

UNIVERSITEIT • STELLENBOSCH • UNIVERSITY

RESOURCE CONSTRAINED STEP SCHEDULING OF PROJECT TASKS

Anton Burger Eygelaar



Thesis presented in partial fulfilment
of the requirements for the degree of
Master of Science in Civil Engineering
at the University of Stellenbosch.

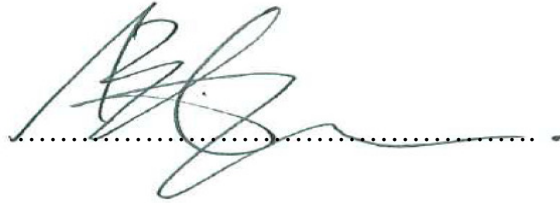
Supervisor: Dr. G.C. van Rooyen

March 2008

DECLARATION

I, the undersigned, hereby declare that the work contained in this report is my own original work and that I have not previously in its entirety or in part submitted it at any university for a degree.

Signature:

A handwritten signature in black ink, appearing to be 'A. J. ...', written over a horizontal dotted line.

Date:

The date '12/1/2008' handwritten in black ink, positioned above a horizontal dotted line.

SYNOPSIS

The logical scheduling of activities in an engineering project currently relies heavily on the experience and intuition of the persons responsible for the schedule. In large projects the complexity of the schedule far exceeds the capacity of human intuition, and systematic techniques are required to compute a consistent sequence of activities. In this study a simple model of the engineering process is described. Based on certain specified relationships between components of the model, a consistent sequence of activities is determined in the form of a logical step schedule. The problem of resource constraints receives special attention. Engineering projects are often executed with limited resources and determining the impact of such restrictions on the logical step schedule is important. This study investigates activity-shifting strategies to find a near-optimal sequence of activities that guarantees consistent evolution of deliverables while resolving resource conflicts within the context of logical step schedules.

SAMEVATTING

Die logiese skedulering van aktiwiteite in 'n ingenieursprojek steun swaar op die ondervinding en intuïsie van die persone wat verantwoordelik is vir die skedule. In groot projekte is die kompleksiteit van die skedule veel hoër as die kapasiteit van die menslike intuïsie, en sistematiese tegnieke word benodig om 'n konsekwente volgorde van aktiwiteite te bereken. In hierdie studie word 'n eenvoudige model van die ingenieursproses beskryf. Gebaseer op sommige relasies tussen komponente van die model, kan 'n konsekwente volgorde van aktiwiteite bepaal word in die vorm van 'n logiese stap-skedule. Die probleem van beperkte hulpbronne ontvang spesiale aandag. Ingenieursprojekte word dikwels uitgevoer met beperkte hulpbronne en dit is belangrik om die impak daarvan op die logiese stap-skedule te bepaal. Die studie ondersoek die gebruik van aktiwiteit-skuiwende strategieë om 'n naby-optimale volgorde van aktiwiteite te vind wat konsekwente ontwikkeling van die projekprodukte waarborg, terwyl hulpbron konflikte opgelos word binne die konteks van 'n logiese stap-skedule.

ACKNOWLEDGEMENTS

Jan and Lynnette Eygelaar, my father and mother. You provided guidance, support, and unconditional love. You will always be in my heart.

Dr. Gert van Rooyen, my supervisor. It is difficult to express my gratitude. You provided excellent guidance and introduced to me the world of engineering informatics. I am forever grateful to you.

Synne Koht Esnali, for your love and support.

Bertie Olivier, for convincing me that engineering informatics is fun.

Deon Heydenrych, my best friend. Need I say more?

Lars Henrik Mikalsen, for your input and smart ideas.

Bjørn Brunstad, for believing in me.

CONTENTS

DECLARATION.....	I
SYNOPSIS	II
SAMEVATTING.....	II
ACKNOWLEDGEMENTS	III
CONTENTS.....	IV
LIST OF FIGURES.....	VI
LIST OF TABLES.....	VII
1 BACKGROUND	1
2 STATE OF THE ART.....	2
3 PROBLEM STATEMENT	4
4 PLEP: PLANNING THE ENGINEERING PROCESS	8
4.1 SPECIFIED COMPONENTS: THE BASIC INGREDIENTS	8
4.1.1 TASK.....	8
4.1.2 DELIVERABLE	8
4.1.3 PERSON	9
4.1.4 TOOL.....	9
4.1.5 STATUSES.....	9
4.1.6 DELIVERABLE EVOLUTION PROFILE	9
4.2 SPECIFIED RELATIONS: CONNECTING THE COMPONENTS	10
4.2.1 TASK-DELIVERABLE RELATION	10
4.2.2 TASK-PERSON RELATION.....	12
4.2.3 DELIVERABLE-TOOL RELATION	13
4.3 MATHEMATICALLY DERIVED RELATIONS	14
4.3.1 RELATION IN THE SET OF TASKS: Task dependencies.....	15
4.3.2 PREDEFINED RULES: Enforcing stage-wise evolution of Deliverables.....	15
5 LOGICAL STEP SCHEDULE: TOPOLOGICAL SORTING OF THE RELATION IN THE SET OF TASKS	20
5.1 DETECTION AND REMOVAL OF CYCLES.....	20
5.1.1 LOOPS.....	21
5.1.2 MULTIPLE EDGES	22
5.2 TOPOLOGICAL SORTING	22
5.3 CHARACTERISTICS OF THE LOGICAL STEP SCHEDULE:	24
6 RESOURCE CONSTRAINTS.....	25
6.1 EXTENDING THE PROCESS MODEL	26
6.1.1 RESOURCE AVAILABILITY.....	26
6.1.2 RESOURCE LOADING AND CONFLICTS.....	26
7 RESOLVING RESOURCE CONFLICTS	29
7.1 TASK-SHUFFLING: LOCAL INTRA-STEP TASK SHIFTING	29
7.1.1 DESCRIPTION OF TASK-SHUFFLING OPERATION.....	30
7.2 TABU-SEARCH: GLOBAL TASK SHIFTING.....	32
7.2.1 TABU SEARCH: PATTERN DEFINITION	32
7.2.2 TABU-SEARCH: APPLYING THE PATTERN TO THE PROBLEM DOMAIN.....	33
7.2.3 FLOW OF THE TABU SEARCH ALGORITHM.....	36
8 IMPLEMENTATION	38
8.1 GRAPHICAL USER INTERFACE	38
8.1.1 MAIN VIEW OF APPLICATION.....	38

8.2	GRAPH LIBRARY TOOLKIT	40
8.2.1	GRAPHS.....	40
8.2.2	ALGORITHMS	43
8.3	MODEL TOOLKIT.....	47
8.3.1	CLASSES OF THE MODEL TOOLKIT.....	48
8.3.2	ADDING, REMOVING AND FINDING COMPONENTS AND RELATIONSHIPS.....	50
8.3.3	CALCULATING TASK DURATION.....	52
8.3.4	CALCULATING AND REMOVING THE <i>u</i> CST.....	53
8.4	RESOURCE CONSTRAINT TOOLKIT	54
8.4.1	LOGICAL STEP SCHEDULE	54
8.4.2	TASK-SHUFFLING ALGORITHM	57
8.4.3	TABU-SEARCH.....	57
8.5	PERSISTENCE.....	58
9	RESULTS	60
9.1	SMALL AEC PROJECT: FICTICIOUS PROJECT.....	60
9.2	LARGE AEC PROJECT: INDUSTRY PARTNER	62
9.3	INTERPRETATION OF RESULTS	64
9.3.1	LOGICAL STEP SCHEDULES.....	65
9.3.2	DURATIONS	65
9.3.3	RESOURCE CONFLICTS.....	66
9.3.4	NUMBER OF MOVES.....	67
9.3.5	AVERAGE TIME PER MOVE.....	67
9.3.6	AVERAGE TIME PER <i>c</i> CLSS	67
9.3.7	EFFECTIVE SEARCHING OF SOLUTION SPACE BY TABU SEACH.....	68
9.3.8	USER INPUT.....	68
10	CONCLUSIONS AND RECOMMENDATIONS	70
10.1	RESOURCE UNCONSTRAINED RESULTS	70
10.2	RESOURCE CONSTRAINED RESULTS	71
10.3	RECOMMENDATIONS	72
10.3.1	IMPROVING THE CURRENT VERSION OF PLEP.....	72
10.3.2	EXTENDING PLEP.....	74
10.4	FINAL REMARKS	77
	APPENDIX A: SMALL AEC PROJECT (FICTITIOUS PROJECT).....	78
	APPENDIX B: LARGE AEC PROJECT (INDUSTRY PARTNER).....	79
	APPENDIX C: SOURCE CODE AND OTHER MATERIAL	81
	BIBLIOGRAPHY	82

LIST OF FIGURES

FIGURE 1: DEPENDENCY STRUCTURE MATRIX.....	3
FIGURE 2: A DEPENDENCY BETWEEN TWO TASKS.....	4
FIGURE 3: OPTIMIZED PARTIAL PROCESSES VS. OPTIMIZED OVERALL PROCESS	6
FIGURE 4: PROCESS MODELS AND INTERDEPENDENCIES BETWEEN HIGH LEVELS OF DETAIL	7
FIGURE 5: CALCULATING THE COMPLETION WEIGHT FOR EACH STATUS LEVEL WHEN THE DEP IS ASSIGNED TO A DELIVERABLE.....	10
FIGURE 6: TASK- DELIVERABLE RELATIONSHIP: READ.....	11
FIGURE 7: TASK- DELIVERABLE RELATIONSHIP: MODIFY	11
FIGURE 8: TASK-DELIVERABLE RELATIONSHIP: CREATE.....	12
FIGURE 9: VISUAL SUMMARY OF ALL TASK-DELIVERABLE RELATIONSHIPS	12
FIGURE 10: ALL POSSIBLE RELATIONS.....	13
FIGURE 11: A SUMMARY VIEW OF THE SPECIFIED COMPONENTS AND RELATIONS BETWEEN THEM.....	14
FIGURE 12: ENTITIES OF RULE 1.....	16
FIGURE 13: ENTITIES OF RULE 2.....	17
FIGURE 14: ENTITIES OF RULE 3.....	18
FIGURE 15: STRONGLY CONNECTED COMPONENTS ARE HIGHLIGHTED.....	21
FIGURE 16: LOOPS AND POSSIBLY MULTIPLE EDGES ARE CREATED WHEN CYCLES ARE CONTRACTED.....	21
FIGURE 17: TOPOLOGICALLY SORTING THE RST DIGRAPH YIELDS THE LOGICAL STEP SCHEDULE	23
FIGURE 18: PARALLEL TASK EXECUTION AND RESOURCE UTILIZATION	25
FIGURE 19: A TASK CAN ONLY REQUIRE THE SAME TOOL ONCE	27
FIGURE 20: RESOURCE LOADING OF A LOGICAL STEP BEFORE <i>TASK-SHUFFLING</i>	30
FIGURE 21: RESOURCE LOADING OF A LOGICAL STEP AFTER <i>TASK-SHUFFLING</i>	31
FIGURE 22: SPREAD OF TASK DURATIONS IN A LOGICAL STEP.....	31
FIGURE 23: A MOVE IS REALIZED BY MEANS OF EDGE INSERTION.....	34
FIGURE 24: SLACK	35
FIGURE 25: HOW THE <i>TABU SEARCH</i> EXPLORES THE SOLUTION SPACE.....	36
FIGURE 26: <i>TABU-SEARCH</i> FLOW DIAGRAM.....	37
FIGURE 27: THE GUI FOR SPECIFYING TASKS, PERSONS, TOOLS AND DELIVERABLES	39
FIGURE 28: THE GUI FOR SPECIFYING TASK-DELIVERABLE RELATIONSHIPS	39
FIGURE 29: THE GUI FOR SPECIFYING ATTRIBUTES, STATUSES, AND DEPS	40
FIGURE 30: THE GUI TO VIEW A BASIC GANTT-TYPE CHART OF THE UCLSS OR CCLSS	40
FIGURE 31: GRAPH INTERFACES.....	42
FIGURE 32: ACTUAL GRAPH IMPLEMENTATION	43
FIGURE 33: THE DFS ALGORITHM PUBLISHES EVENTS AS IT TRAVERSES THE GRAPH.....	45
FIGURE 34: SIMPLE CLASS DIAGRAM OF PLEP CLASSES.....	49
FIGURE 35: <i>PLEPModel</i> CLASS ATTRIBUTES AND METHODS.....	50
FIGURE 36: CALCULATING THE DURATION OF A TASK	53
FIGURE 37: <i>RULEMachine</i> CLASS ATTRIBUTES AND METHODS	53
FIGURE 38: THE MAIN COMPONENTS OF THE LOGICAL STEP SCHEDULE.....	54
FIGURE 39: REQUIRED CLASSES TO MODEL LOGICAL STEP SCHEDULE.....	57
FIGURE 40: PLOTS THE BEST CCLSS FOUND AT EACH <i>TABU SEARCH</i> ITERATION FOR THE SMALL AEC PROJECT ..	62
FIGURE 41: PLOTS THE BEST CCLSS FOUND AT EACH <i>TABU SEARCH</i> ITERATION FOR THE LARGE AEC PROJECT..	64
FIGURE 42: CCLSS DURATION FREQUENCY PLOT OF THE LARGE AEC PROJECT.....	64
FIGURE 43: A CCLSS WITH A SHORTER DURATION THAN THE UCLSS	65
FIGURE 44: HYPERBOLIC GRAPH VIEW OF AN AEC PROCESS MODEL	75
FIGURE A.1: SMALL AEC PROJECT’S UCLSS WITH TASK-SHUFFLING APPLIED.....	78
FIGURE A.2: SMALL AEC PROJECT’S CCLSS WITHOUT TASK-SHUFFLING AND WITH <i>TABU-SEARCH</i> APPLIED	78
FIGURE A.3 SMALL AEC PROJECT’S CCLSS WITH TASK-SHUFFLING AND <i>TABU-SEARCH</i> APPLIED	78
FIGURE B.1: LARGE AEC PROJECT’S UCLSS WITH TASK-SHUFFLING APPLIED	79
FIGURE B.2: LARGE AEC PROJECT’S CCLSS WITHOUT TASK-SHUFFLING AND WITH <i>TABU-SEARCH</i> APPLIED.....	79
FIGURE B.3 LARGE AEC PROJECT’S CCLSS WITH TASK-SHUFFLING AND <i>TABU-SEARCH</i> APPLIED.....	79

LIST OF TABLES

TABLE 1: OVERVIEW OF HOMOGENEOUS BINARY RELATIONS	14
TABLE 2: NUMBER OF USER SPECIFIED COMPONENTS AND RELATIONSHIPS FOR SMALL AEC PROJECT	61
TABLE 3: RESULTS OF A SMALL AEC PROJECT PLEP PROCESS MODEL	61
TABLE 4: NUMBER OF USER SPECIFIED COMPONENTS AND RELATIONSHIPS FOR LARGE AEC PROJECT	63
TABLE 5: RESULTS OF A LARGE AEC PROJECT PLEP PROCESS MODEL	63

1 BACKGROUND

The use of process models in the analysis, optimization and simulation of processes has proven to be extremely beneficial in the instances where they could be applied appropriately. The motor manufacturing industry, for example, has reaped huge benefits by applying process models to deliver products of high quality, while increasing their profit margins. As a result a significant amount of research has been done in all fields of engineering with regards to process models. Until now, however, the Architecture/Engineering/Construction (AEC) industries have not been very successful in this regard, since its processes present unique challenges that complicate their mapping to appropriate generic process models.

AEC projects are complex and volatile by nature. Modern AEC projects can be enormous in budget and physical size, involve many interested parties ranging from multi-national conglomerates to an intricate mixture of small companies and include a wide range of technologies. All of these characteristics add to the complexity of an AEC project and in particular to the complexity of planning it.

AEC projects are normally executed only once and the process cannot be streamlined like the manufacturing of a motor vehicle. Therefore, the possibility of mass production at project level is ruled out. The variations from one AEC project to the next are induced by factors such as unique client requirements, contract structures, and budget constraints. Furthermore, projects typically undergo constant modifications during the planning and execution phase due to factors such as weather conditions, site conditions, changes in the supply of building materials, changing labour market, and so forth. A proper risk analysis can alleviate the impact of the inherent volatility of AEC projects, but the process model must be able to embrace these constant modifications.

Attempts at formulating standardized process models are weighed down by these variations, and typically result in models that are too complex to be of practical use. Some success has been achieved in the design of workflows that encapsulate best practice approaches in executing AEC projects. However, the main downfall of the existing types of process modeling is that the consistency of results is not guaranteed, since they are all adapted by specialists who rely on experience, intuition and empirical guidelines.

2 STATE OF THE ART

After the scope of the AEC process is defined a process model evolves through three steps. First the activities have to be identified (and maintained), then the dependencies between the activities have to be identified (and maintained), and then the process model can be evaluated and the results measured. The results may be fed back into the process model to improve it. Specialists have a wide variety of techniques and tools at their disposal to create, maintain, evaluate and measure process models. It is important that the techniques and tools are easy to use, efficient, deliver consistent results and interact seamlessly with the process model.

To identify and maintain the activities in a process model a Work Breakdown Structure (WBS) is typically used to decompose the process and collect all the activities. It is very important to carefully decompose the process into meaningful activities and this is achieved by a group of specialists from different functional groups involved in the AEC project. Together they list all the activities that comprise the process as a whole. Once all the activities are collected the dependencies between the activities have to be identified. The Dependency Structure Matrix (DSM) is a technique and tool that visually aids the specialists to list the dependencies between the activities. In Figure 1 the directed graph [1] represents the specialists' knowledge about the process model which consists of activities and dependencies. The specialists document the dependencies in the DSM by marking an *X* in the corresponding rows and columns. Referring to Figure 1, activity 3 is dependent on activity 1. In other words activity 1 has to be executed before activity 3. This dependency between activity 3 and activity 1 is shown as an *X* in row 3 and column 1. An activity cannot be executed before itself therefore the diagonal is blacked out. When all the dependencies are crossed off the DSM is transformed into lower triangular form. There might be instances where it is impossible to transform the DSM into lower triangular form, because circuits exist between activities. An example of a circuit can be seen in Figure 1 where activity 4 has to be executed before activity 6, but activity 6 also has to be executed before activity 4. Circuits are indicated by colored squares. Circuits are removed from the DSM by a procedure called tearing which involves the removal of a minimum number of intra-circuit dependencies. Essentially the purpose of tearing is to choose an entry and exit activity for the circuits. When the DSM is in lower triangular form the row or column order of the activities represents the sequence in which the activities have to be executed, as seen on the right side of Figure 1. The DSM offers

specialists a compact matrix view of the whole process model that makes the input of dependencies easier. DSM also provides an opportunity to apply matrix algebra algorithms to produce certain results.

