procedure is sensitive to a task and resource allocation for the execution of the task involved in the project.

The procedure is a single pass method, which employs S-SGS and one priority rule in order to obtain one feasible schedule. Multi-priority rule method is employed by using SGS several times and each time a different priority rule is used. To obtain the optimal schedule, SGS is employed several times and so the optimal solution is obtained. It is assumed that the priority rule is defined on the activities with respect to precedence constraints and resources, which gives tasks scheduling. The project activities considered here include sequential and branch-merge patterns with synchronization time rule. The duration of all the tasks and their resource requirements are received and a task is put on the schedule if its precedence and resource constraints are met. After the initial dummy task is put on the schedule the decision set is identified and the decision set is the collection of tasks that are as yet unscheduled and are precedence and resource feasible. The priority value (utility value) for each of the tasks is calculated and the task that has the best priority value is selected to be put on the schedule. A resource is allocated to the tasks which are having the same utility value (priority) by using resource allocation game. The key mechanism is to increase the resources availability for tasks that are on critical path by releasing resources earlier before the completion of the task.

To develop the procedure, two or more of renewable resources are used and furthermore, the set of all tasks is partitioned into disjoint subsets, and all tasks forming one subset have to be performed by the same resource. The procedure for resource allocation is given in Figure 6.1.
<table>
<thead>
<tr>
<th>Step 1:</th>
<th>Set of tasks, set of indivisible resources, the resource usage time period, resource require / release time points, precedence relations and task durations are received.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 2:</td>
<td>Task dependencies are formed by using the precedence relations.</td>
</tr>
<tr>
<td>Step 3:</td>
<td>Resource dependencies are formed by using the functions <code>require( )</code> and <code>release( )</code>.</td>
</tr>
<tr>
<td>Step 4:</td>
<td>The tasks are grouped into different categories based on the resource dependency using <code>depend_on_for( )</code>. That is, the set of tasks which are using the resource $R_i$ are brought into a single group.</td>
</tr>
<tr>
<td>Step 5:</td>
<td>In case of sequential tasks to be executed, the tasks are allowed to overlap if the relationship between the tasks is start-start. The tasks which are to be executed together (branch-merge pattern) are identified in each group using precedence relations and task dependencies.</td>
</tr>
</tbody>
</table>
| Step 6: | The time period and the time points of resource require / release are used for the tasks which are to be executed in branch pattern.  
Case (a): When the time points for resource require / release are known.  
The tasks are allowed to overlap if the time point of resource require ($t_{req}$) and the time point of resource release ($t_{rel}$) are satisfied and if $t_{req} \geq t_{rel}$.  
Then using the S-SGS, the schedule is generated.  
Case (b): When the time points for resource require / release are not known and the resource usage time period is known. The utility value for each task is calculated based on the time period over a resource. There arise two cases.  
(i) Different utility values:  
When the tasks are having different utility values, a resource is allocated to a task based on a priority rule and a feasible schedule is generated. Next, the resource is allocated to tasks based on another priority rule and then a feasible schedule is generated. Repeatedly producing different SGSs, the optimal schedule is obtained.  
(ii) Same utility values:  
When the tasks are having the same utility values, the resource allocation game is used for resource allocation to break the tie and based on the results of the game the SGS is employed and the optimal schedule is obtained. |

**Figure 6.1 An overall procedure for resource allocation**
The above procedure is used to identify the resource and the precedence dependencies between tasks and also it is used to allocate the resources to tasks in the branch-merge patterns by using priority rules described in chapter 5. Based on above procedure, the following algorithms have been developed.

