edu**
  - **edu.penn.rtg.schedulingapp** – basic Scheduling Tree data structure implementation.
  - **edu.penn.rtg.schedulingapp.algo –** scheduling algorithm implementation.
  - **edu.penn.rtg.schedulingapp.input.treeGUI –** GUI for scheduling Tree.
  - **edu.penn.rtg.schedulingapp.input.treeGUI.dialog –** source code for GUI dialog on add/update/edit Scheduling Components and Tasks.
  - **edu.penn.rtg.schedulingapp.xml –** XML interpreter to parse CARTS XML file.
  - **edu.penn.rtg.schedulingapp.output –** source code on GUI rendering of analysis output.
  - **edu.penn.rtg.schedulingapp.output.graph –** source code on plotting graph for analysis output.

- **ptolemy** – contains source code on graph which is used to render the analysis results from Scheduling Tree.

PART III

Extension to CARTS

The following gives a list of changes one has to do in source code whenever a new field/algorithm is added to CARTS.

1. Add New Fields

1.1 Add new fields to *Scheduling Component*

Basic Changes

1. Add the new field in *SchedulingComponent* class.
2. Add the new field as an argument to the constructor taking values for other fields. Initialize the new field in the constructor.
3. Add set and get functions to the class if needed.
4. Add the field to *updateComponent* method and update the field with the new value.
5. Add the new field in writing the *SchedulingComponent* in the XML file.

GUI

1. Add the new field to replyFromAddCompDialog function and use it to make the new SchedulingComponent object
2. Add the new field into AddComponentDialog.
3. In the handler for OK, read the value of the field and pass the field to replyFromAddCompDialog function.
4. Add the new field to replyFromEditCompDialog function. Add the field as an argument to updateComponent method.

1.2 Add new fields to *Task*

Basic Changes

1. Add the new field in Task class.
2. Add the new field as an argument to the constructor taking values for other fields. Initialize the new field in the constructor.
3. Add set and get functions to the class if needed.
4. Add the field to updateTask method and update the field with the new value.
5. Add the new field in writing the Task in the XML file.

GUI

1. Add the new field to replyFromAddTaskDialog function and use it to make the new Task object.
2. Add the new field into AddTaskDialog box
3. In the handler for the dialog OK, read the value of the field and pass the field to replyFromAddTaskDialog function.
4. Add the new field to replyFromEditTaskDialog function. Add the field as an argument to updateTask method.

2. Add New Algorithms

Basic Changes

1. Add a new class for the new algorithm to **edu.penn.rtg.schedulingapp.algo** package.
2. Add a process function in SchedulingTree class.

CLI

1. Add the new algorithm as an option that can be given as command line argument.

GUI

1. Add the new algorithm as an option in the popup menu in createPopupComponentMenu.
2. Add a new handler for the algorithm in ComponentUI
3. Add changes to ComponentMenu handler to call the handler when the user chooses the new algorithm in the menu.

Output

Presently the analysis output is rendered as a Tree in GUI. The output can also be saved as an XML. The present tree output can be replaced with any other renderer, provided the new renderer implements *displayOutput* method which is defined in the *OutputI* interface file.

If any new field is to be added to the output, they will be read automatically, provided they are given in the present Processes Task/Resource Model output structure. The field will also be written into the XML file.

REFERENCES

[1] Arvind Easwaran, Madhukar Anand, and Insup Lee. Compositional analysis framework using EDP resource models. In Proceedings of IEEE Real-Time Systems Symposium, pages 129-138, 2007.

[2] Arvind Easwaran, Insup Lee, Oleg Sokolsky, and Steve Vestal. A compositional scheduling framework for digital avionics systems. In Proceedings of IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2009.

[3] Insik Shin and Insup Lee. Periodic resource model for compositional realtime guarantees. In Proceedings of IEEE Real-Time Systems Symposium, pages 2-13, 2003.

CONTACT INFORMATION

For more information on CARTS, email to: carts@seas.upenn.edu

CARTS

http://rtg.cis.upenn.edu/carts