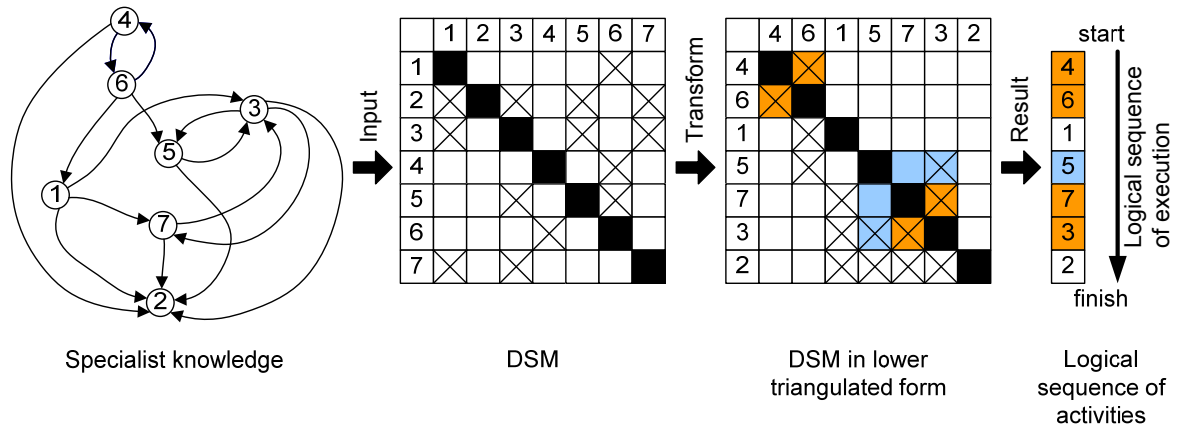


Figure 1: Dependency Structure Matrix

At this stage the basic process model is created and a logical sequence of activities is available as input for further evaluations. The results produced from the evaluations are measured against criteria. Satisfactory results can be used directly by the specialists or can be fed back into the process model to improve it. If the results are not satisfactory, the process model has to be adapted and reevaluated. A process model may go through several evaluation and measure iterations until expected criteria are met.

The procedure, as outlined above, is an industry accepted recipe. Although WBS and DSM might not be universally used it does represent a formalization of a generic thinking procedure. In essence, all specialists go through the same generic thinking procedure to create and modify a process model. The identification and modification of activities is a relatively straightforward task; however, the identification and modification of dependencies between activities is very difficult. Even with the aid of techniques and tools such as DSM the task of dependency identification and modification remain complex, inefficient and prone to errors. The process model must be easily adaptable according to the results obtained from the process model evaluations.

3 PROBLEM STATEMENT

Identifying and maintaining the activities, also known as Tasks, is a straightforward undertaking. Although Tasks may depend on each other, the dependencies between the Tasks (also known as Task-Task dependencies) are modeled as separate entities. The Tasks themselves are isolated elements which make their identification and modification a relatively easy undertaking.

Unfortunately, Task-Task dependencies are not so straightforward to identify and maintain. A *Task-Task* dependency has a meaning, a reason, and two Tasks in between which the dependency exists. In the context of AEC planning processes the meaning of all Task-Task dependencies is “*has to be executed before*”. From the “*has to be executed before*” meaning it can be inferred that the Task-Task dependency indicates an ordering between the two Tasks. The reason of a Task-Task dependency specifies why the dependency exists and every dependency can have a different reason. Knowing that a Task-Task dependency indicates an ordering between two Tasks it is therefore necessary to specify from- and to which Task the dependency will exist. See Figure 2 for an explanation.

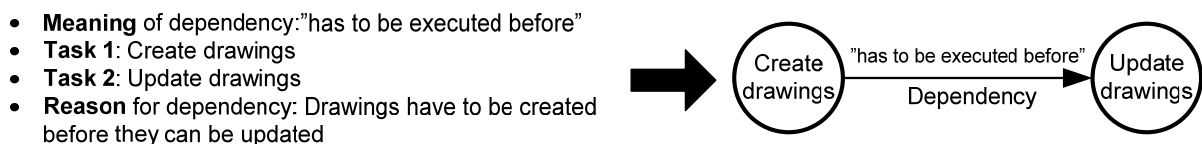


Figure 2: A dependency between two tasks

Although some Task-Task dependencies are obvious and easy to identify, as in Figure 2; there will be others that are obscure. Nevertheless it is essential that all the Task-Task dependencies are correctly identified and maintained otherwise the process model is inconsistent and can produce incorrect results. It only requires the omission or misinterpretation of one dependency for a process model to produce incorrect results.

Current techniques and tools help specialists with the definition and visualization of Task-Task dependencies, but not with the specification of the reason and the Tasks between which the Task-Task dependencies reside. Specialists rely heavily on experience and intuition when specifying the Task-Task dependencies. The reliance on subjective aids might not pose a

problem when dealing with small AEC projects, but as the size of AEC projects increases, specialists experience problems with the identification and maintenance of Task-Task dependencies.

The volatility of AEC projects compounds the problem. The process model must be synchronized with the constantly changing AEC project. The changes range from adding-, removing-, and modifying Tasks to adding-, removing-, and modifying Task-Task dependencies. In order to make consistent changes the specialist must be able to interpret the current state of the process model. Specialists are experiencing problems reinterpreting the reason for each dependency, because the reasons are seldom fully documented and are the product of a subjective procedure. The end results are that large parts of the process model are scrapped and specified again to eliminate confusion and reduce inconsistencies.

The results produced from process models may be fed back into the process model to improve it. Feedback iterations require modifications to the process model and the same difficulties as mentioned above are exposed.

To create and maintain a process model is a labour intensive exercise and it does not scale well. As the size of AEC projects increases the usefulness of the process model decreases, because it requires too much work to create and maintain. Besides being labour intensive the consistency of the process model is difficult to guarantee. In desperation the specialists attempt to simplify the procedure by optimizing partial processes and concatenating these to create an overall optimized process. However, concatenation of optimized partial processes does not guarantee an overall optimized process. This phenomenon can be explained due to the interdependencies between activities of different partial processes. When these interdependencies are taken into consideration the optimized partial processes will be interrupted by activities or sequences of activities from other partial processes. See Figure 3 as an example. Another approach to simplify process modeling is to optimize less detailed process models. However, it is not possible to extract a detailed optimized process from a less detailed process model. Figure 4 shows an example where an optimized process is modeled at a low detailed level. From the example it becomes clear that interdependencies between optimized more detailed processes nullifies the optimized low detail process model.

A new process model [2] named PLEP; has been developed to address the mentioned problems. PLEP introduces the concept of a deliverable and the dependencies between Tasks and Deliverables (Task-Deliverable dependencies). In PLEP Task-Task dependencies are not directly specified, but are calculated from Task-Deliverable dependencies. It is easier to specify and maintain Task-Deliverable dependencies and then to calculate the Task-Task dependencies than directly specifying and maintaining Task-Task dependencies. See Chapter 4 for a complete explanation of PLEP.

Unfortunately PLEP does not take resource constraints into account. Resources are expensive and it may be impossible to secure sufficient resources due to scarcity. It is therefore important to model the impact resource constraints will have on the results produced by PLEP. Of particular interest is the impact resource constraints will have on the sequence of Tasks which is one of the main results produced by PLEP.

In the following sections PLEP is described and explained in detail. Resource constraints are introduced to PLEP and a strategy is investigated to modify PLEP to produce results which account for resource constraints.

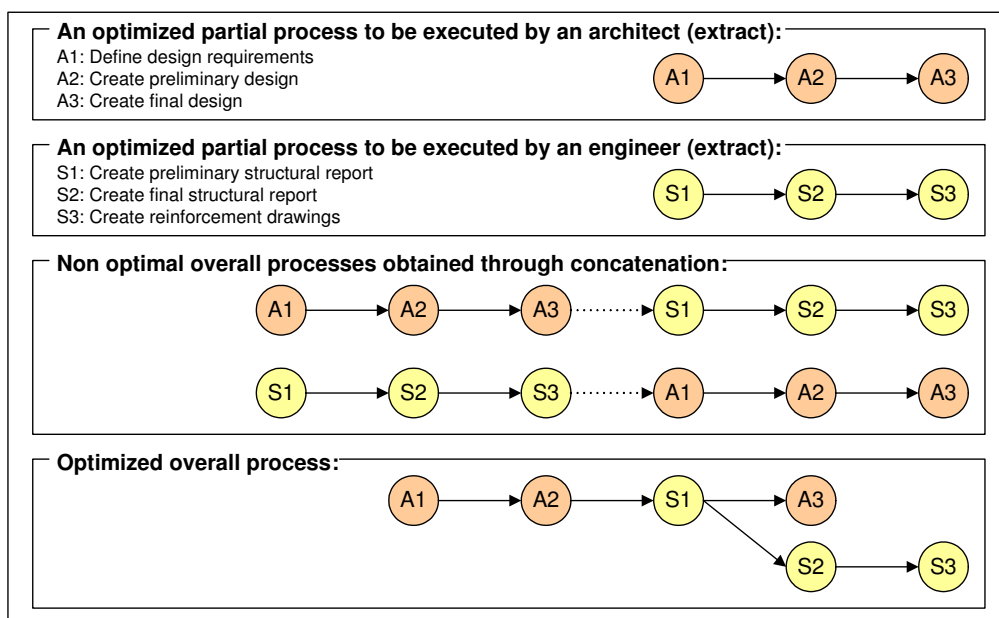


Figure 3: Optimized partial processes vs. optimized overall process

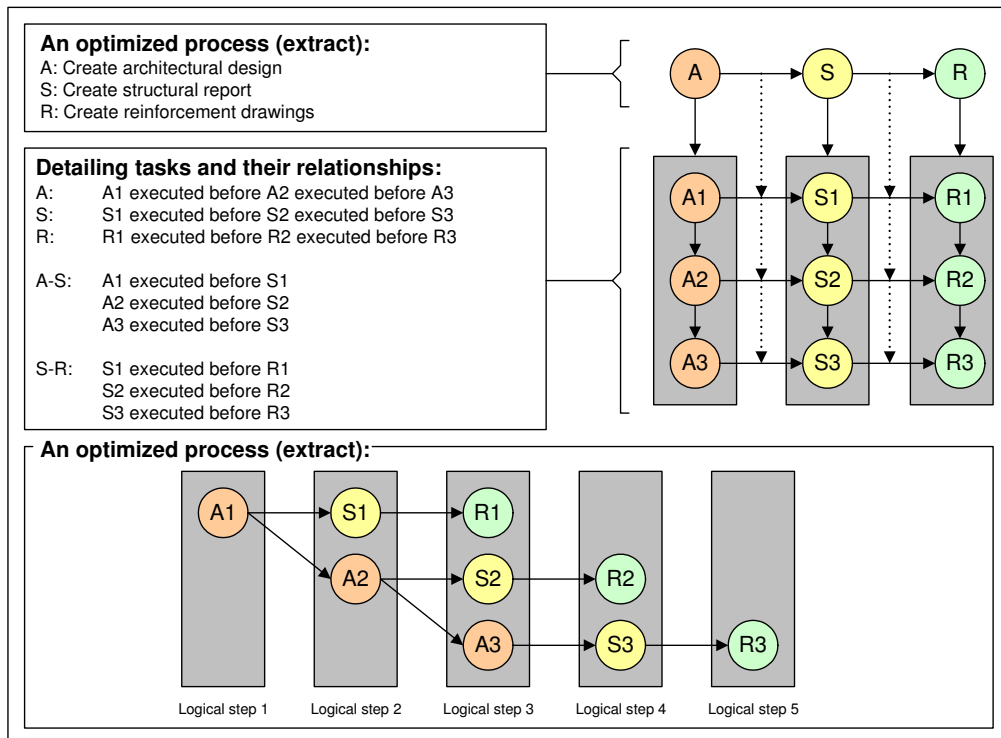


Figure 4: Process models and interdependencies between high levels of detail

4 PLEP: Planning the Engineering Process

A simple Engineering process model, based on the specification of Tasks, Deliverables, Persons and Tools, and certain relations between them, has been developed, and its advantages over conventional techniques have been illustrated [2] [3]. This model is based on the premise that the engineering planning process concerns itself with the stage-wise development of Deliverables. These Deliverables may be drawings, reports, specifications, analysis, design sheets, etc. Persons specifically execute Tasks for the purpose of developing the Deliverables. The stages in the evolution of Deliverables have given Status values, e.g. *preliminary, engineered, checked, final*. Tools are used to operate on Deliverables.

By mapping Tasks, Deliverables, Persons and Tools to vertices [1], and the relationships between these components to edges [1], directed graphs can be formed. The directed graphs are used to mathematically derive results such as the logical sequence of Tasks. In following sections the new process model is discussed in detail.

4.1 SPECIFIED COMPONENTS: The basic ingredients

The specified components are the basic entities that the user must specify to produce a complete PLEP process model.

4.1.1 TASK

A set of Tasks represent the work that needs to be done. Tasks are terminal work activities that are typically the leaf nodes in a Work Breakdown Structure (WBS); it is not further subdivided. Terminal activities are the items that are estimated in terms of resource requirements, budget and duration, linked by between each other by dependencies, and scheduled. When a Task operates on a Deliverable, the Deliverable is assigned a Status. The Status represents the work performed by the Task on the Deliverable.

4.1.2 DELIVERABLE

Deliverables represent the information that is created, modified or read during the execution of Tasks. A Deliverable can be in the form of technical drawings, technical models, etc. A Deliverable must be assigned a Deliverable Evolution Profile (DEP) which contains a collection of Statuses through which the Deliverable has to evolve as Tasks operate on the

Deliverable. Each Deliverable is assigned a completion weight that is required to develop the Deliverable to completion. A *Task* raises a Deliverable's Status to a known level after the Deliverable has been created or modified. The Status must be sourced from the DEP of the Deliverable.

4.1.3 PERSON

A set of Persons represent the people that execute Tasks and is considered a resource.

4.1.4 TOOL

A set of Tools represent the tools that are used to edit Deliverables and is considered a resource. Tools can be in the form of software programs, drawing boards etc.

4.1.5 STATUSES

A set of Statuses represent all the possible development levels the set of Deliverables of a project can evolve through. The set of Statuses must be totally ordered; therefore a unique natural number must be assigned to each Status known as the Status's rank. Statuses and their rankings are sourced from best practice methodologies.

4.1.6 DELIVERABLE EVOLUTION PROFILE

A Deliverable Evolution Profile (DEP) is a subset of the set of Statuses through which a Deliverable evolves to reach completion. For example, a Drawing-DEP may contain the Statuses of Created, Corrected and Signed while a Structural Analysis-DEP may contain the Statuses of *Created* and *Analysed*. Each Status in the DEP is assigned a *Percentage of Completion* (PoC) that is specific to the DEP the Status is located in. The PoCs must sum up to one hundred percent. When the DEP is assigned to a Deliverable the PoCs are used to calculate completion weights for each Status level. See Figure 5 for an explanation.

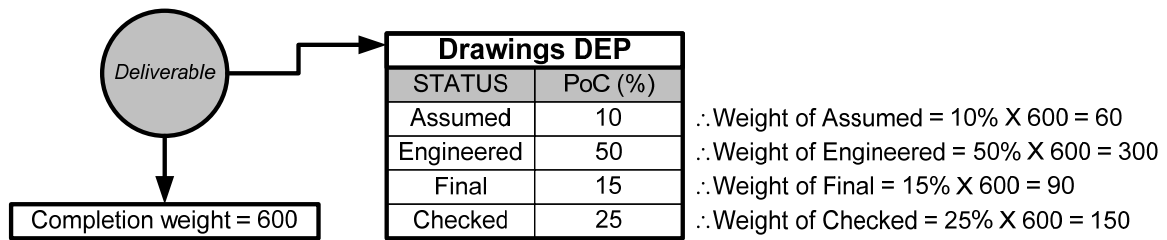


Figure 5: Calculating the completion weight for each Status level when the DEP is assigned to a Deliverable.

4.2 SPECIFIED RELATIONS: Connecting the components

Given the four sets of Tasks, Deliverables, Persons and Tools sixteen possible binary relations, as seen in Figure 10, can be identified between the components. Twelve of the sixteen binary relations are heterogeneous [4] and the remaining four are homogeneous [4]. In traditional process models all sixteen binary relations have to be specified manually to set up a complete process model. To reduce the effort of specification and the possibility of erroneous input, only three of the sixteen binary relations have to be specified while the remaining binary relations are mathematically derived. Refer to Table 1 for an overview of the sixteen binary relations, their meanings that are also known as semantics, and which three relations have to be specified. The three manually specified heterogeneous binary relations will be discussed in detail in the following sections.

4.2.1 TASK-DELIVERABLE RELATION

Deliverables are produced or operated upon during the execution of Tasks. Three different types of Task-Deliverable relationships can be specified from a Task to a Deliverable:

- A Task can read a Deliverable
- A Task can modify a Deliverable
- A Task can create a Deliverable

A Task cannot delete a Deliverable since records must always be available for future reference. Multiple Task-Deliverable relationships can be assigned from a Task to a Deliverable object, but the same Task cannot operate on the same Deliverable more than once.

Read

A “*read*” Task-Deliverable relationship indicates that when the Task is executed the Task reads the Deliverable. It is not compulsory that each Task must read a Deliverable or that each Deliverable must be read by a Task.

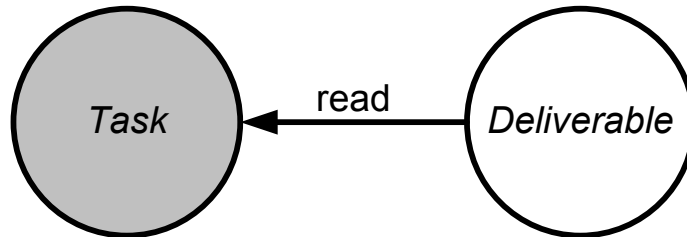


Figure 6: Task- Deliverable relationship: read

Modify

A “*modify*” Task-Deliverable relationship indicates that when the Task is executed the Task modifies the Deliverable. During modification a Task actually reads and overwrites the same Deliverable, but it is considered a single relationship. The Status level of the Deliverable is increased when the modification is completed and the assigned Status is a property of the “*modify*” Task-Deliverable relationship.

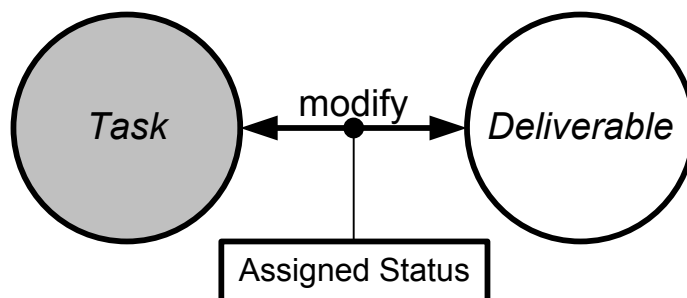


Figure 7: Task- Deliverable relationship: modify

Create

A “*create*” Task-Deliverable relationship indicates that when the Task is executed the Task creates the Deliverable. Not all Deliverables need to be created by Tasks in the process model as Deliverables may already be in existence. Not all Tasks have to create Deliverables either. Deliverables are created at a specific Status level and the assigned Status is a property of the “*create*” Task-Deliverable relationship.

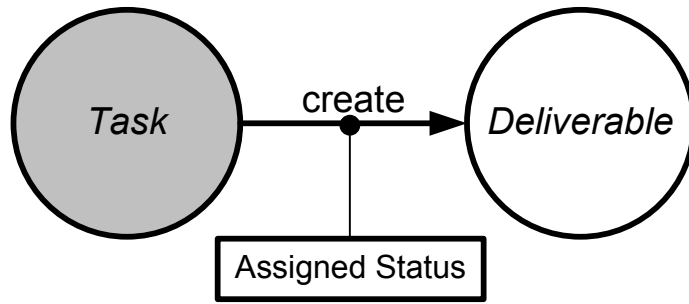


Figure 8: Task-Deliverable relationship: create

Refer to Figure 9 for a visual representation of all possible Task-Deliverable relationships and the definition of input- and output Deliverables. Modification of Deliverables is considered as input and output.