1. Task dependency identification
2. Resource dependency identification
3. Task overlapping
4. Resource allocation with priority rules
5. Overlapped project makespan

6.2 IDENTIFICATION OF TASK DEPENDENCY

Precedence constraints are binary conditions and state that the tasks $T_i$ and $T_j$ have to be executed in the given order. As defined already, in the project network, Petri net representations are used between activities in order to identify the relationship between activities more clearly. A finish-start precedence constraint between $T_i$ and $T_j$ requires that end of $T_i < \text{start of } T_j$, while a start-start precedence constraint only describes start of $T_i \leq \text{start of } T_j$. These will be denoted by $(T_i < T_j)$ and $(T_i \leq T_j)$ respectively, and determine a directed acyclic graph over the set of the tasks together. Note that although the description of RCPSP contains only finish-start precedence constraints, an additional start-start, start-finish and finish-finish precedence constraints are identified during the solution process. If a task $T_j$ is having start-start relationship with its predecessor $T_i$, then $T_i$ is allowed to start first followed by $T_j$ and therefore, these two tasks are having precedence relations suitable for overlapping. An algorithm to find the task dependency is given in Figure 6.2.
Step 1: Identify the number of predecessors and successors for each task.

Step 2: If a task is having a predecessor, then set predecessor = true else set predecessor = false.

Step 3: If a task is having a successor, then set successor = true else set successor = false.

Step 4: If the predecessor of a task is false, then identify the task as start task.

Step 5: If the successor of a task is false, then identify the task as end task.

Step 6: If the predecessor for a task is true and the number of predecessor = 1 and the successor of the task is true and the number of successor = 1 then the given task is identified as sequential task.

Step 7: If the number of successors of a task is ≥ 2, then identify branch and if the number of predecessors of the task ≥ 2, then identify merging.

Step 8: Identify the set of all tasks which are having the same predecessor and successor as branch-merge tasks.

Figure 6.2 Algorithm for identifying task dependency

Branch-merge pattern

In the simple case, set of all tasks which are having unique predecessor (i.e. predecessors of set of all the tasks are same) and unique successor (i.e. successors of set of all the tasks are same) is identified as branch-merge pattern. If a task which is having two or more successors, then the task is identified as branching task and if a task is having two or more predecessors, then the task is identified as merging task and in this way, simple branch-merge pattern is identified. If a task is having two or more successors, and if it is one of the successors of a task, then a branch within a branch is identified. Conversely, if a task is having two or more predecessors and if it is one of the predecessors of a task, then a merge within a merge is identified and in this way a branch-merge pattern within a branch-merge pattern is identified. Therefore, based on the above algorithm, sequential and branch-merge patterns are identified in a given project.
Step1: Identify tasks to overlap and current resource transfer among them.

Step2: Overlap the tasks based on resource require/release time points and resource usage time periods.

Step3: For sequential tasks to be overlapped, overlap the tasks using resource dependency and precedence relations if the relation type between the two tasks is start-start.

Step4: For the tasks in branch-merge, overlap the tasks within the branch-merge by using resource dependency.

Step5: If upstream task releases the resources earlier perform an overlap else if downstream task requires resources after starting then perform an overlap.
else if upstream task releases the resource earlier and downstream task requires resources after starting, then perform an overlap.

| Figure 6.4 Procedure for task overlapping |

When the resource require/release time points are known instead of resource usage time periods, the overlapping between two tasks are employed in two ways as given in Figure 6.4 for task overlapping. The first way, the overlapping is performed in sequential pattern of tasks based on the relationship between the two tasks. Relationship between two tasks such as start-start, start-finish, finish-start and finish-finish are used and overlapping is allowed between two tasks when the relation type is start-start. Otherwise, no overlap is allowed though the resource is released before completion or required after starting. In the second way, the overlapping is performed in the branch-merge patterns. Since the tasks in this pattern are executed serially one by one (because of resource constraint), then overlapping is employed between tasks in this pattern by not considering the relationship between the tasks (obviously, the relationship among the tasks allows overlapping). Therefore, an overlapping is performed depending upon the resource release and resource require time points.
When the resource usage time period is known instead of resource require / release time points, then the overlapping is performed in two ways. In case of sequential tasks, three types of overlapping are performed. If the resource usage time period of the upstream activity is at the end and if the resource usage time period of downstream activity is at the beginning no overlapping is performed. If the resource usage time period of the upstream activity is at the end and if the resource usage time period of the downstream activity is also at the end then an overlap is performed and it is named as *predictive overlap*. If the resource usage time period of the upstream activity is at the beginning and if the resource usage time period of the downstream activity is at the beginning, then an overlap is performed and it is called as *early overlap*. If the resource usage time period of the upstream activity is at the beginning and if the resource usage time period of the downstream activity is at the end, an overlap is performed and it is called as *partial overlap*.