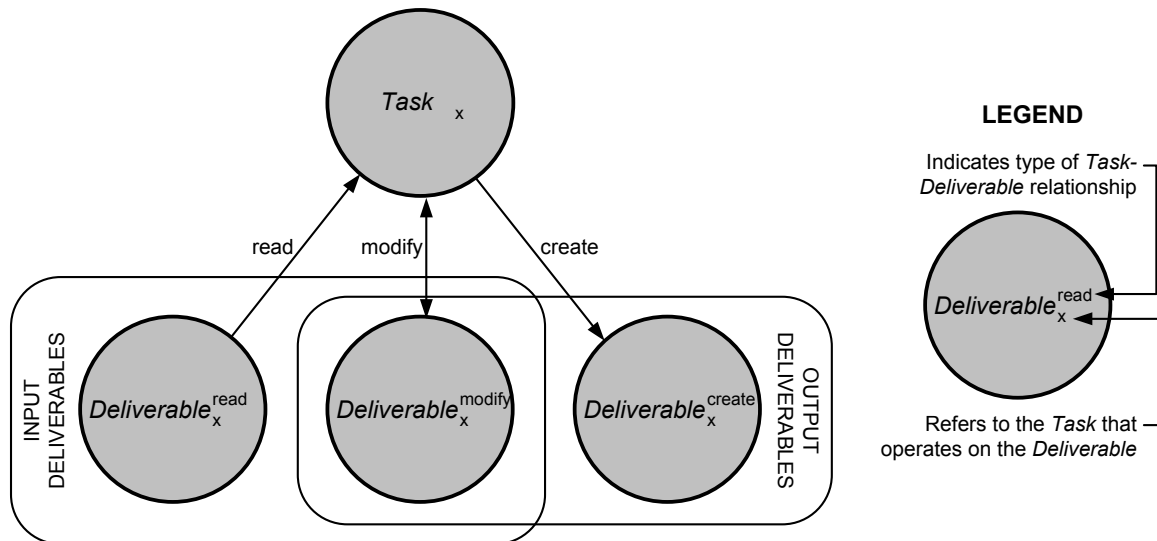


Figure 9: Visual summary of all Task-Deliverable relationships

4.2.2 TASK-PERSON RELATION

The responsibility of a Person to execute a Task is assigned by Task-Person relationship. The semantics of this relationship should be interpreted as a Task “*is executed by*” a Person. Even if the Task is automated, the responsibility must still be assigned. Thus, a Task must always be assigned at least one Person with the possibility that multiple Persons might be assigned to the same Task. However it is not allowed to assign the same Person to the same Task more than once. It is important to note that only the responsibility of Task execution is assigned by a Task-Person relationship and that assigning more Persons to a Task will not reduce the execution duration of the Task.

4.2.3 DELIVERABLE-TOOL RELATION

Deliverables are read, modified or created using a Tool. A Tool can take on many different forms ranging from CAD software to a drawing board. The semantics of this relationship should be interpreted as a Deliverable “*is edited by*” a Tool. When this relationship is specified a Tool operates on a Deliverable. At least one Deliverable-Tool relationship should be specified for each Deliverable. Multiple Tools can be assigned to a Deliverable; however it is not allowed to assign the same Tool to the same Deliverable more than once. It is important to note that the Deliverable-Tool relationship only assigns which Tools operate on a Deliverable and that assigning more Tools to a Deliverable will not reduce the execution duration of the Tasks operating on the Deliverable.

	Person	Task	Deliverable	Tool
Person	Personnel loading	executes	access (read/write)	use
Task	is executed by	uCST graph	access (read/write/modify)	requires
Deliverable	is accessed by (read/write)	is accessed by (read/write/modify)	Deliverable evolution	is edited by
Tool	is used by	is required by	edit	Tool loading

*Orange highlighted binary relations are user specified.

Figure 10: All possible relations

Only the three heterogeneous binary relations, as described above, need to be specified. The remaining heterogeneous binary relations can be mathematically derived by either finding the inverse of a specified relation, the composition of more than one of the specified relations or by a combination of both operations.

The homogenous binary relations are located on the diagonal of the matrix in Figure 10 and are derived from the specified relations. The relation in the set of Tasks is the key derived relation and is of special interest. In the following sections the relation in the set of Tasks will be discussed in detail.

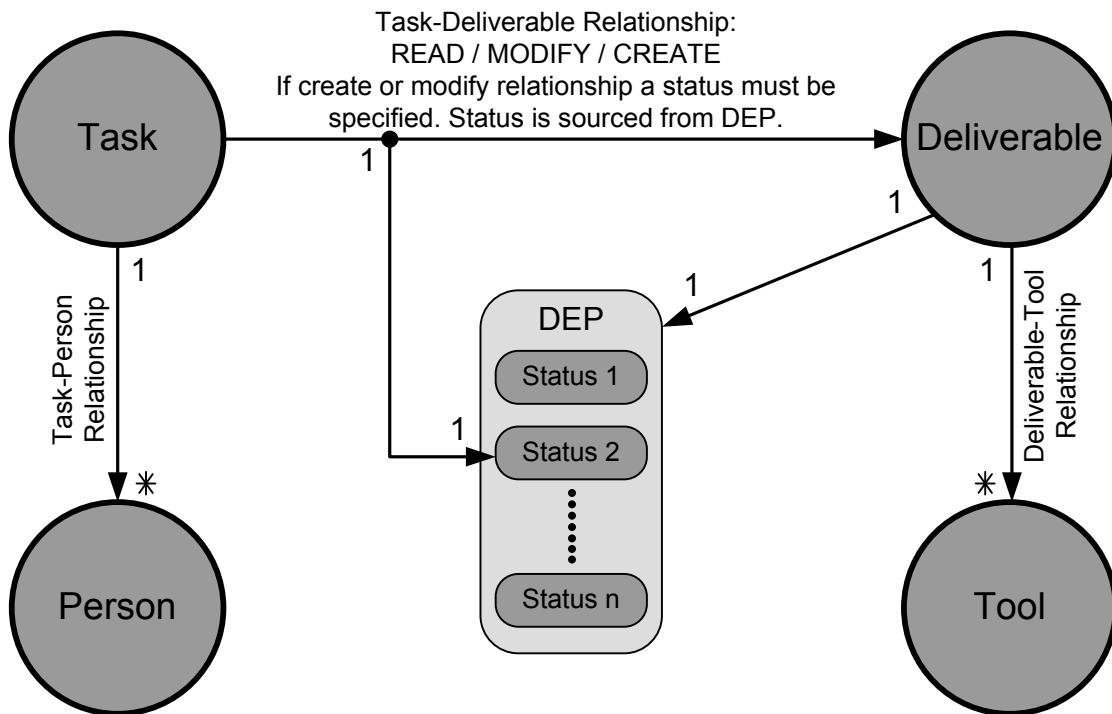


Figure 11: A summary view of the specified components and relations between them

4.3 MATHEMATICALLY DERIVED RELATIONS

The derived homogeneous binary relations provide a wealth of information that can be used to interpret the underlying engineering process model. Of all the derived relations the relation in the set of Tasks is the most important, because the derivation of the remaining three homogeneous binary relations requires the relation in the set of Tasks as input. See Table 1 for the meaning of the derived relations as well as the information required to derive the relations.

Table 1: Overview of homogeneous binary relations

Derived homogeneous binary relation	Meaning of relation	Required information to derive relation
Relation in the set of <i>Tasks</i>	Logical sequence of <i>Tasks</i>	<i>Task-Deliverable</i> relation and three predefined rules
Relation in the set of <i>Persons</i> (<i>Person-Person</i> relation)	<i>Person</i> loading (Which <i>Person</i> is utilized in what logical step)	Relation in the set of <i>Tasks</i> and <i>Task-Person</i> relation
Relation in the set of <i>Tools</i> (<i>Tool-Tool</i> relation)	<i>Tool</i> loading (Which <i>Tool</i> is utilized in what logical step)	Relation in the set of <i>Tasks</i> and <i>Task-Tool</i> relation
Relation in the set of <i>Deliverable</i> (<i>Deliverable-Deliverable</i> relation)	<i>Deliverable</i> evolution (What a <i>Deliverable</i> 's <i>Status</i> is at the beginning and end of each logical step)	Relation in the set of <i>Tasks</i> and <i>Task-Deliverable</i> relation

4.3.1 RELATION IN THE SET OF TASKS: Task dependencies

The relation in the set of Tasks (RST) is a derived homogeneous binary relation and thus only contains relationships between the Tasks. The semantics of this relation is “*has to be executed before*” and therefore each Task-Task relationship dictates an order between its incident Tasks. In other words, if Task_x has to be executed before Task_y then a Task-Task relationship from Task_x to Task_y will be present in the RST. The RST is an order relation that can produce a logical execution sequence of Tasks.

Each Task must be compared to every other Task to see if an order relationship exists between the Tasks. The Cartesian product of the set of Tasks produces all possible Task pairings (1). Each Task pairing is checked against three predefined rules and if any of the three rules are satisfied then a “*has to be executed before*”-relationship exists between the two Tasks in the pairing.

Relation in the set of Tasks :=

$$\{ (task_x, task_y) \in T \times T \mid task_x \neq task_y \wedge task_x \text{ “has to be executed before” } task_y \} \quad (1)$$

Where: T is the Set of Tasks

The collection of the “*has to be executed before*”-relationships constitutes the RST. The three predefined rules are discussed in the following section.

4.3.2 PREDEFINED RULES: Enforcing stage-wise evolution of Deliverables

The function of the three predefined rules is to govern the “*has to be executed before*” ordering of Tasks in an engineering project based on information provided by the Task-Deliverable relation. The rules are defined and described below.

RULE 1: A Deliverable has to be created before it can be used:

Rule 1 is always checked before Rule 2 and 3 are applied. If Rule 1 identifies a “*has to be executed before*” relationship then Rule 2 and 3 are not applied. The entities needed to express this rule mathematically are shown in Figure 12.

It is a basic rule that Deliverables have to be created before it can be used. For example, if Deliverable_{D1} is created by Task_x, and the same Deliverable_{D1} is either read or modified by Task_y, the relationship Task_x “has to be executed before” Task_y is true, i.e. the pair (Task_x,Task_y) is an element of the relation in the set of Tasks. Since a Task can read, create or modify more than one Deliverable, the example described above has to be extended to fit generic cases:

Given: $\{deliverable_x^{create}\} = \text{Set of Deliverables created by Task}_x$
 $\{deliverable_y^{read}\} = \text{Set of Deliverables read by Task}_y$
 $\{deliverable_y^{modify}\} = \text{Set of Deliverables modified by Task}_y$

The mathematical representation of Rule 1 is:

If:

$$\{deliverable_x^{create}\} \cap \{\{deliverable_y^{read}\} \cup \{deliverable_y^{modify}\}\} \neq \emptyset \text{ (the empty set)}$$

Then: $(Task_x) R (Task_y) = true$

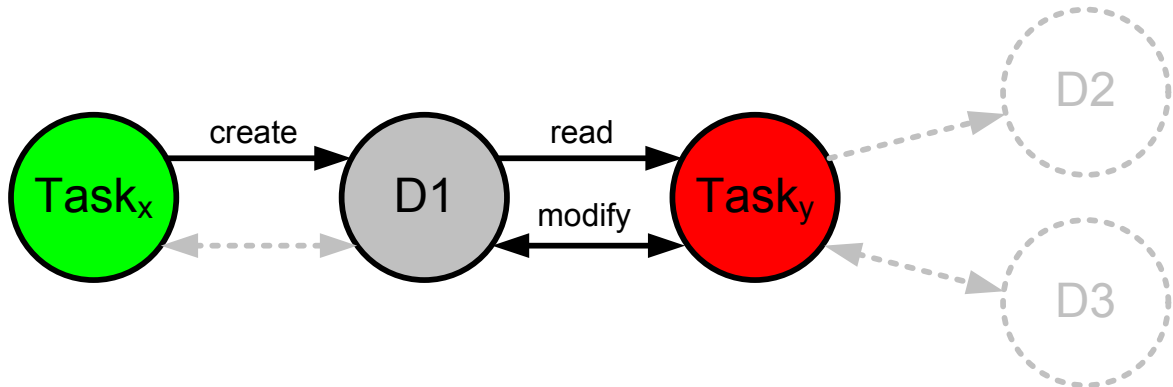


Figure 12: Entities of Rule 1

RULE 2: The status of data has to increase during modification

Rule 2 is always checked after Rule 1 and before Rule 3 is applied. If Rule 2 identifies a “has to be executed before” relationship then Rule 3 is not applied. The entities needed to express this rule mathematically are shown in Figure 13.

Different Tasks can modify the same Deliverable and at the conclusion of each Task the Deliverable has a certain Status, e.g. *preliminary* or *engineered*. For example, Task_x modifies Deliverable_{D1} and increases its Status rank to r(D)_x. Task_y modifies the same Deliverable_{D1} and increases its status rank to r(D1)_y. If r(D1)_x is smaller than r(D1)_y, then the relationship

Task_x “has to be executed before” Task_y is true, i.e. the pair (Task_x, Task_y) is an element of the relation in the set of Tasks. Since a Task can modify more than one Deliverable, the example described above has to be extended to fit generic cases:

Given: $\{deliverable_x^{modify}\} = \text{Set of Deliverables modified by Task}_x$

$\{deliverable_y^{modify}\} = \text{Set of Deliverables modified by Task}_y$

The mathematical representation of rule 2 is:

If: $p \in \{deliverable_x^{modify}\} \cap \{deliverable_y^{modify}\} \wedge r(p)_x < r(p)_y$

Then: $(Task_x) R (Task_y) = true$

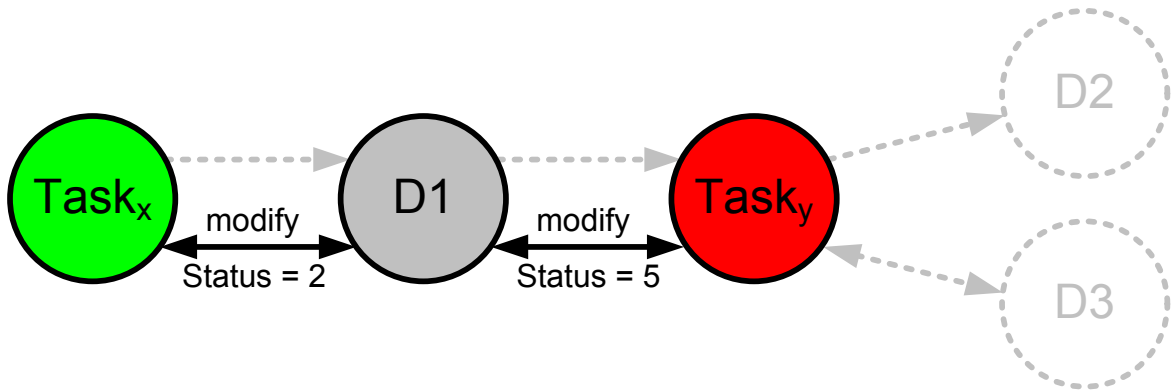


Figure 13: Entities of Rule 2

RULE 3: For any Task, the highest status rank of its output data cannot be better than the lowest status rank of its disjoint input data

Rule 3 is always checked after Rule 1 and 2 have been applied. The entities needed to express this rule mathematically are shown in Figure 14. Refer to Figure 9 for the definition of input and output Deliverables.

Rule 3 focuses on a Task delivering a set of Deliverables which another Task requires as a set of input Deliverables and, the disjoint set of output Deliverables generated. The Statuses of the set of input Deliverables must be at a sufficient level in order to produce a disjoint set of output Deliverables at specific Status levels. For example, the following Status ranking is available: *Assumed* (1) \rightarrow *Engineered* (2) \rightarrow *Final* (3) (low to high). Task_x modifies Deliverable_{D1} to Status level *Engineered*. Task_y reads Deliverable_{D1}, and creates a **different** Deliverable_{D2} at a Status level *Assumed* and modifies another **different** Deliverable_{D3} to a Status level *Final*. Thus, Task_x “has to be executed before” Task_y to ensure that the minimum

Status level of input Deliverables are brought up to a larger Status level compared to the maximum Status level of output Deliverables.

Given: $\{deliverable_x^{modify}\} = \text{Set of Deliverables modified by Task}_x$
 $\{deliverable_y^{read}\} = \text{Set of Datasets read by Task}_y$
 $\{deliverable_y^{create}\} = \text{Set of Datasets created by Task}_y$
 $\{deliverable_y^{modify}\} = \text{Set of Datasets modified by Task}_y$

The mathematical representation of rule 3 is:

If: $INPUT = \{Deliverables\} = \{deliverable_x^{modify}\} \cap \{deliverable_y^{read}\}$
 $OUTPUT = \{Deliverables\} = \{deliverable_y^{create}\} \cup \{deliverable_y^{modify}\}$
 $r(min)_{INPUT} = \text{Minimum Status rank in INPUT}$
 $r(max)_{OUTPUT} = \text{Maximum Status rank in OUTPUT}$
 And if: $r(min)_{INPUT} \leq r(max)_{OUTPUT}$
 Then: $(Task_x) R (Task_y) = true$

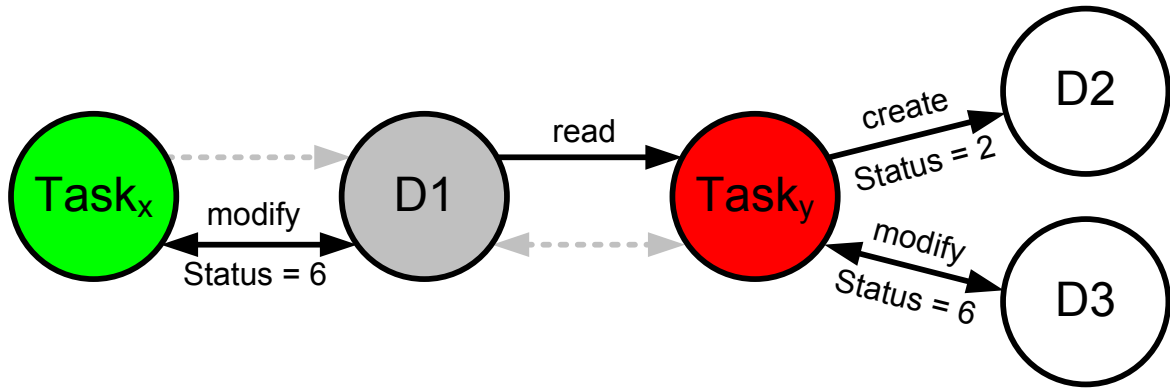


Figure 14: Entities of Rule 3

If Figure 12, Figure 13, and Figure 14 are viewed together it is apparent that Rule 1, 2 and 3 constitute all possible Task-Deliverable relationship permutations between Task_x and Task_y. After all the ordered Task pairs produced by the Cartesian product in the set of Tasks have been subjected to the three predefined rules, a subset of ordered Task pairs that satisfies the rules are collected. This subset constitutes the RST.

The ordering of Tasks in the RST is based on the application of the three predefined rules. In turn the three rules are only concerned with the proper evolution of Deliverables. If Task_x “*has to be executed before*” Task_y according to the three rules it means that Task_x operates on a Deliverable that Task_y require in some way.

The RST can be topologically sorted to produce a logical step schedule and the following section will explain the operations that are necessary to produce a logical step schedule.

5 LOGICAL STEP SCHEDULE: Topological sorting of the relation in the set of Tasks

The RST can be viewed as a directed graph, also called a digraph, and therefore known graph theory algorithms can be applied on the RST digraph. The RST digraph does not directly yield the logical step schedule. The RST digraph has to be topologically sorted to determine the logical step schedule. A logical step schedule depicts the logical sequencing of Tasks and it is not time-based scheduling even though Tasks have the notion of duration. Tasks that can be executed in parallel are located in the same step of the logical step schedule. See Figure 17 for an example of a RST digraph and its corresponding logical step schedule.

The RST digraph must be acyclic before it can be topologically sorted. If the digraph contain any cycles they must be removed.

5.1 DETECTION AND REMOVAL OF CYCLES

The derived RST digraph is not guaranteed to be acyclic. If cycles are present they have to be detected and removed before topological sorting can commence. If cycles are not removed the topological sort algorithm will become trapped in the cycles and fail. Kosaraju's algorithm [5] is used to detect strongly connected components in the RST digraph. Strongly connected components are maximal subgraphs where for every pair of vertices u and v there is a path from u to v and a path from v to u . If a strongly connected component has a cardinality larger than one it is considered a cycle. A strongly connected component may contain nested cycles, but it is considered as one cycle. In Figure 15 the RST digraph contains six strongly connected components. The cardinality of two strongly connected components is larger than one, $\{\text{Task}_B, \text{Task}_C, \text{Task}_D, \text{Task}_E\}$ and $\{\text{Task}_G, \text{Task}_H, \text{Task}_I\}$, and they are therefore considered cycles that must be removed from the RST digraph.

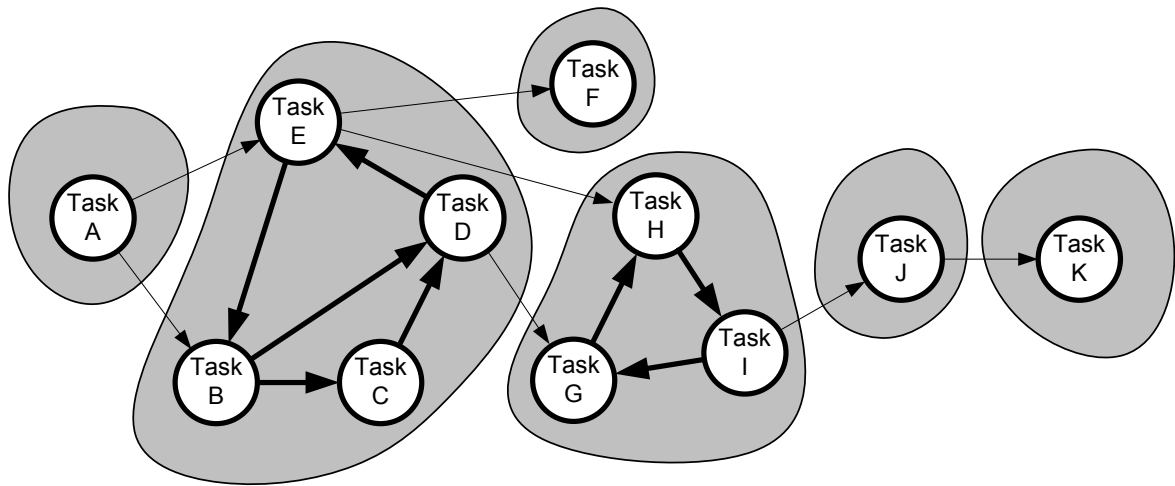


Figure 15: Strongly connected components are highlighted

Cycles in the RST digraph have a specific meaning. Cyclic Tasks are dependent on each other and must be executed as a single unit (in parallel). To model this phenomenon the cyclic Task vertices are contracted into a single super Task vertex (See Figure 16). Contracting the cyclic Task vertices into a single super Task vertex also removes the cycle from the RST digraph. When all the cycles are contracted the RST digraph is acyclic. However, contracting cycles creates loops and possibly multiple edges as meaningless by-products that must to be removed from the RST digraph.

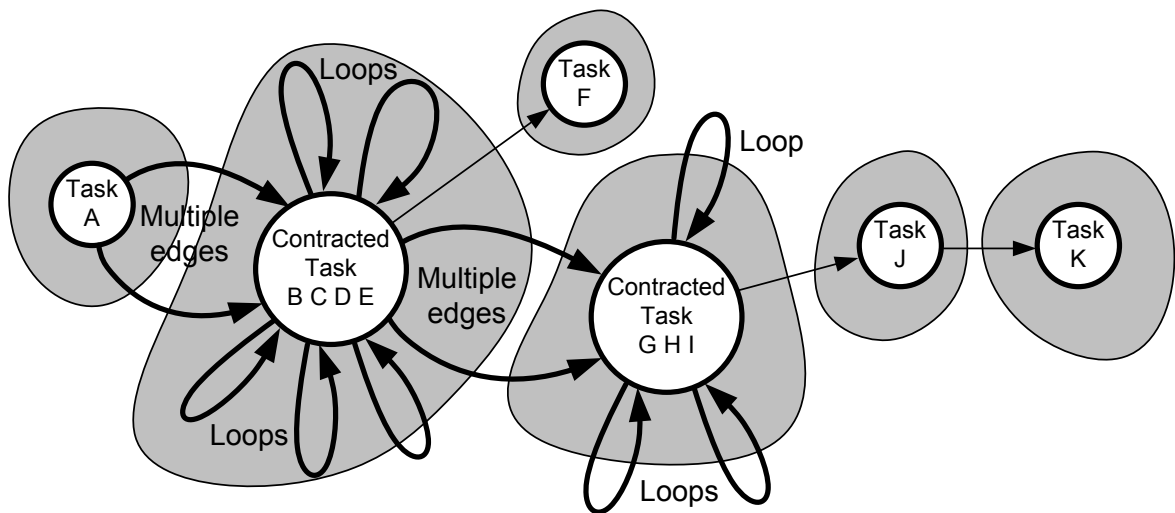


Figure 16: Loops and possibly multiple edges are created when cycles are contracted

5.1.1 LOOPS

Loops are edges whose start- and end vertex are the same vertex. Loops are introduced to the RST digraph only as a by-product of cycle contraction and are never derived in the RST

digraph. Cyclic edges, edges whose start- and end vertex are located in the same cycle, will become loops when the cyclic Task vertices are contracted into a single super Task vertex. Loops have no meaning in the context of the RST, because a Task cannot be executed before itself. Therefore, all loops are removed from the RST digraph.

5.1.2 MULTIPLE EDGES

An edge is considered multiple if there are more edges with the same start- and end vertex. Multiple edges are introduced to the RST graph only as by-products of cycle contraction and are never derived in the RST digraph. If there are edges that share a common acyclic Task vertex and their other incident Task vertices are located in the same cycle; those edges will become multiple edges when the cycle is contracted. In the context of the RST multiple edges have no meaning and therefore are replaced by a single edge with the same start- and end Task vertex.

5.2 TOPOLOGICAL SORTING

Topological sorting can commence when an acyclic RST digraph is available. Topological sorting assigns every Task vertex x in the RST digraph an execution rank $r(x)$, which is a natural number, and indicates in which logical execution step the Task vertex must be executed. Execution rankings have the following properties:

1. A Task vertex x has the execution rank $r(x) = 0$ if it does not have any predecessors.
2. A Task vertex x has the execution rank $r(x) = k > 0$ if it has a k^{th} predecessors and no $(k+1)^{\text{th}}$ predecessors.

The complete Task vertex set V is divided into disjoint subsets V_k where each Task vertex x in subset V_k has an execution rank $r(x) = k$. Task vertex x can only belong to one V_k . All Task vertices in V_k have to be executed before Task vertices in V_{k+1} can be executed.

The subsets V_k have the following ordinal properties:

1. The set V_0 contains all the Task vertices of the lowest execution rank 0. These Task vertices have no predecessors in V . They are therefore minimal and have to be executed first.
2. The set V_{n-1} contains all the Task vertices of the highest execution rank $n-1$. These Task vertices has no successors in V . They are therefore maximal, but there may be

- other maximal Task vertices with no successors that are not in V_{n-1} . Therefore maximal Task vertices in V_{n-1} are executed last.
3. Every Task vertex x in V_k with $k > 0$ has at least one predecessor y in V_{k-1} . If that is not the case x would not have any k^{th} ancestors and therefore would not belong to V_k .
 4. A Task vertex of execution rank k has neither a predecessor nor a successor in V_k .

Each subset V_k is used to populate the k^{th} execution step in the logical step schedule. All the Tasks in an execution step have to be executed before any Task in the next execution step may be executed. The logical step schedule together with the Task-Deliverable, Task-Person, and Deliverable-Tool relations provides to opportunity to derive a wealth of information.

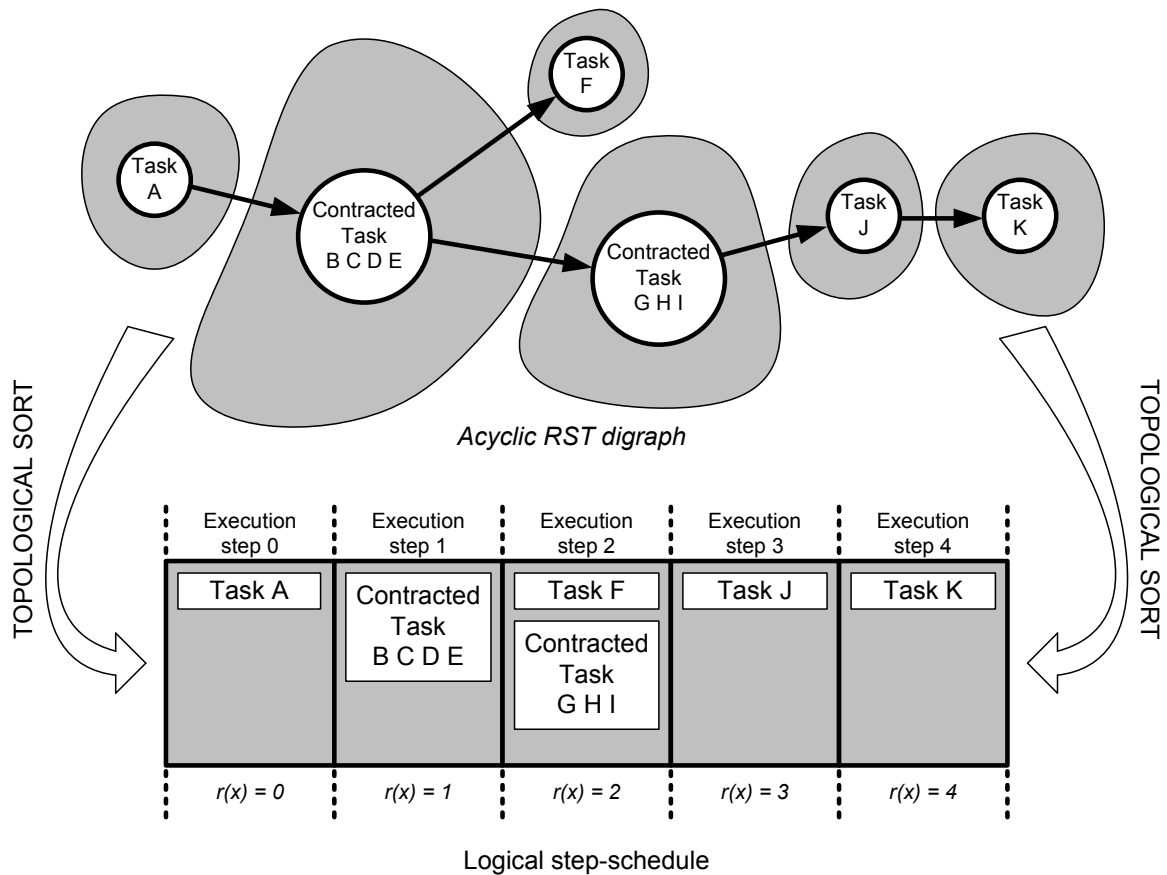


Figure 17: Topologically sorting the RST digraph yields the logical step schedule

5.3 CHARACTERISTICS OF THE LOGICAL STEP SCHEDULE:

The logical step schedule has the following characteristics:

- There are no derived “*has to be executed before*” relationships between Tasks in the same logical step.
- Tasks are placed in the earliest (or lowest ranked) logical step.
- The logical step schedule contains the minimum number of steps.
- The ordering of Tasks into steps in the step schedule is logical ordering and not time-based (calendar time) scheduling.
- Each logical step has a duration. The duration is determined by the Task with the longest duration located in the step. Therefore within each logical step there is a notion of time and durations.
- The total duration of the logical step schedule is the summation of the logical step durations.
- There are no dependencies between Tasks in the same step. Therefore, Tasks in a logical step are assumed to be executed in parallel. Doing so will result in the shortest step durations.

6 RESOURCE CONSTRAINTS

The RST and logical step schedule is based on the proper evolution of Deliverables and resources are not taken into account at any stage. The Task-Person and Deliverable-Tool relation that models resource interaction are not used when the RST is derived. Therefore the RST and logical step schedule are resource unconstrained and consistent in terms of Deliverable evolution. The RST is also known as the resource Unconstrained Consistent Sequence of Tasks (uCST) and the logical step schedule as resource Unconstrained Consistent Logical Step Schedule (uCLSS).

Ignoring the impact of resources is equivalent to assuming that an unlimited amount of resources are available or that the resources are unconstrained. With unlimited resources and no dependencies between Tasks in the same logical step, the assumption that Tasks in the same logical step are executed in parallel is an optimal assumption from a step duration point of view. It guarantees that each step is executed as quickly as possible.

When the impact of limited resources is taken into account it might not be possible to execute all the Tasks in the same logical step in parallel. In Figure 18 three Tasks utilise $Engineer_A$ and one Task utilise $Architect_A$ during their execution. If all the Tasks are executed in parallel three $Engineer_A$ and one $Architect_A$ would be required. If the required Persons are not available resource conflicts are present and the resources are considered over allocated or over utilised. The same principle applies to the utilization of Tools.

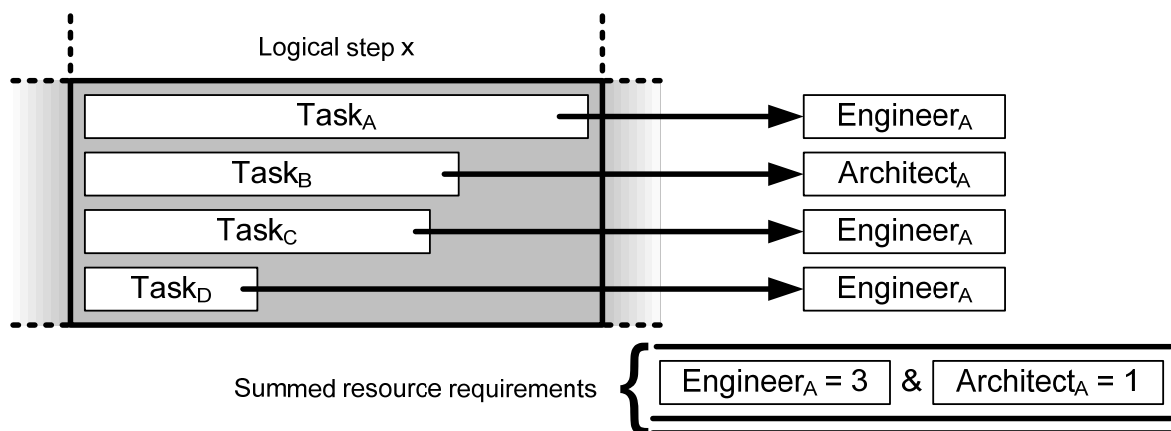


Figure 18: Parallel Task execution and resource utilization

In the next section the model is extended to include the concepts associated with limited resource availability.

6.1 EXTENDING THE PROCESS MODEL

The main aim is to derive a resource Constrained Consistent Sequence of Tasks (cCST), based on the uCST. It is not the aim to derive the required resource availabilities based on a specified schedule. Before a cCST can be derived the concept of limited resource availability has to be incorporated into the process model. First the resource components have to incorporate the concept of availability, secondly resource loading and conflicts have to be identified and thirdly resource conflicts have to be resolved.

6.1.1 RESOURCE AVAILABILITY

Persons and Tools are the resource components of a project and are utilised when Tasks operate on Deliverables. The number of resource components available for utilization during Task execution is the basic information required to define resource constraints.

To simplify the application of resource constraints it is assumed that resources have a constant (non-changing) availability over the duration of the project. Therefore time does not have an influence on the availability of resources.

Persons and Tools are extended by the addition of an availability attribute. The availability attribute is modelled by a natural number.

6.1.2 RESOURCE LOADING AND CONFLICTS

Determining the resource loading of Tasks is vital when resource conflicts are to be identified. A logical step schedule, the uCLSS, and the three specified binary relations are used to determine resource loading (level of utilization) in every logical step. Resource loading for a logical step is compared to the resource availability to identify resource conflicts.

For each Task, the Person and Tool loading is calculated using the three specified relations in conjunction with a logical step schedule: Task-Deliverable, Task-Tool, and Deliverable-Tool

relation. The following assumptions are made and operations are performed to calculate a Task's resource loading.

Person loading of a Task:

To calculate Person loading the Task-Person relation is used to determine which Persons have been assigned to the Task.

A Task can be assigned an arbitrary number of Persons, but the same Person cannot be assigned to the same Task more than once. Therefore for a specific Person, a Task cannot produce a resource requirement of more than one.

Tool loading of a Task:

To calculate Tool loading the Task-Deliverable and Deliverable-Tool relations are concatenated to produce the Task-Tool relation that is used to determine what Tools have been indirectly assigned to the Task.

A Task can operate on the same Deliverable only once and a Deliverable be assigned to the same Tool only once. Referring to Figure 19, when the Task-Tool relation is computed it may seem as if a Task_A requires 2 of Tool_B. This would be true if the Deliverables are operated on by Task_A in parallel. An assumption is made that Deliverables are always operated on by Tasks in series. This assumption prevents a Task from having a resource requirement of more than one for the same Tool.