6.4.1 Modes of task overlapping

The task overlapping is performed for the tasks with resource require / release time points and resource usage time periods. When the RrTP and RiTP are known, the following overlapping is established. Figure 6.5 summarises some of the findings and the left part illustrates *early overlap* strategy in which downstream task relies on resource release from upstream task. *Early overlapping* is allowed when the upstream task releases the resource at the time point before completing the task completion. The overlapping period of *early overlap* is equal to the resource release time point of the upstream task and the percentage of overlap is calculated as $\lambda = \frac{t_{rel}}{d_i + d_j}$ where $t_{rel}$ and $d_i$ are the resource release time point and duration of the upstream task respectively and $d_j$ is the duration of the downstream task. The middle part illustrates
predictive overlap strategy in which downstream task relies on early starting before releasing the resource. Predictive overlapping is allowed when the downstream task require the resource at the time point after the starting and by anticipating the resource. The overlapping period of predictive overlap is equal to the resource require time point of the downstream task and the percentage of overlap is calculated as

$$\lambda = \frac{t_{\text{req}}}{d_i + d_j}$$

where $t_{\text{req}}$ and $d_j$ are the resource require time part and duration of the downstream task respectively and $d_i$ is the duration of the upstream task. The right part summarises a partial overlap strategy in which upstream task relies on resource release and downstream task relies on late resource require. Partial overlapping is allowed when the upstream task releases the resource before completion and the downstream task requires the resource after starting. The overlapping period of partial overlap is equal to be the sum of resource require time point of the downstream task and the resource release time point of the upstream task and the percentage of overlap is calculated as

$$\lambda = \frac{(t_{\text{req}} + t_{\text{rel}})}{(d_i + d_j)}$$

where $t_{\text{req}}$ and $d_j$ are the resource require time point and duration of the downstream task and $t_{\text{rel}}$ and $d_i$ are the resource release time point and duration of the upstream task.

![Figure 6.5 Modes of task overlapping](image-url)
The time periods of resource usage for all the tasks in the branch-merge pattern are less than or equal to the durations of the corresponding tasks. When the resource usage time periods are known, then overlapping is allowed based on the unequal resource usage time periods. There arises three ways of resources usages such as at start, at middle and at the end of the tasks. In the first one, all the tasks in the branch-merge pattern need the resource at the beginning and the resource is released before the completion of tasks and so early overlap is performed. In the second one, all the tasks in the branch-merge need the resource at the middle. That is, if a task releases the resource before completion and another task requires the resource after starting and so, partial overlap is performed. In the third one, all the tasks in the branch-merge pattern need the resource at the end. That is, if a task needs the resource at the end, then by anticipating the resource, it is allowed to start its execution and predictive overlap is performed. The overlap amount of task is determined by both precedence and resource constraints with other tasks in the critical path.

6.4.2 Overlapping level identification

Overlapping level can be achieved based on the high or low scheme introduced which helps to identify the types of overlapping between tasks. In the resource require tasks, the following two cases are introduced for task overlapping.

Low: Resource is required to begin the task.

High: Resource is required to end the task.

In the resource release tasks, the following two cases are introduced for task overlapping.

Low: Resource is released at the end of the task.

High: Resource is released before the end of the task.
Table 6.1 Overlapping level identification

<table>
<thead>
<tr>
<th>Resource release</th>
<th>Resource require</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Low (to begin)</td>
<td>High (to end)</td>
</tr>
<tr>
<td>Low (at end)</td>
<td>Sequential</td>
<td>Anticipation</td>
</tr>
<tr>
<td></td>
<td>No overlap</td>
<td>overlap</td>
</tr>
<tr>
<td>High (before end)</td>
<td>Early overlap</td>
<td>Partial overlap</td>
</tr>
</tbody>
</table>

Table 6.1 shows the four extreme situations likely to arise when the upstream resource release is high or low and when the downstream resource require is high or low. The activities are to be overlapped differently for each combination of resource require and resource release and summarised as below.

- When the upstream resource is low and the downstream resource require is low, the overlapping is sequential (no overlap); downstream activity begins the resource release from the upstream activity at the end.
- When the upstream resource release is high and the downstream resource require is low, the overlapping is early and the downstream activity starts with prerelease of resource from upstream activity.
- When the upstream resource release is low and the downstream resource require is high, the overlapping is predictive, and the upstream activity releases the resource at the end and downstream activity to start early by anticipating the resource.
- When the upstream resource release is high and the downstream resource require is high, the overlapping is partial and downstream activity starts before resource release from upstream activity and upstream activity release the resource before completion.
From a schedule perspective, there exist four types of dependency before the pair of tasks as discussed earlier. Among the four types of dependency, the start-start relation allows overlapping and this relation is used to overlap the sequential tasks in the project. To have overlapping in branch-merge pattern, the next activity to be scheduled is selected based on the resource allocated.

6.5 RESOURCE ALLOCATION

In scheduling, a resource is allowed to have transition from unavailable to available (and vice versa) instantaneously. A resource r that is assigned to an activity $T_i$ is unavailable to all activities other than $T_i$ from precisely the start of $T_i$ to its end and it is immediately assigned to another activity $T_j$ at any time point before or after the execution of $T_i$. Hence, a simple accounting of resource usage over time is sufficient to determine the set of instances $\{I\}$ during which resource $r$ is available for assignment to some unscheduled activity. Resource is allocated to the tasks based on the precedence relations and the resource is allowed to flow from the upstream activities to downstream activities. In case of more than one resource, all the resources are allowed to flow independently from their corresponding upstream activities to downstream activities and the set of all tasks using a particular distinct. Resources are available for release at an earlier time from the upstream task and they are used by the downstream tasks. Otherwise, resources are required at a later time for the downstream task and this task is allowed to start earlier before the resource release from the upstream task. A set of active tasks which have started to work in the current schedule is identified. For each task in this schedule, its overlapped part of the task has been performed in prior schedule(s).
In branch-merge pattern of tasks, the precedence relations allow the tasks to execute in parallel. Since all the parallel tasks in this pattern are using the same resource, they are allowed to execute sequentially. In this case, a resource is allowed to flow in all the activities in this pattern based on the priority rules defined in Chapter 5. An algorithm is presented in Figure 6.6 to allocate the resources in branch-merge pattern. To allocate the resources to tasks, utility value of each task is calculated.

| Step 1: | For the tasks in the sequential pattern, allocate the resources to tasks sequentially one by one based on resource require/release time points. |
| Step 2: | For the tasks in the branch-merge pattern, calculate the utility value of each task. |
| Step 3: | Allocate a resource to a task based on priority rules by using the utility value. |
| Step 4: | Order the tasks based on RrTP priority rule and allocate the resource accordingly. If two tasks are having the same RrTP, then go to step 8. |
| Step 5: | Order the tasks based on RjTP priority rule and allocate the resource accordingly. If two tasks are having the same RjTP, then go to step 8. |
| Step 6: | Order the tasks based on LRTP priority rule and allocate the resource accordingly. If two tasks are having the same LRTP, then go to step 8. |
| Step 7: | Order the tasks based on SRTP priority rule and allocate the resource accordingly. If two tasks are having the same SRTP, then go to step 8. |
| Step 8: | Allocate the resource to a task based on the results received from the resource allocation game. |