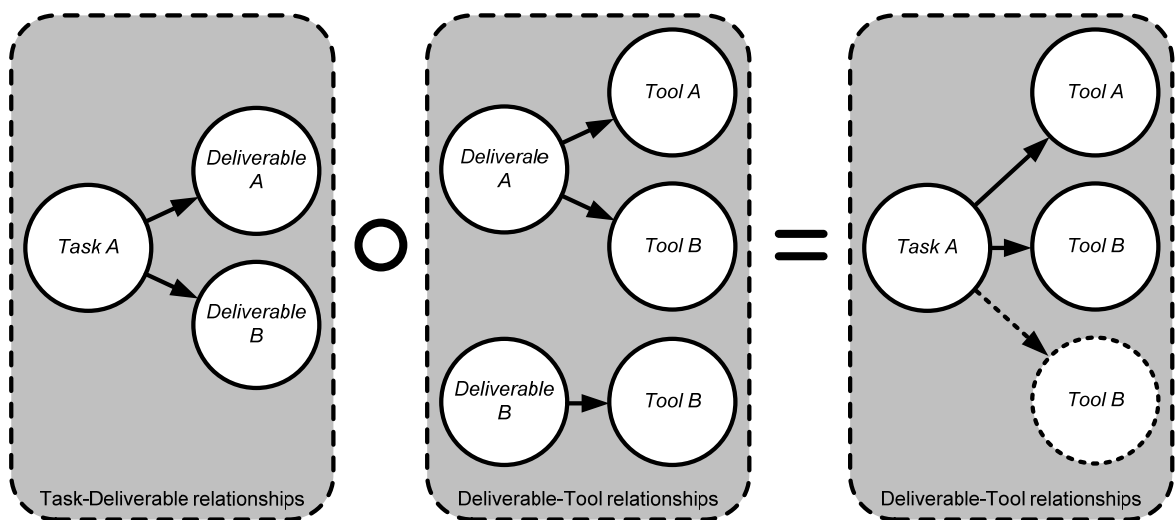


Figure 19: A Task can only require the same Tool once

Resource loading of contracted super Tasks:

Tasks in a cycle have to be executed as a single unit. All the Tasks in a cycle have to be completed before any of their logical successors can commence. The preferred assumption would be to execute the cyclic Tasks in parallel, but some of the cyclic Tasks may be assigned to the same Person. If the Person is assigned more than its availability then some of the cyclic Tasks within the contracted Task will have to be executed in series. To avoid this complex scenario it is assumed that all Tasks within a contracted super Tasks is executed in series. Further research can investigate these fine grained characteristics. If all the Tasks are assumed to be executed in series then the maximum resource loading a contracted super Task can achieve is one per resource.

Since all Tasks, single Tasks or contracted super Tasks, in the same logical step are assumed to be executed in parallel their resource loadings are summed to determine the resource loading of the logical step. A resource is considered to be over allocated or over utilised if its resource loading in a logical step is higher than its availability. Resource conflicts must be resolved if the Tasks are to be executed according to a resource Constrained Consistent Logical Sequence of Tasks (cCLSS). In the following section two strategies to resolve resource conflicts are explained.

7 RESOLVING RESOURCE CONFLICTS

The uCST and uCLSS are very useful results. Unfortunately they do not take resource constraints into account. The problem of deriving a cCST and cCLSS directly for the specified information is complex and has not been solved in closed form. A heuristic approach is proposed here, in which the uCST is used as a starting solution and resource constraint information is injected into the uCST and uCLSS to produce a cCST and cCLSS.

Looking at the uCST and uCLSS there are two options available to reduce and remove resource conflicts. The trivial option is to increase resource availability until resource requirements are satisfied. This option is not desirable, because it may be expensive or impossible to secure sufficient resources. The AEC industry is experiencing severe resource shortages and relying on increasing resource availability is not a reliable option.

The other option, if using the uCST as starting solution, is to modify the uCST and uCLSS by incorporating resource constraint information until a desirable cCST and cCLSS is found. Two strategies are investigated to reduce and ultimately remove resource conflicts. Both strategies are concerned with the reduction of parallel execution of Tasks and thereby the reduction of resource loading.

7.1 TASK-SHUFFLING: Local intra-step Task shifting

Task-Shuffling is a strategy to reduce the overall resource loading. This is achieved by reducing the parallel execution of Tasks and increasing the serial execution of Tasks inside a logical step. There is no guarantee that all resource conflicts will be resolved after this strategy has been performed.

Task-Shuffling operates exclusively in the domain of the uCLSS. It is considered to be a non-intrusive strategy since it does not make structural changes to the uCST. The uCST is exactly the same before and after *Task-Shuffling* has been performed.

7.1.1 DESCRIPTION OF TASK-SHUFFLING OPERATION

There are no “has to be executed before” relationships between Tasks located in the same logical step, because if such relationships were present some of the Tasks should have been positioned in subsequent logical steps. Therefore, inside a logical step no restrictions are placed on the sequence of Tasks.

The absence of resource constraints coupled with the lack of Task sequencing inside a logical step supports the assumption that all Tasks in a logical are executed in parallel. Parallel execution of Tasks in a logical step guarantees minimum logical step duration. In Figure 20 all the Tasks are assigned the same resource. This may be a Tool or a Person. When all the Tasks are executed in parallel the resource loading of Resource_A is six.

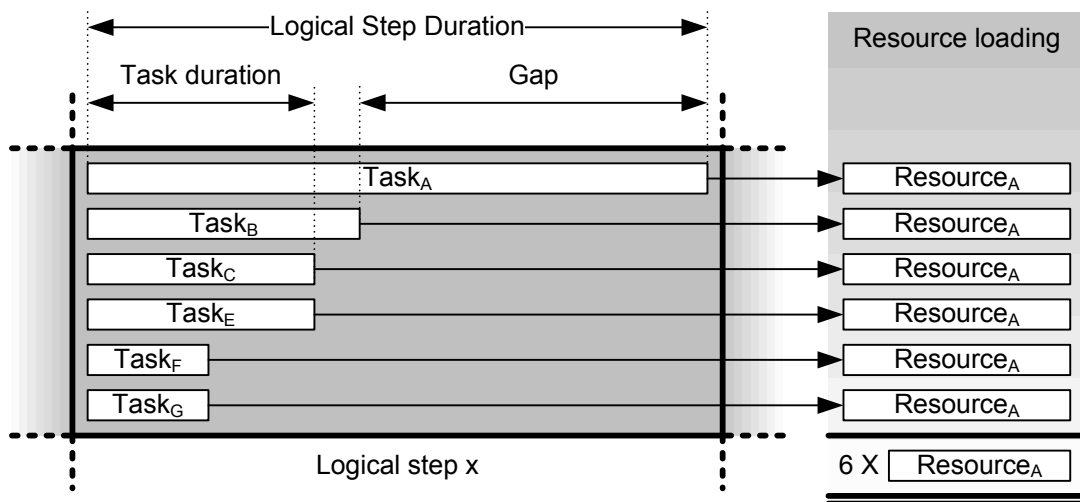


Figure 20: Resource loading of a logical step before *Task-Shuffling*

However, when resource constraints are taken into account the parallel execution of Tasks inside a logical step may cause resource conflicts. A resource conflict will be present in the logical step if the availability of Resource_A is less than six.

Task-Shuffling reduces the parallel execution of Tasks inside a logical step as much as possible by shifting Tasks into serialized positions within the logical step. By reducing the parallel execution of Tasks the resource loading is also reduced. In Figure 21 Task_C, Task_F and Task_G has been shifted into a serialized position. The resource loading of Resource_A is reduced to three.

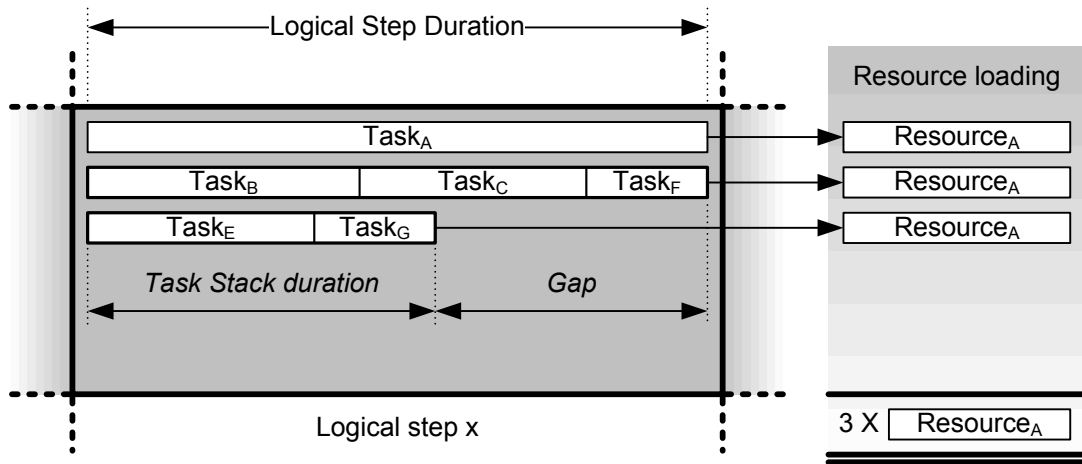


Figure 21: Resource loading of a logical step after *Task-Shuffling*

The effectiveness of *Task-Shuffling* is highly dependent on the spread of Task durations in the logical step. If the Tasks have approximately the same durations, i.e. a bad spread of durations like on the right hand side of Figure 22, then there are not sufficient gaps to shift Tasks into serialized positions. If the Task durations in a logical step have a good spread (see Figure 20), then *Task-Shuffling* will be effective (see Figure 21).

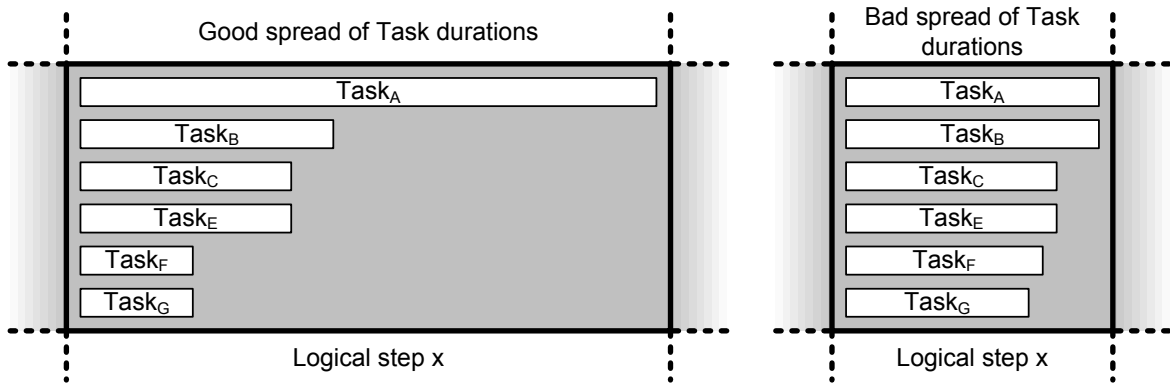


Figure 22: Spread of Task durations in a logical step

Task-Shuffling is effective in reducing overall resource loading within a logical step. However, the strategy is confined to the boundaries of each logical step. It is not a strategy that guarantees to resolve all resource conflicts, unless the length of steps is increased artificially to allow sufficient shuffling. However, artificially increasing step length is undesirable since it does not account for under-utilisation of resources that may exist in certain steps and the overall length of the project cannot be minimized. To achieve complete resource conflict resolution another strategy must be applied in conjunction with the *Task-Shuffling* strategy.

7.2 TABU-SEARCH: Global Task shifting

After *Task-Shuffling* is applied to the uCLSS there may be remaining resource conflicts. To resolve the remaining resource conflicts requires a more intrusive strategy. The only option is to delay Tasks by shifting Tasks to subsequent logical steps. This strategy is considered intrusive, because it makes structural changes to the uCST. Delaying Tasks reduces parallel execution of Tasks. If the right combination of Tasks is delayed all the resource conflicts are resolved. Tasks cannot be shifted to earlier logical steps, because the uCLSS procedure positions Tasks in the earliest possible logical step.

Choosing which combination of Tasks to shift and how many steps to shift them is not a trivial undertaking. It is a combinatorial problem with an explosive growth in complexity as the problem domain grows. First, *Task-Shuffling* is applied to the uCLSS and if resource conflicts remain a heuristic search technique is applied to the uCST to try and find the best combination of Tasks to delay in pursuit of an acceptable cCST and cCLSS.

Heuristic search techniques make it possible to find solutions with desired properties that are located in complex domains. The value of using heuristic search techniques becomes more apparent when the size of the solution domain increases. In many instances heuristics provide the opportunity to be able to find a solution where it was not possible using only closed deterministic techniques. Many heuristic search techniques have enjoyed extensive research. However, it is the *Tabu-Search* that proved to be the most compatible and easiest to map to the problem domain at hand.

7.2.1 TABU SEARCH: PATTERN DEFINITION

Tabu Search is a form of local neighbourhood search. The current solution (S) is used as a starting point for the search. Its local neighbourhood of solutions is explored and the best solution (S^*) is selected as the new current solution. S^* is reached from S by an operation called a *move*. *Moves* can be evaluated against certain predefined *metrics* to determine which *move* has the best value. The decision to choose the best *move* is based on the assumption that good *moves* have a higher probability to lead to optimal or near-optimal solutions. This does not necessarily mean that S^* is an improvement over S , but it is this feature that enables escaping from local optima.

The central concept behind the *Tabu Search*, and where it got its name from, is the *Tabu List*. Recent *moves* are registered in the *Tabu List* and it then becomes taboo to perform those registered moves again for a certain period. *Aspiration criteria* have to be satisfied for a *move* registered in the *Tabu List* to be performed. The *Tabu List* forces the *Tabu Search* into unexplored solution space and prevents it from circling back and getting trapped in local optima.

7.2.2 TABU-SEARCH: APPLYING THE PATTERN TO THE PROBLEM DOMAIN

The *Tabu Search* is a pattern and must be mapped to the problem domain at hand. Tasks are positioned in the earliest possible logical step in the uCLSS and using the uCST as a starting solution means that all resource constrained solutions will contain some subset of Tasks that are delayed from their initial step positions in the uCLSS. This fact provides a vital clue about how the *Tabu-Search* will operate in the specific domain of PLEP.

START SOLUTION:

Like many heuristic search techniques, the *Tabu-Search* requires a start solution to seed the search. A random starting solution may be used, but using a good starting solution reduces the solution space the *Tabu-Search* has to search to find an acceptable cCST. Using the uCST as a start solution allows the *Tab-Search* to ignore very large areas of the solution space.

CURRENT SOLUTION AND NEW SOLUTION:

When the *Tabu-Search* starts the current solution is the start solution. A new solution is reached after a *move* is performed in the current solution. Then the new solution becomes the current solution until another *move* is performed to reach yet another new solution.

STABLE SOLUTION:

When the *Tabu-Search* finds a solution that is desirable the solution is considered stable. The stable solution is saved as a possible cCST and the *Tabu-Search* resets back to the start solution (uCST). It can also occur that the *Tabu-Search* cannot find a desirable solution in which case the *Tabu-Search* resets back to the starting solution. The *Tabu List* is not reset when the *Tabu-Search* resets back to the start solution.

MOVE:

The only way to reach a new solution from the current solution is to delay a Task; or in other words to shift a Task across a logical step boundary to a subsequent step. A *move*, and thereby shifting a Task to a subsequent logical step, is realized by means of *edge insertion*. *Edge insertion* always occurs between Tasks in the same logical step. An edge is inserted from an *anchor-Task* to a *drifter-Task*. The *anchor-Task* is a Task least desirable to shift and the *drifter-Task* is the most desirable Task to shift. In Figure 23 Task_B is chosen as the *anchor* and Task_C as the *drifter*. S^* is then reached from S by inserting an edge from Task_B to Task_C. Topological sorting of S^* , i.e. computing the logical sequence of Tasks of S^* , reveals the impact of the *move*. Task_C is shifted from step 2 to step 3.

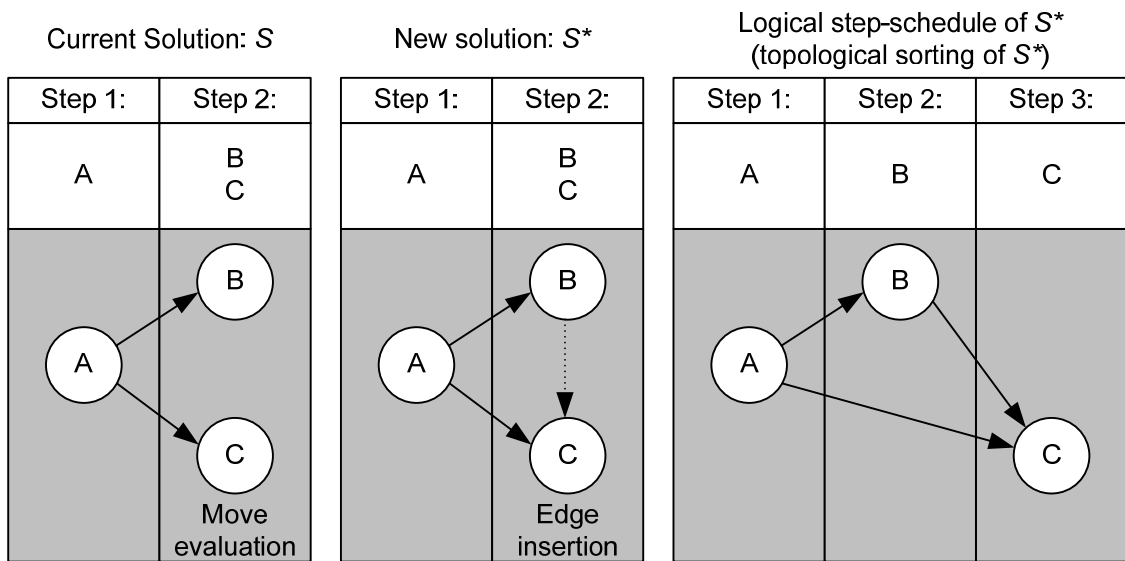


Figure 23: A move is realized by means of edge insertion

NEIGHBOURHOOD OF MOVES:

When the *Tabu-Search* is at S several moves may be available to reach an S^* . This collection of moves is known as the *neighbourhood of moves* of S . One *move* has to be chosen to reach S^* and the choice is based on *metrics*.

METRICS:

Predefined metrics are applied to the *neighbourhood moves* to choose the best *move* in terms of the metrics. There is only one *metric* at this stage that measures the slack of the *anchor-Task* and *drifter-Task* of each potential move in the neighbourhood. A *move* with an *Anchor-Task* that has no slack and a *drifter-Task* that has the most slack is preferred. In Figure 24 the *slack*

of each Task is shown. *Slack* is defined as the number of logical steps a Task can be delayed before the number of logical steps increase.

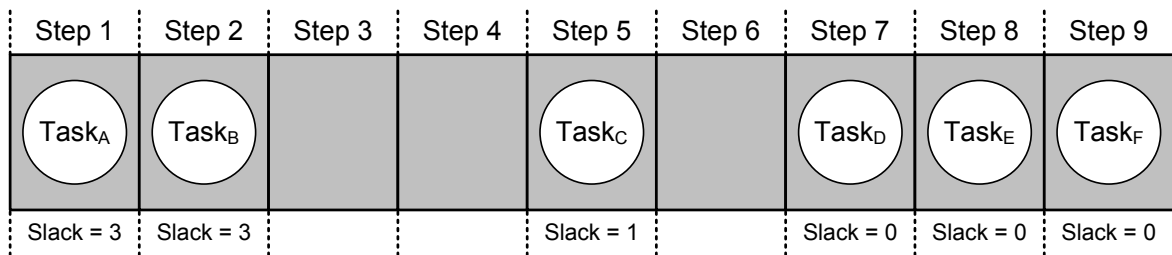


Figure 24: Slack

There are also predefined *metrics* for solutions. Although there is only one uCST there can be a large amount of cCSTs. The best cCST should be chosen from the available cCSTs and the cCST must also be compared to the uCST. Two *metrics* are defined for solutions. The one solution *metric* is used to define a stable solution and checks whether the solution has resource conflicts or not. If a solution has no resource conflicts it is considered stable. The other solution *metric* is the duration of the logical step schedule. Shorter solution durations are preferred.