Figure 6.6 Algorithm for resource allocation

6.5.1 Utility value

The resource utility value for a task is calculated based on the following. When RrTP is known, the resource usage time period is calculated as \( RU_i = d_i - t_{req} \) where \( d_i \) is the duration of the task \( T_i \) and \( t_{req} \) is the time point of resource required. The total duration of resource usage is calculated as \( RU = \sum (d_i - t_{req}) \). When RjTP is known, the resource usage time period is calculated as \( RU_i = d_i - t_{rel} \) where \( d_i \) is the duration of the task \( T_i \) and \( t_{rel} \) is the time point of resource release. The total duration of
resource usage is calculated as \( RU = \sum (d_i - t_{i\text{req}}) \). When the \( R_rTP \) and \( R_iTP \) are known for each task \( T_i \), the value of resource usage time period is calculated as \( RU_i = d_i - (t_{i\text{req}} + t_{i\text{rel}}) \) where \( d_i \) is the duration of the task \( T_i \) and \( t_{i\text{req}} \) and \( t_{i\text{rel}} \) are the time points of resource require and resource release respectively for task \( T_i \). The total duration of the resource usage is calculated as \( RU = \Sigma (d_i - (t_{i\text{req}} + t_{i\text{rel}})) \). Then, the resource utility value of a task is calculated as \( U_i = RU_i / RU \) where \( i = 1, ..., n \). When the resource usage time period is known, then calculate the total resource usage time period of all tasks as \( RU = \Sigma t_i \), where \( t_i \) is the resource usage time period of the task \( T_i \). Then the resource utility value for the task \( T_i \) is calculated as \( U_i = t_i / RU \) where \( i = 1, ..., n \). Utility value for each task is calculated using the above said formulae and the calculated utility value is used to allocate the resource to a task in the branch-merge pattern.

6.5.2 Resource allocation using \( R_rTP \) and \( R_iTP \) priority rules

The priority value for each agent over a resource is calculated in such a way that it should minimise the finish time of the tasks. Also, it is found that the number of coordinated relationships with the tasks and the start time of the task should both be considered when computing the priority of a task. The reason for considering the coordinated relationships is apparent: a task that requires the resource at the beginning has higher priority. To summarise, the resource is allocated to the task which is the earliest task from the set with the highest priority.

When the \( R_rTP \) priority rule is selected for the resource allocation, the next task to be scheduled from the branch tasks is selected based on the \( R_rTP \) and the resource is allocated first to a task which requires the resource at the beginning. Then,
the resource is allocated to the task which requires the resource at the time point after starting. In case of two tasks which are having the same RₚTPs, then the resource is allocated to the task which is having shorter task duration. If task durations and RₚTPs are same for both the tasks, then the resource is allocated based on the result of the resource allocation game.

When RᵢTP priority rule is selected for the resource allocation, the next task to be scheduled from the branch tasks is selected based on the RᵢTP and the resource is allocated first to a task which releases the resource before completion. Then, the resource is allocated to the task which releases the resource at the end. In case of two tasks which are having the same RᵢTPs, then the resource is allocated to the task which is having longer task duration. If task durations and RᵢTPs are same for both tasks, then the resource is allocated based on the result of the resource allocation game.