TABU LIST:

When a *move* is performed it is placed in the *Tabu List*. The *Tabu List* acts as short-term memory. When a move is in the *Tabu List* it is considered taboo to perform the *move* again. Only when the *move* is released from the *Tabu List*, or if *aspiration criteria* allow it, can a taboo *move* be performed again. The function of the *Tabu List* is to force the *Tabu Search* into unexplored solution space.

ASPIRATION CRITERIA:

The traditional use of *aspiration criteria* is to allow a taboo *move* to be performed if it will result in a new solution that is better than the best solution. In this implementation *aspiration criteria* fulfil a slightly different role.

To reach a specific stable solution a unique combination of *moves* has to be performed. In Figure 25 it can be seen that to reach each stable solution requires a unique combination of *moves*. However, all eight stable solutions stem from the same first *move*. When stable solution #1 is reached the first *move* is already placed in the *Tabu List*. If the first *move* is

strictly blocked the remaining stable solutions will never be found by the *Tabu-Search*, because the *Tabu-Search* always resets back to the same start solution (uCST). Therefore an aspiration criteria is introduced that allow a *move* to be performed a certain number of times before it strictly becomes taboo. This is called the *Move Redo Limit (MRL)*.

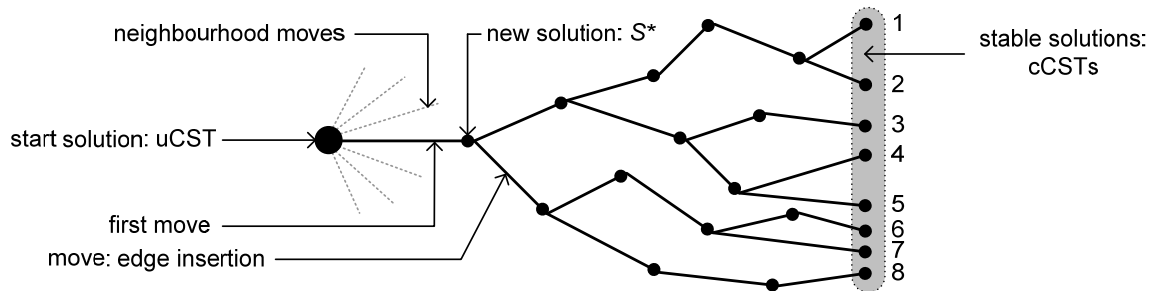


Figure 25: How the *Tabu Search* explores the solution space

7.2.3 FLOW OF THE TABU SEARCH ALGORITHM

In Figure 26 the flow of the *Tabu-Search* is shown. Each *Tabu-Search* iteration will consist of several *move* iterations. When a *Tabu-Search* iteration completes it produces a result. The result may be a cCST or a partially resource constrained consistent sequence of Tasks. When a cCST or partially resource constrained consistent sequence of *Tasks* is found the *Tabu-Search* resets back to the start solution (uCST). The number of *Tabu- Search* iterations must be specified and dictates how thoroughly the *Tabu-Search* will explore the solution space.

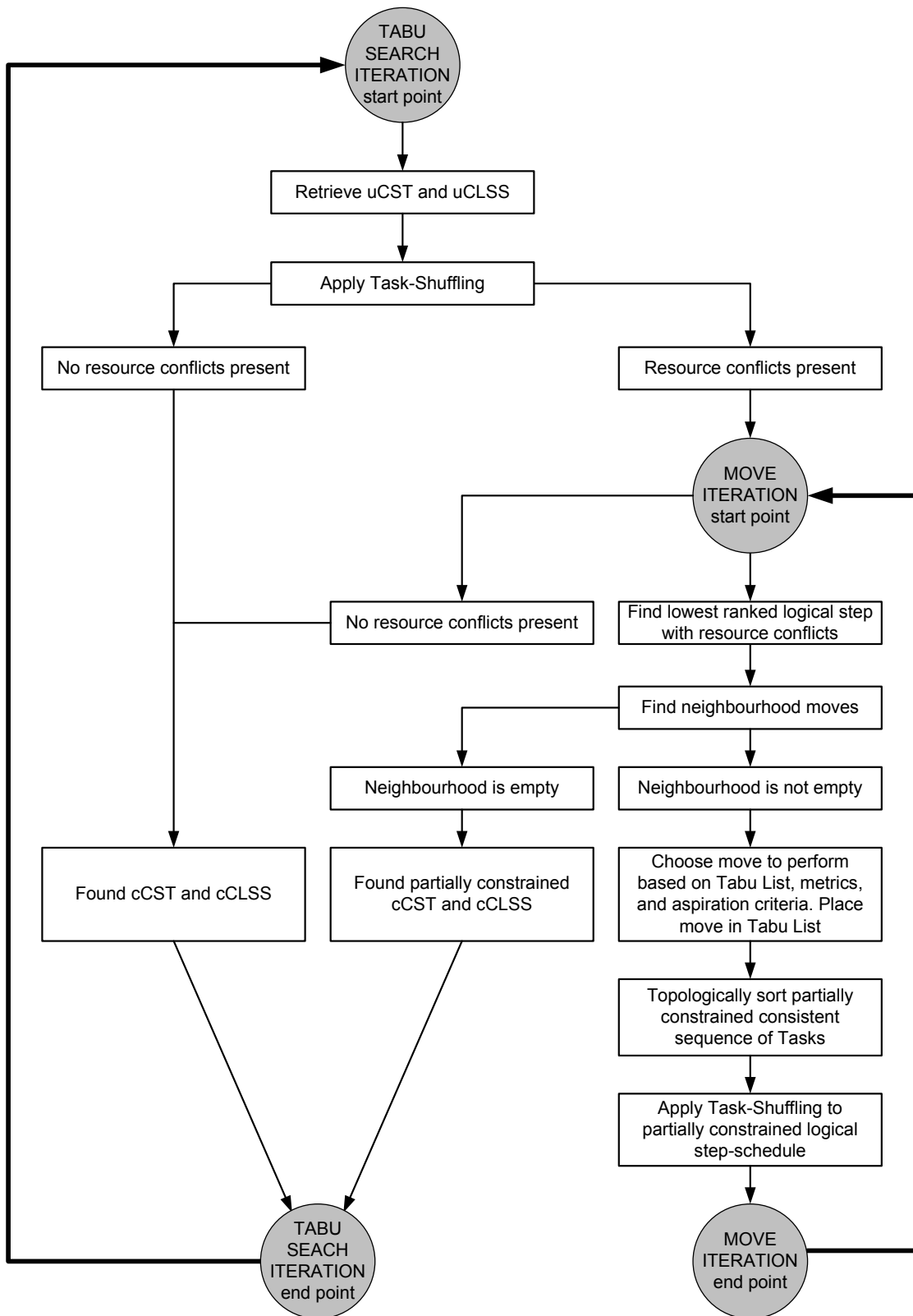


Figure 26: Tabu-Search flow diagram

8 IMPLEMENTATION

An object-oriented application is implemented using the Java programming language to test PLEP and resource constraint concepts. The main focus areas of the application are:

- Graphical User Interface
- Graph Library Toolkit
- Model Toolkit
- Resource Constraint Toolkit
- Persistence

Refer to APPENDIX C: SOURCE CODE AND OTHER MATERIAL for all implementation examples.

8.1 GRAPHICAL USER INTERFACE

A Graphical User Interface (GUI) is required to populate the model with the necessary components and relations. A complete model can be created and modified in the GUI. Limited GUI functionality is provided to display a logical step schedule in the form of a basic Gantt chart.

8.1.1 MAIN VIEW OF APPLICATION

The *MainFrame* class provides the entry point and main view of the application. From the menu of *MainFrame* new projects are created, existing projects are opened, open projects are saved, the application can be exited, and the uCST and cCST calculated. In the top half of *MainFrame* the Tasks, Persons, Tools and Deliverables are created, modified and removed as seen in Figure 27. These component lists can be sorted and filtered according to certain criteria. Popups are used to facilitate user input when these components are created or modified.

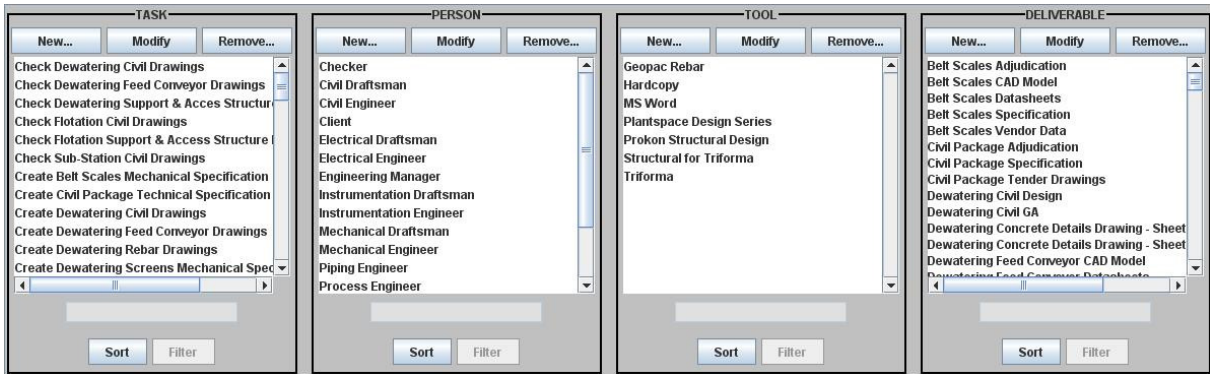


Figure 27: The GUI for specifying Tasks, Persons, Tools and Deliverables

In the bottom half of *MainFrame* the relations, Attributes, Statuses, and DEPs are created, modified and removed as well as the results viewed. The relationships tab has three sub-tabs, one for each user specified relation. The three relations share the same GUI layout of which Task-Deliverable GUI, as shown in Figure 28, is the most complex of the three. All three GUIs for relation specification have the concept of a current object. The current object is the start vertex of the relationship. In Figure 28 the current object is a Task. The current object is dragged from a component list, as seen in Figure 27, to the text field of the current object. Once there is a current object the target vertex for a relationship can be dragged in from an applicable component list. In Figure 28 three different types of Task-Deliverable relationships can be create depending on where the Deliverable is dropped. When a Create or Modify Task-Deliverable relationship is created the output Status must be specified.

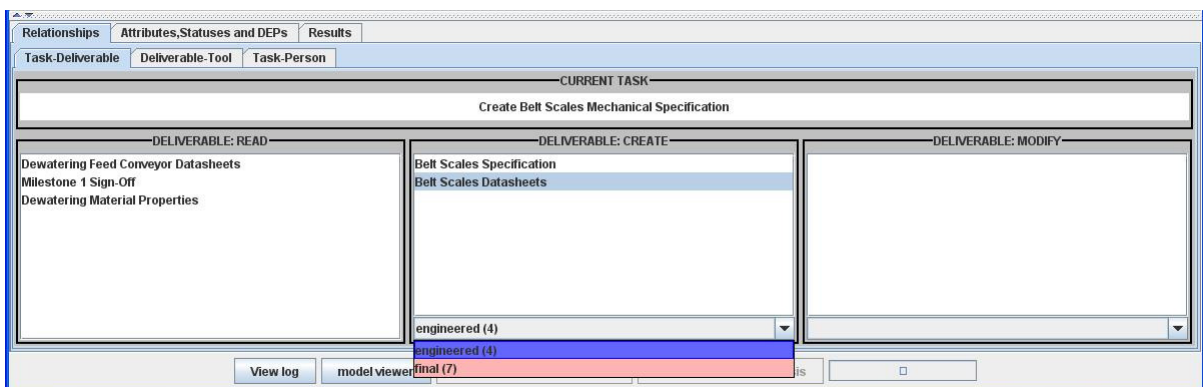


Figure 28: The GUI for specifying Task-Deliverable relationships

The Attributes, Statuses and DEPs are created, modified and removed in their specific tab as seen in Figure 29. Popups are used to facilitate user input when these components are created or modified.

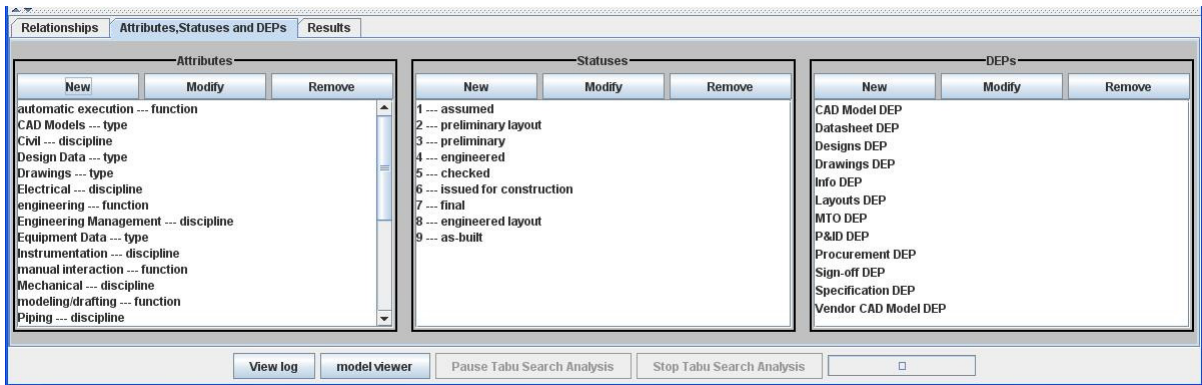


Figure 29: The GUI for specifying Attributes, Statuses, and DEPs

The logical step schedule tab display the uCLSS and cCLSS as a basic Gantt as seen in Figure 30. The steps that are highlighted in a pink colour contain resource conflicts. When a Task is selected its predecessors (green) and successors (red) are highlighted as shown in Figure 30.

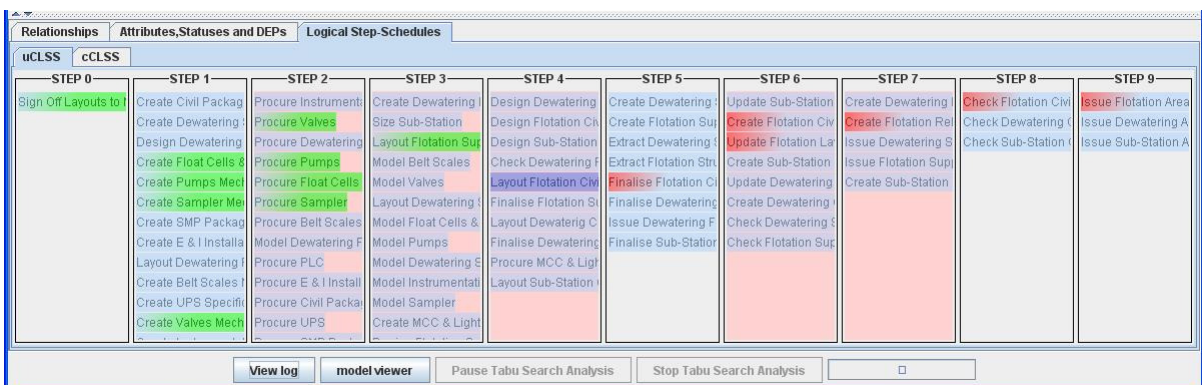


Figure 30: The GUI to view a basic Gantt-type chart of the uCLSS or cCLSS

8.2 GRAPH LIBRARY TOOLKIT

The Model Toolkit relies heavily on graph structures and algorithms and therefore a generic Graph Library Toolkit was required. The generic Graph Library Toolkit focuses on providing generic graph and algorithm implementations that can easily be extended. Algorithms are independent of concrete graph implementations. Therefore concrete graph implementations can be interchanged without affecting the algorithms.

8.2.1 GRAPHS

Directed graphs consist of vertices and edges that link the vertices. A vertex itself has no structural properties in terms of a graph. In other words, a vertex does not know about the

other vertices in the graph. What defines a vertex is the content of the vertex. Therefore an explicit *Vertex* class is not required. The classes¹ that define vertex content are used to model the vertices themselves. The structure of a graph is defined by its edges that link the vertices. The start and end vertex of an edge define its structural properties. An edge also has edge content² that contains information pertinent to the edge. Therefore an explicit *Edge* class is required to model the structural properties of an edge and contain the edge content. The type of the start- and end vertex as well as the type of the edge content is each set by their own generic [6] parameter.

A graph class is required to manage the vertices and edges. Several generic interfaces define how the Graph Library Toolkit is used. Efficient concrete implementations of these interfaces can be created to satisfy specific problem domains. All algorithms are coded against the graph interfaces. This renders the algorithms graph-implementation agnostic. The class hierarchy of the interfaces in Figure 31 make it possible to accurately specify the behaviour required from a graph. Algorithms are defined using only the essential graph interfaces, which increases the reusability of the algorithms.

The *IGraph* interface has no generic parameters. *IVertexListGraph* and *IVertexMutableGraph* have a generic parameter for the type of vertex content, while the remaining interfaces have an additional generic parameter for the type of edge content. The generic parameters enforce type safety and increase the reusability of the graph implementations.

The following interfaces are implemented:

- ***IGraph***: The *IGraph* interface provides the most basic graph functionality of which the most important is the ability to clear a graph of vertices and edges.
- ***IIncidenceGraph*<*TVertexContent*, *TEdgeContent*>**: The *IIncidenceGraph* interface refines the *IGraph* interface and adds the functionality to access the out-edges, out-degree, and all the successor vertices of a specific vertex.
- ***IBidirectionalGraph*<*TVertexContent*, *TEdgeContent* >**: This interface refines the *IIncidenceGraph* interface and adds the functionality to access the in-edges, in-degree, degree and all predecessor vertices of a specific vertex. Not all directed graphs require access to in- and out-edges and therefore this interface is separated from the

¹ *PlepTask*, *PlepDeliverable*, *PlepPerson* and *PlepTool* class

² *PlepTaskDeliverable*, *PlepTaskPerson*, *PlepDeliverableTool*, *PlepTaskTask*, *GenericEdgeContent* class

IncidenceGraph interface. Undirected graphs will only implement the *IncidenceGraph* interface above. Since the in-edges map to the out-edges they both return the edges incident to a vertex.

- ***IVertexListGraph*<*TVertexContent*>**: This interface refines the *IGraph* interface and adds the functionality to iterate over all the vertices in the graph.
- ***IEdgeListGraph*<*TVertexContent*, *TEdgeContent*>**: This interface refines the *IGraph* interface and adds the functionality to iterate over all the edges of the graph.
- ***IVertexAndEdgeListGraph*<*TVertexContent*, *TEdgeContent*>**: This interface combines the *IVertexListGraph* and *IEdgeListGraph* interfaces. No additional functionality is added.
- ***IVertexMutableGraph*<*TVertexContent*>**: This interface refines the *IGraph* interface and adds the functionality to add and remove vertices.
- ***IEdgeMutableGraph*<*TVertexContent*, *TEdgeContent*>**: This interface refines the *IVertexMutableGraph* interface and adds the functionality to add and remove edges.

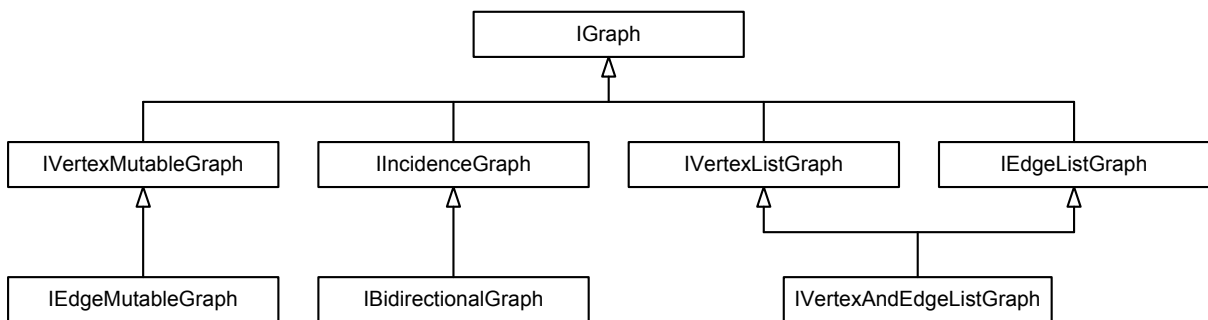


Figure 31: Graph interfaces

Only one concrete graph class, *Graph*<*TVertexContent*, *TEdgeContent*>, is implemented and it implements the full range of interfaces as shown in Figure 32. Therefore *Graph* provides access to all the graph functionality as described in the above mentioned list of interfaces. However, the scenario may present itself where an algorithm will not require all the functionality of the *Graph* class and places a high value on efficiency. For example, an algorithm requires a graph that only provides access to its vertices, but the efficiency of the graph is vitally important. Such an algorithm requires that the graph implement only the *IVertexListGraph* interface. A lightweight and efficient graph can be implemented that only implements the *IVertexListGraph* interface. This lightweight graph will be able to provide more efficient access to its vertices compared to the versatile *Graph* class.

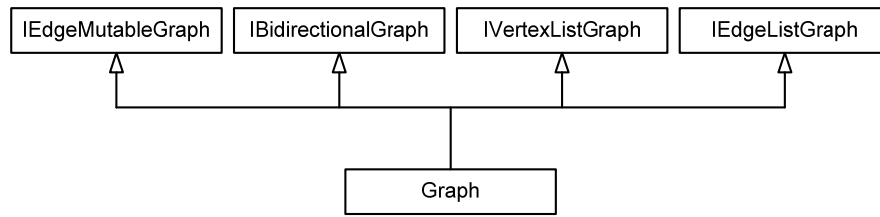


Figure 32: Actual graph implementation

Two other graphs are implemented, but they delegate all their functionality to a referenced *Graph* instance. These two delegate graph classes are *TransposedGraph* and *FilteredGraph*. The *TransposedGraph* provides a transposed view of a *Graph* instance. When a graph is transposed its edges are reversed. The *FilteredGraph* provides a filtered view of a *Graph* instance. Vertices and edges of the *Graph* instance are filtered according to specified vertex- and edge predicates.

8.2.2 ALGORITHMS

The algorithms are classified into two sets. The first set of algorithms is considered the core algorithms of the Graph Library Toolkit. The core algorithms by themselves do not compute meaningful information other than the traversal of the graph. The second set of algorithms is considered specialization algorithms and uses, as much as possible, the core algorithms as building blocks to implement more meaningful algorithms.

CORE ALGORITHMS

The core algorithms are traversal algorithms and comprise the Depth First Search (DFS) and Breadth First Search (BFS) algorithms. However, only the DFS algorithm is implemented. The core algorithms do not store any information when a graph is traversed; they publish events at key points during the traversal. Observers can register to receive the events to gather the data. This Visitor pattern provides a mechanism for extending the generic core traversal algorithms.

The DFS algorithm performs a depth-first traversal of the vertices in a directed graph. When possible, a depth-first traversal chooses a vertex adjacent to the current vertex to visit next. If all adjacent vertices have already been discovered, or there are no adjacent vertices, then the algorithm backtracks to the last vertex that had undiscovered neighbours. Once all reachable vertices have been visited, the algorithm selects from any remaining undiscovered vertices and continues the traversal. The algorithm finishes when all vertices have been visited.

Colour markers are used to keep track of which vertices have been discovered. White marks vertices that are yet to be discovered, gray marks a vertex that is discovered but still has vertices adjacent to it that are undiscovered. A black vertex is a discovered vertex that is not adjacent to any white vertices (its sub-tree traversal completed).

The order in which the DFS algorithm publishes traversal events can be seen in Figure 33. The first three slides are explained:

- **Slide 1:** All the vertices are initialized to white.
- **Slide 2:** Vertex u is randomly picked as the start vertex and the *Start Vertex* event is raised. Vertex u is then discovered and the *Discover Vertex* event is raised. Vertex u is marked as gray.
- **Slide 3:** Edge $u-v$ is examined and the *Examine Edge* event is raised. Edge $u-v$ is a tree edge and therefore the *Tree Edge* event is raised. Vertex v is discovered and the *Discover Vertex* event is raised. Vertex v is marked as gray.
- The procedure is continued until all the vertices in the graph are marked as black.

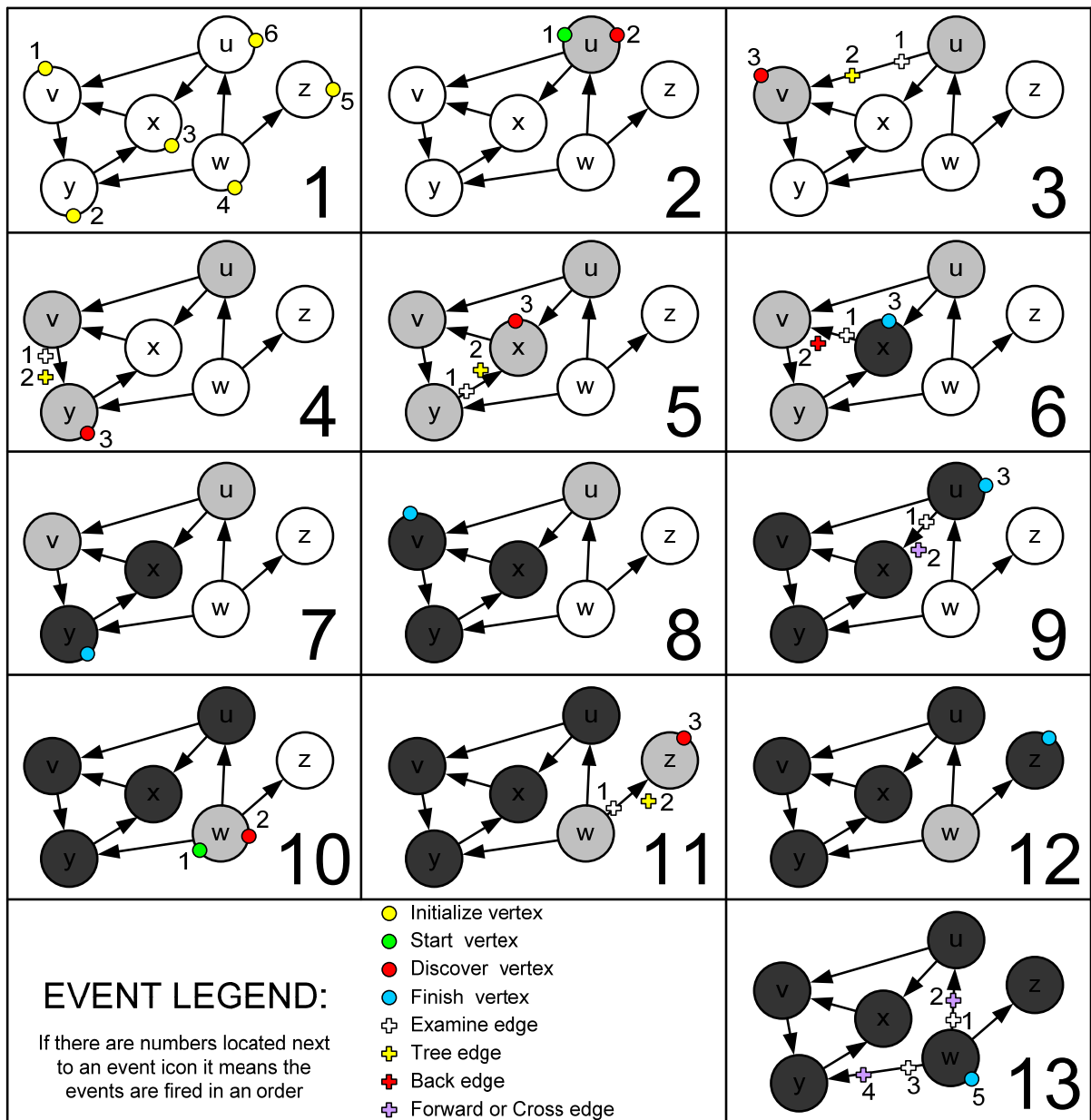


Figure 33: The DFS algorithm publishes events as it traverses the graph

SPECIALIZATION ALGORITHMS

The specialization algorithms use the core algorithms as building blocks to implement specialized meaningful algorithms. To achieve this, a specialization algorithm registers as an observer to the required events that a core algorithm publishes. When the events are raised the specialization algorithm uses the data to perform specialized actions and may even publish its own events. In this way specialization algorithms can use other specialization algorithms as building blocks.

For example, if the pre- and post-ordering of vertices are required by a specialization algorithm, the algorithm implementation registers to receive the *Discover Vertex* and *Finish Vertex* events of the DFS algorithm. During the depth-first traversal of the graph the *Discover Vertex* and *Finish Vertex* events are raised and the specialization algorithm is informed. When a *Discover Vertex* event is received the specialization algorithm assigns a running pre-order number to the discovered vertex. Similarly when a *Finish Vertex* event is raised the specialization algorithm assigns a running post-order number to the finished vertex. The pre- and post-ordering specialization algorithm can publish its own events when pre- and post-ordering are assigned to the vertices, and other specialization algorithms can register to receive these events in turn.

The described visitor pattern makes it easy to implement specialization algorithms by registering for the events the core algorithms publish during traversal. Two specialization algorithms are implemented, namely Kosaraju's algorithm (KS) and Topological Sort algorithm (TSA).

KOSARAJU'S ALGORITHM

Kosaraju's algorithm is used to find the Strongly Connected Components (SCCs) of a directed graph G . Refer to Section 5.1 for an explanation of a SCC. Kosaraju's algorithm is based on the fact that a transposed graph G^T will contain the same SCCs as the original graph G . The algorithm is performed as follows:

1. Compute G^T by using the *TransposedGraph* implementation (8.2.1 GRAPHS)
2. Perform a DFS(G^T) and record running post-ordering for each vertex when DFS(G^T) *Finish Vertex* event is raised.
3. Perform a DFS(G) starting from the highest post-ordered vertex. Remove vertex from post-order ranking when DFS(G) *Discover Vertex* event is raised.
4. Each tree found in DFS(G) forest in point 3 is a SCC. When a tree is discovered Kosaraju's Algorithm's *Discover Strongly Connected Component* event is raised.

Kosaraju's algorithm utilizes the DFS algorithm and its *Discover-* and *Finish Vertex* events to perform its specialized actions. Other implementations using Kosaraju's algorithm must register for its *Discover Strongly Connected Component* event to receive the SCCs.

TOPOLOGICAL SORT ALGORITHM

The Topological Sort algorithm is used to find the logical sequencing of the vertices in the directed graph. The Topological Sort algorithm can only be performed on acyclic directed graphs. Refer to Section 5.1 for how cycles are removed from cyclic graphs. Unfortunately the Topological Sort algorithm is not compatible with the DFS algorithm or BFS algorithm for traversal of graph G . Not all specialized algorithms will be able to use the standard DFS or BFS algorithms. Nevertheless the core traversal algorithms should be used where possible. The Topological Sort algorithm relies on the graph's incoming– and outgoing edges to perform its traversal and applies its own vertex colour markings. A vertex is coloured white if it has not been traversed and black if it has. The gray vertex colouring is not used. The algorithm is performed as follows:

1. Initialize vertices by marking them white.
2. Find root vertices: vertices with no incoming edges.
3. Check the unchecked root vertices. If an unchecked root vertex has incoming vertices that are marked white³, the unchecked root vertex is removed from the set of unchecked root vertices.
4. Checked root vertices are marked black. Topological Sort Algorithm's *Discover Vertex Group* event is raised.
5. Find the outgoing vertices. If there are outgoing vertices repeat step 3 with the outgoing vertices as unchecked root vertices. If there are no outgoing vertices the Topological Sort algorithm is completed.

Other algorithms that use the Topological Sort algorithm must register to its *Discover Vertex Group* event to receive the logical sequencing of the vertices.

8.3 MODEL TOOLKIT

All PLEP specific functionality and logic are implemented in the Model Toolkit. The Model Toolkit makes extensive use of the Graph Library Toolkit, but never exposes the Graph Library Toolkit through the software's public interface. The main responsibilities of the Model Toolkit are to provide the classes to model a PLEP process model, to provide a model that manages instances of these classes, to calculate the duration of Task, and to calculate the uCST.

³ This can happen if step 3 follows step 5 below.

8.3.1 CLASSES OF THE MODEL TOOLKIT

Several classes are necessary to model a PLEP process model as discussed in Chapter 4. A simplified class diagram of the main components and relationships are shown in Figure 34.

Below are short descriptions of the main components in the Model Toolkit:

- ***PlepObject***: Every *PlepObject* must have a name and a unique id.
- ***IAttributable* <<interface>>**: Provides the functionality to add and remove *PlepAttributes*.
- ***PlepAttribute***: *PlepAttributes* refines *PlepObject* and are used to tag *IAttributable* objects. *IAttributable* objects can be sorted and filtered according to their tags.
- ***PlepStatus*<*T*>**: *PlepStatus* refines *PlepObject* and has an order since Statuses must be globally ordered. The type of the order object is set by the generic parameter *T*. *T* must implement the *Comparable*<*T*> interface.
- ***PlepDeliverableEvolutionProfile*<*T*>**: A *PlepDeliverableEvolutionProfile* refines *PlepObject* and contains a subset of *PlepStatus*<*T*> instances. Within every *PlepDeliverableEvolutionProfile* each *PlepStatus* is mapped to a *Percentage of Completion* weight.
- ***PlepTask***: *PlepTask* refines *PlepObject*, but adds no extra functionality.
- ***PlepDeliverable*<*T*>**: *PlepDeliverable* refines *PlepObject* and has a comparable completion weight and a *PlepDeliverableEvolutionProfile*<*T*>.
- ***PlepResource***: *PlepResource* refines *PlepObject* and has availability.
- ***PlepPerson* and *PlepTool***: *PlepPerson* and *PlepTool* both refine *PlepObject* and add no extra functionality.

Below are short descriptions of the classes that model the content of relationships (edges):

- ***PlepTaskDeliverable*<*T*>**: *PlepTaskDeliverable* models the edge content of a Task-Deliverable relationship. *PlepTaskDeliverable* refines *PlepObject*. The *PlepTaskDeliverable* can be of type *Create*, *Modify* or *Read*. The *PlepTaskDeliverable* contains an assigned *PlepStatus*<*T*> if it is of type *Create* or *Modify*.
- ***PlepTaskPerson***: *PlepTaskPerson* models the edge content of a Task-Person relationship. *PlepTaskPerson* refines *PlepObject*, but adds no extra functionality.
- ***PlepDeliverableTool*<*T*>**: *PlepDeliverableTool* models the edge content of a Deliverable-Tool relationship. *PlepDeliverableTool* refines *PlepObject*, but adds no extra functionality.

- **PlepTaskTask:** *PlepTaskTask* models the edge content of a Task-Task relationship. *PlepTaskTask* refines *PlepObject*, but adds no extra functionality.

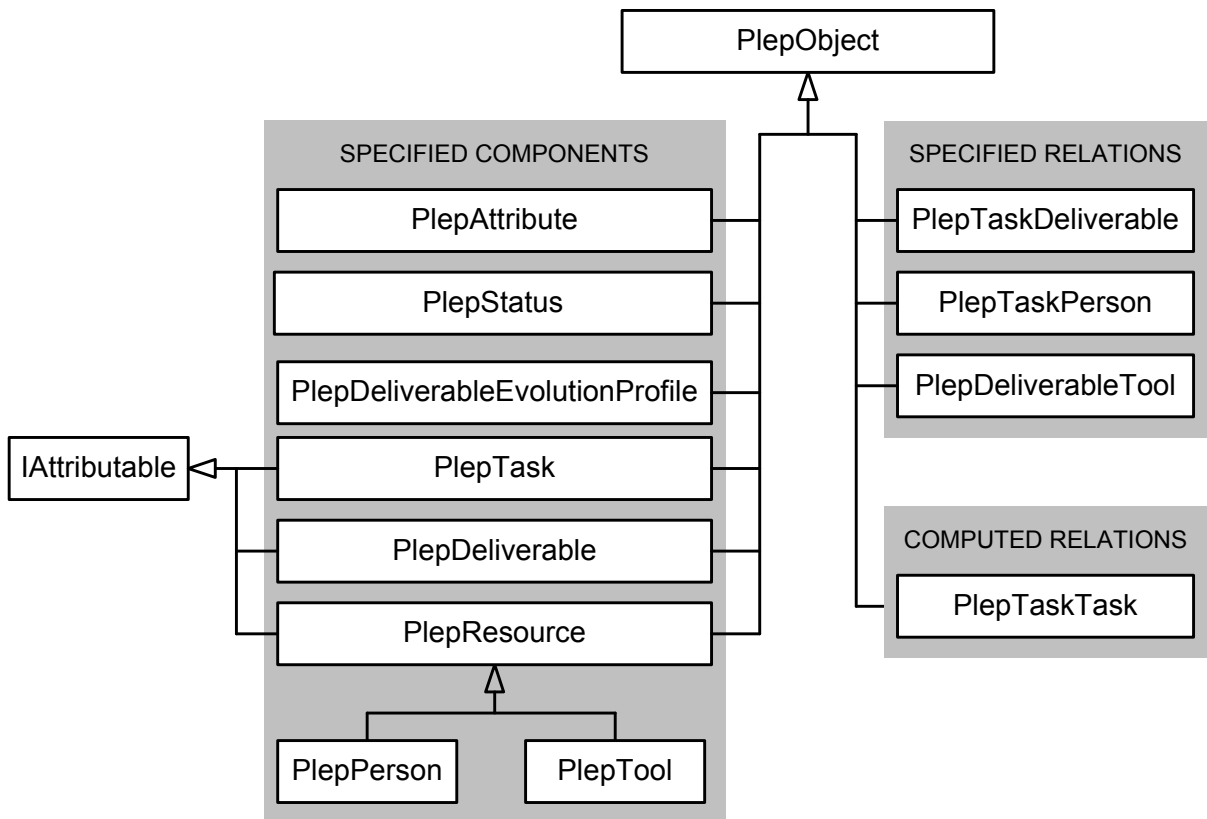


Figure 34: Simple class diagram of PLEP classes

All the above mentioned components and relations are stored in and managed by the *PlepModel* class. *PlepModel* is responsible for the functionality to add, remove and find components and relations. *PlepModel* is also responsible for calculating the duration of a Task and calculating and removing the uCST. The *PlepAttributes*, *PlepStatuses*, and *PlepDeliverableEvolutionProfiles* are stored in *HashSets*. The remaining components and relationships are stored in a *Graph* (See Figure 35).