6.5.3 Resource allocation using LRTP and SRTP priority rules

When the LRTP priority rule is selected for the resource allocation, then the next task to be scheduled is selected from the branch tasks based on LRTP rule and the resource is allocated to a task which is having the highest LRTP. Then, the resource is allocated to a task which is having next highest LRTP. In this way the resource is allocated to tasks from the highest LRTP to the lowest LRTP. In case of tie (two tasks are having the same LRTPs), then the resource is allocated to a task which is having shorter task duration. If task durations and LRTPs are same for both the tasks, then the resource is allocated based on the result of the resource allocation game.
When the SRTP priority rule is selected for resource allocation, then the next task to be scheduled in the branch tasks is selected based on SRTP rule and the resource is allocated to a task which is having the shortest SRTP. Then, the resource is allocated to a task which is having next shortest SRTP. In this way the resource is allocated to tasks from the shortest SRTP to the highest SRTP. In case of tie (two tasks are having the same SRTPs), then the resource is allocated to a task which is having longer task duration. If task durations and SRTPs are same for both tasks, then the resource is allocated based on the result of the game.

6.5.4 Resource allocation using game

Formally, the coordination problem is seen as the problem of selecting one out of many Nash equilibria in the resource allocation game. There are several ways to solve the game in which time constraints on the possible actions choices of the agents. It is noted that the agents have the ability to identify one another, a simple convention is created using the following three rules: The set of agents is ordered, the set of actions of each agent is ordered, and these orderings are common knowledge among agents using some predefined ways. The choice for an optimal joint action proceeds as follows. The first agent in the agent ordering chooses an optimal action (that corresponds to Nash equilibrium) that appears first in its action ordering. The next agent then chooses its first optimal action in its action ordering given the first agent’s choice. This procedure continues until all agents have chosen their actions. This general, domain independent method will always result in an optimal joint action and moreover it is easy to implement. During execution the agents do not have to explicitly coordinate their actions via negotiation.
When the utility values and the durations of two tasks in the branch-merge patterns are same, then the resource allocation game is employed to allocate a single resource to two activities. Although it happens rarely in practice if there is still a tie (exactly the same priority and equal resource require time period), it can be broken by using the resource allocation game. Since a single resource is used in each branch-merge pattern, then the ordering of (sequencing) activities in the branch-merge pattern is made based on the resource allocation game.

6.5.5 Overlapped project makespan

The total project duration is calculated based on the critical path method and the overlapped project duration is calculated based on the overlapped time period. The algorithm to compute the overlapped project duration is given in Figure 6.7.

<table>
<thead>
<tr>
<th>Step1:</th>
<th>Initialise the durations $d_i$ of tasks using different resources.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step2:</td>
<td>Calculate the duration of each schedule from starting to ending task.</td>
</tr>
<tr>
<td>Step3:</td>
<td>Identify the set of active tasks in the current state satisfying precedence and resource constraints based on the priority rules.</td>
</tr>
<tr>
<td>Step4:</td>
<td>Adjust the duration of the active tasks and the lead time.</td>
</tr>
<tr>
<td>Step5:</td>
<td>Find the minimum project makespan after the possible overlapping.</td>
</tr>
<tr>
<td>Step6:</td>
<td>Find the project makespan based on optimum priority rule or based on the selected priority rule.</td>
</tr>
</tbody>
</table>

**Figure 6.7 Algorithm to compute overlapped project makespan.**

Initially, the durations of the critical path ($d_{cpm}$) and the overlapped path ($d_{ovp}$) are set to zero. If an overlap is not performed, add the durations $d_i$ and $d_j$ of the tasks $T_i$ and $T_j$ in the critical path ($i < j$) with $d_{cpm}$ and $d_{ovp}$. If an overlap is performed, then add the durations $d_i$ and $d_j$ with $d_{cpm}$. Then add any one of the following values with $d_{ovp}$. When the resource require and release time points are known the overlap is an
early overlap then resource release time point of the task $T_i$ is subtracted from $d_i$ and
the value $(d_i - t_{rel}) + d_j$ is added with $d_{ovp}$. If the overlap is predictive overlap, then the
resource require time point of the task $T_j$ is subtracted from $d_j$ and the value $(d_j - t_{req}) +
d_i$ is added with $d_{ovp}$. Otherwise, the overlap is a partial overlap. Then, the resource
release time point of $T_i$ is subtracted from $d_i$ and the resource require time point of $T_j$
is subtracted from $d_j$ and the value $(d_i - t_{rel}) + (d_j - t_{req})$ is added with $d_{ovp}$. When the
resource usage time periods of the tasks are known and when the overlap is an early
overlap, the value $(d_i - t_i) + d_j$ is added with $d_{ovp}$. If the overlap is predictive overlap,
then the value $(d_j - t_j) + d_i$ is added with $d_{ovp}$. Otherwise, the overlap is a partial
overlap and then the value $(d_i + d_j) - (t_i + t_j)$ is added with $d_{ovp}$ to find the overlapped
project makespan.

6.6 PSEUDO CODE FOR PROJECT SCHEDULING

A project with sequential and branch-merge patterns of activities is scheduled
by using the pseudo code given in this section. The pseudo codes task_dependence()
(Figure 6.9) and resource_dependence() (Figure 6.10) are used to identify the task and
resource dependencies respectively. The pseudo code time_period_schedule( )
(Figure 6.11) is used to schedule the activities in the project by overlapping the
activities based on resource require/release, resource usage time periods and priority
rules.

```plaintext
Procedure Driver ()
{
    Input: Set of all functions required to complete the branch - merge patterns
    Output: Resultant identification of task and resource dependencies.
    1. invoke Procedure Task_Dependence()
    3. invoke Procedure Resource_Dependence()
    4. invoke Time_Period_Schedule()
    5. exit( ) on completion
}
```

Figure 6.8 Pseudo code for project scheduling
**Procedure Task_dependence()**

```
{  
    **Input:** Tasks required from T₁ to Tₙ  
    **Output:** Identify the tasks with dependencies
1. Get the tasks required Tᵢ for i = 1 to n.
2. Assign the specific tasks with their relationships with its predecessors as SS, SF, FF, and FS respectively depending on the nature of the tasks and its requirements for further processing.
3. For i=1 to n do
   If rel of Ti with its predecessor = SS then Tᵢ_overlap = TRUE
   Else Tᵢ_overlap = FALSE where Tᵢ_overlap is a boolean flag set to indicate the overlap or non overlap status
4. Use linked list and store the following values:
   Tᵢ_s -> the successor of Tᵢ
   Tᵢ_p -> the predecessor of Tᵢ
5. If set of Tᵢ_s are same and set of Tᵢ_p are same then b_m_pattern ++;
   Display Tᵢ as branch and merge satisfier.
   /* Displaying branch and merge task */
6. Otherwise, Tᵢ++
7. Repeat the step from 4 to 7 until the tasks are finished
8. display b_m_pattern value that counts the number of tasks that satisfy the branch_merge pattern.
9. Otherwise, display the sequential patterns.
}
```

**Figure 6.9 Pseudo code for identification of task dependency**

Dependency relationships between two tasks such as sequential and branch-merge patterns are identified using the pseudo code given in the Figure 6.9. If the relationship between a task and its predecessor is Start-Start, then the overlap flag is set and overlapping is performed between these two tasks during schedule. If the set of all successor tasks and the set of all predecessor tasks are same for a set of tasks, then a simple branch-merge pattern is identified.
Procedure Resource_Dependence()
{
Input: Previous function, Set of all Tasks Ti, Resources ri.
Output: Resource dependent tasks
1. Get the set of tasks Ti and resources ri for i, j=1 to n and j<i often;
2. if (require(Ti, ri)) /* Boolean Value*/ then set the require = TRUE
   else set i++, j++
   /* require (Ti, ri) is a function that assigns the specific resource require to
   specific task */
3. repeat through step 2 for i, j = 1 to n.
4. if (release(Ti, ri) /* Boolean Value*/) then set the release = TRUE
5. else set i++, j++
   /* release (Ti, ri) is a function that assigns the specific resource release to
   specific tasks */
6. repeat through step 4 for i = 1 to n.
7. if (require(Ti, ri) ^ release(Ti, ri) then depend_on_for = TRUE
8. else j++, i++ depend_on_for = FALSE
9. display those tasks and resources that satisfy the condition.
10. repeat through step 10 until the resource and tasks are exhausted.
}

Figure 6.10 Pseudo code for identification of resource dependency

Resource dependencies between two tasks such as require relation and release
relation are identified using the pseudo code given in the Figure 6.10. If the
resource relationships such as require and release between a task and its
predecessor are TRUE, then set depend_on_for = TRUE and a resource
dependency is formed between these two tasks during schedule.
Procedure Time_Period_Schedule()
{
    Input : Previous Functions, Tasks, Resources, Resource_require time list, Resource_release time list. Resource usage time period list.
    Output: Identify the tasks that are to be executed based on the branch_merge_condition.
    1. Get resources $r_j$, tasks $T_i$, $treq_i$, $trel_i$ for each task.
    2. if (require($T_i$, $r_j$, $treq_i$) /* Boolean value*/
    3. then set resource_require = TRUE
    4. else $i++, j++$
       /* Require($T_i$, $r_j$, $treq_i$) is a function that has a task that is assigned the specific resource and the resource is allocated based on the priority rule. */
    5. repeat through step 2 until $i >=$ number of tasks.
    6. if (release($T_i$, $r_j$, $treq_i$) /* Boolean value*/
    7. then resource_release= TRUE
    8. else $i++, j++$
       /* Release($T_i$, $r_j$, $trel_i$) is a function that has a task that releases to the specific resource */
    9. repeat through step 6 until $i >=$ number of tasks.
    10. if require($T_i$, $r_j$, $treq_i$) and release ($T_i$, $r_j$, $trel_i$) \& $treq_i >= trel_i$ then
        Constraint ($T_{i+1}$, $T_i$, $r_j$, $treq_i$, $trel_i$) = TRUE
    11. Constraint ++; Display the specific task.
    12. else $i++, j++$
       /* assign resources to tasks based on the constraint */
    13. repeat through step 10 until any of the input are given.
    14. Display the number of tasks with the specified constraint ie, constraint++.
    15. select $r_j$ where resource_release = TRUE and display them.
    16. repeat through step 15 for $i = 1$ to $n$.
    17. Calculate the utility value for each task based on resource usage time period and allocate the resources to tasks based on optimum priority rule.
}

Figure 6.11 Pseudo code for project scheduling

Resource constraints are formed based on the dependencies between two tasks such as resource_require and resource_release using the pseudo code given in the Figure 6.11. Then, resource allocation for project scheduling based on priority rules are performed.
6.7 DISCUSSION

An overall scheduling procedure and five algorithms have been presented which would help the project scheduling to decide when and how to overlap activities. Activities need not occur in a predetermined sequence and can be re-sequenced or even overlapped and activities are not always points in time and indeed release or require resources at distributed times. All the activities of the project are carefully planned in advance and the presented algorithms replace the central computing, each endowed with a local view of its execution and the ability to respond to the project. The overall system performance is not globally planned, but emerges through the resource and precedence information of agents during scheduling. These algorithms are based on distributed control and the system is based on the viewpoint of project management.

Task dependency identification algorithm is used to find the pattern of tasks in the project such as sequential and branch-merge and further, the algorithms are used to sequence the activities in the project. With the help of resource dependency and resource allocation algorithms, sequencing of activities in the project is easily identified. The scheduling algorithm schedules the activities in the project based on resource and task dependencies. Overlapping algorithm is used to overlap the two activities in three possible ways such as early overlap, predictive overlap and partial overlap. Finally, the overlapped project makespan algorithm helps to find the project durations under critical path and overlapped path. To realise the above algorithms, a software prototype has been designed and developed.