PlepModel<T extends Comparable<T>>
attributeSet : HashSet<PlepAttribute> statusSet : HashSet<PlepStatus<T>> depSet : HashSet<PlepDeliverableEvolutionProfile<T>> plepGraph : Graph<PlepObject , PlepObject>
Adding, Removing, and Finding HashSet based components Adding, Removing, and Finding Graph based components Calculating duration of a Task Calculating and Removing the uCST

Figure 35: *PlepModel* class attributes and methods

8.3.2 ADDING, REMOVING AND FINDING COMPONENTS AND RELATIONSHIPS

Components are instantiated and added to a *PlepModel* using the *PlepModel's* *Add*-methods. Added components can be removed from a *PlepModel* using the *PlepModel's* *Remove*-methods. For example, to add and remove a Task from a model:

```

PlepModel<Integer> model = new PlepModel<Integer>( );
PlepTask task1 = new PlepTask( "design foundation reinforcement" );
model.addTask( task1 ); // Task is added to model
model.removeTask( task1 ); // Task is removed from model

```

To find components their name or id attributes are used as input to *Find*-methods in *PlepModel*:

```

PlepModel<Integer> model = new PlepModel<Integer>( );
...
// Many components are added to the model:
PlepTask task1 = new PlepTask( "design foundation reinforcement", "xyz123" )
...
PlepTask task1 = model.findTask( "design foundation reinforcement" );
// OR
PlepTask task1 = model.findTask( "xyz123" );

```

Relationships are modelled with *Edge* class objects and are composed of a start vertex, end vertex and an edge data. The start- and end vertex defines the structural properties of the edge and the edge data defines the content of the edge. The start vertex, end vertex and edge data are instantiated and added to the model using specific *Add*-methods. Once the relationships are added they can be removed. For example, to add and remove a Task-Person relationship from a model:

```

PlepModel<Integer> model = new PlepModel<Integer>( );
// Instantiate and add Task to model:
PlepTask task1 = new PlepTask( "design foundation reinforcement" );
model.addTask( task1 ); // Task is added to model
// Instantiate and add Person to model:
PlepPerson person1 = new PlepPerson( "engineer" );
model.addPerson( person1 );
// Create Task-Person edge data:
PlepTaskPerson taskPerson1 = new PlepTaskPerson( "taskPerson1" );
// Add Task-Person relationship to model:
model.addTaskPerson( task1, person1, taskPerson1 );
// Remove Task-Person relationship from model:
model.removeTaskPerson( task1, person1 );

```

The *Add*-, *Remove*, and *Find*-methods for components and relationships delegate to *HashSet* or *Graph* instances, depending on where the objects references are stored. *PlepModel* must also provide the functionality to find more intricate information as stated below:

- **Statuses:**
 - Find the assigned Status to a specific Task-Deliverable relationship:
model.findAssignedStatus(PlepTask task, PlepDeliverable deliverable)
 - Find all assigned Statuses for a specific Deliverable.
- **Deliverables:**
 - Find all Deliverables adjacent to a specific Task where the Task-Deliverable relationships are of type *Create*, and/or *Modify*, and/or *Read*.
- **Persons:**
 - Find all Persons adjacent to a specific Task.

- **Tools:**
 - Find all Tools adjacent to a specific Deliverable.
 - Find all Tools adjacent to Deliverables that are adjacent to a specific Task.
 - Find all Tools adjacent to Deliverable that is adjacent to a specific Task where the Task-Deliverable relationships are of a type *Create*, and/or *Modify*, and/or *Read*.

8.3.3 CALCULATING TASK DURATION

A Task does not reference a *PlepModel* and therefore cannot determine the references of Deliverables it is operating on. *PlepModel* is responsible to calculate the duration of a Task. *Read* Task-Deliverable relationships do not contribute the duration of its incident Task. Only the output Task-Deliverable relationships contribute to the duration of its incident Task.

Each output Task-Deliverable relationship has an assigned Status that is sourced from the DEP of the incident Deliverable. In the DEP the assigned Status is mapped to a *Percentage of Completion*. The *Percentage of Completion* is multiplied with the *Completion weight* of the incident Deliverable and represents the duration contribution of the Task-Deliverable relationship to its incident Task. There may be several output Task-Deliverable relationships incident to a Task. Their contributions to the duration of the Task are summed to determine the total duration of the Task.

In Figure 36 the Task is operating on three Deliverables. Each output Task-Deliverable relationship's contribution to the duration of the Task is:

- Task-Deliverable_A: $60\% \times 400 = 240$
- Task-Deliverable_B: $100\% \times 1300 = 1300$
- Task-Deliverable_C: $20\% \times 2000 = 400$

The duration of the Task equals 1940, which is the sum of all the Task-Deliverables' contributions.

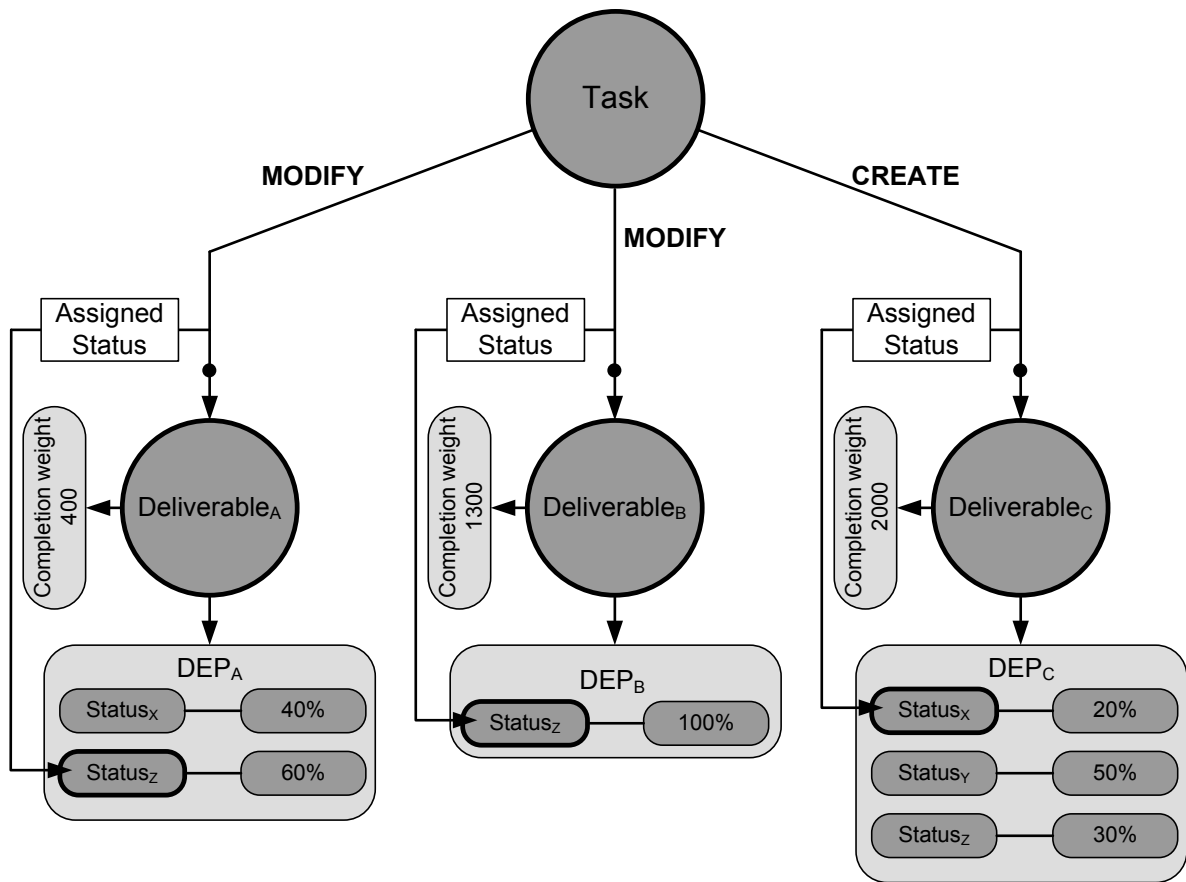


Figure 36: Calculating the duration of a Task

8.3.4 CALCULATING AND REMOVING THE uCST

PlepModel uses the *RuleMachine* class to calculate the uCST. The three predefined rules, as discussed in Section 4.3.2, are implemented in the *checkRules()* method. The *RuleMachine* publishes an event when a Task-Task relationship is found. *PlepModel* is registered to this event and when the event is raised *PlepModel* adds the Task-Task relationship that was found to itself. Before an uCST is calculated *PlepModel* removes all Task-Task relationships.

RuleMachine<T extends Comparable<T>>
ruleListeners : ArrayList<ITaskTaskRelationshipFoundListener>
model : PlepModel<T>
addRuleListener (listener : ITaskTaskRelationshipFoundListener)
checkRules ()

Figure 37: *RuleMachine* class attributes and methods

8.4 RESOURCE CONSTRAINT TOOLKIT

In Chapter 7 all the concepts involved with resolving resource constraints in the context of a logical step schedule are explained. Looking from a high-level perspective functionality for a logical step schedule, *Task-Shuffling* and *Tabu-Search* must be implemented.

8.4.1 LOGICAL STEP SCHEDULE

A logical step schedule implementation must be capable of determining its resource loading, calculate its duration, and perform *Task-Shuffling*. Four classes are implemented to model a logical step schedule as seen in Figure 39.

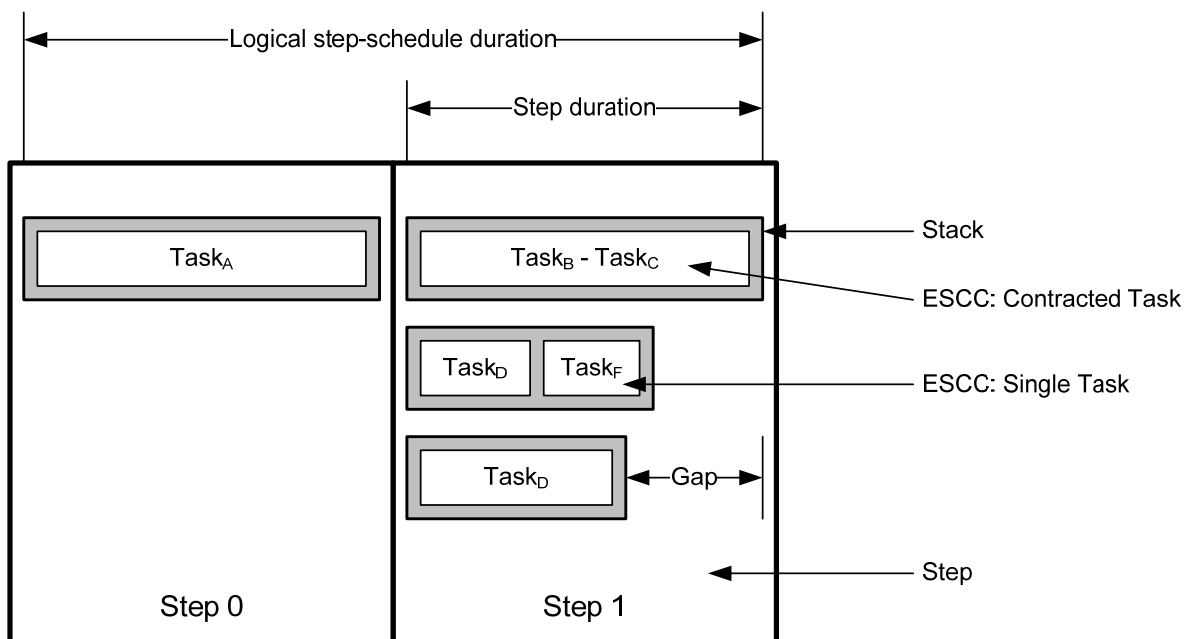


Figure 38: The main components of the logical step schedule

EnhancedStronglyConnectedComponent (ESCC) class

This is a wrapper class for the *graphToolKit.StronglyConnectedComponent* class and adds additional PLEP specific functionality. The ESCC class can calculate the summed duration, Person loading, and Tool loading of its internal Tasks.

duration() : double

Iterate over the internal Tasks and sum their durations.

personLoading() : HashSet<PlepPerson>

Iterate over the internal Tasks and find all the Persons utilised by each Task with the *modelToolKit.PlepModel.findPersons(PlepTask task)* method. The assumption was

made that a Task can load a Person only once⁴. Therefore the result is returned in a set, because a set does not allow duplicate entries.

toolLoading() : HashSet<PlepTool>

Iterate over the internal Tasks and find all the Tools utilised by each Task with the *modelToolKit.PlepModel.findTools(PlepTask task)* method. The assumption was made that a Task can load a Tool only once⁵. Therefore the result is returned in a set, because a set does not allow duplicate entries.

Stack class

The *Stack* class is modelled on the concept of a bucket. A *Stack* is composed of one or more *ESCCs*. Before Task-Shuffling is performed all *Stacks* are composed of one *ESCC* and after Task-Shuffling is performed a *Stack* may be composed of several *ESCCs*. The *Stack* class can calculate the summed duration, Person loading, and Tool loading of its internal *ESCCs*.

duration() : double

Iterate over the internal *ESCCs* and sum their durations.

personLoading() : HashSet<PlepPerson>

Iterate over the internal *ESCCs* and find all the Persons utilised by each *ESCC* with the *ESCC.personLoading()* method. The *ESCCs* are sequenced in series and therefore each *ESCC* can load a Person only once. Therefore the result is returned in a set, because a set does not allow duplicate entries.

toolLoading() : HashSet<PlepTool>

Iterate over the internal *ESCCs* and find all the Tools utilised by each *ESCC* with the *ESCC.toolLoading()* method. The *ESCCs* are sequenced in series and therefore each *ESCC* can load a Person only once. Therefore the result is returned in a set, because a set does not allow duplicate entries.

Step class

The *Step* class is composed of one or more *Stack* instances. A *Step* can calculate its duration, determine if it contains resource conflicts, calculate its Person and Tool loading, and perform Task-Shuffling.

duration() : double

The duration of a *Step* is dictated by the *ESCC* with the longest duration.

⁴ See Section 4.2.2 and Section 6.1.2

⁵ See Section 6.1.2

personLoading() : ArrayList<PlepPerson

Iterate over the internal *Stacks* and sum their Person loadings. A *Step* can load a Person more than once and therefore the result is returned as a list of Persons.

toolLoading() : ArrayList<PlepTool

Iterate over the internal *Stacks* and sum their Tool loadings. A *Step* can load a Tool more than once and therefore the result is returned as a list of Tools.

hasResourceConflicts() : boolean

The method returns *true* if:

- The frequency of each Person from the *personLoading()* result exceed the availability of the Person.
- The frequency of each Tool from the *toolLoading()* result exceed the availability of the Tool.

shuffle() : void

Refer to Section 7.1.

LogicalStepSchedule class

This class is composed of one or more *Steps*. It can calculate its duration, determine if it contains resource conflicts, perform Task-Shuffling and update itself.

duration() : double

Iterate over the *Steps* and sum their durations.

hasResourceConflicts() : boolean

Iterate over *Steps* and call their *hasResourceConflicts()* method. If one *Step* returns true then the *LogicalStepSchedule* contains resource conflicts.

shuffle() : void

Iterate over *Steps* and call their *shuffle()* method.

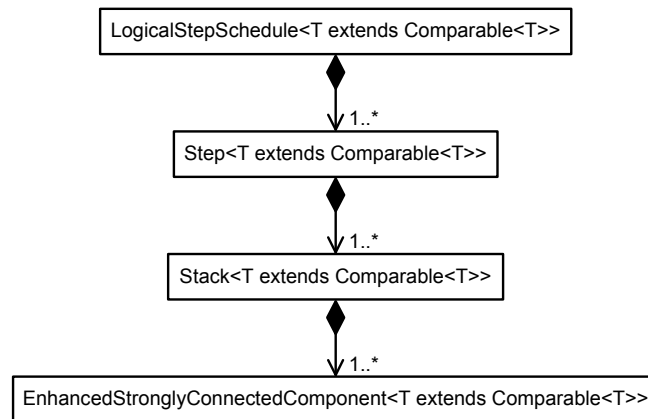


Figure 39: Required classes to model logical step schedule

8.4.2 TASK-SHUFFLING ALGORITHM

Task-Shuffling is implemented inside the *Step* class. It is a simple algorithm that attempts to fit as many of the *Step's ESCCs* into each *Stack*. The algorithm starts with a list of the *Step's ESCCs*. The list is sorted according to descending duration. A new *Stack* is created and the sorted list is traversed in descending order. Each *ESCC* is attempted to be inserted into the *Stack*. If an *ESCC* is inserted into the *Stack* the *ESCC* is removed from the sorted list. When all the *ESCC* are traversed a new *Stack* is created and the sorted list of remaining *ESCCs* is traversed again adding *ESCCs* into the new *Stack*. This process is repeated until the sorted list is empty. The *Stacks* are then filled with *ESCCs* and *Task-Shuffling* is completed.

8.4.3 TABU-SEARCH

The *TabuSearchAlgorithm* class performs a *Tabu-Search* on a given *PlepModel*. The start solution is the uCST and its corresponding uCLSS is used to find resource conflicts, slack, and neighbourhood of moves. The flow diagram of the *Tabu-Search* algorithm in Figure 26 maps exactly to the implementation of the algorithm. Some elements of the *TabuSearchAlgorithm* class are discussed below:

Tabu List:

The *Tabu List* is implemented as a mapping to incorporate the Move Redo Limit (MRL). *Moves* are mapped to the number of times the *move* has been performed. A *move* cannot be performed more than the MRL.

Move:

The *TabuMove* class models the actual *move* that was performed. A *TabuMove* has an *anchor-* and *drifter-Task*.

Neighbourhood of moves:

The Task pairs produced by the Cartesian product of the Tasks in a logical step constitute the *neighbourhood of moves*. Pairs containing the same Task are dropped since an edge cannot be inserted between a Task and itself.

Move Selection Strategy:

There are four strategies how a move is selected from the neighbourhood of moves:

- Random: A move is randomly picked from the neighbourhood.
- AnchorOrdered: A move is picked with the least amount of *slack* on the *anchor*.
- DrifterOrdered: A move is picked with the most amount of *slack* on the *drifter*.
- TotalOrdered: A move is picked with the least amount of *slack* on the *anchor* and the most amount of *slack* on the *drifter*.

Iteration limit and Analysis limit:

Each time a cCLSS is found a *Tabu-Search Iteration* has been completed. The number of *Tabu-Search Iterations* to perform is set by the *setIterationLimit(int limit)* method. The whole *Tabu-Search Analysis* can be repeated by the *setAnalysisIterationLimit(int limit)* method.

Performing a *Tabu-Search* is computationally expensive and cannot be executed in the same thread as the GUI since it will make the GUI unresponsive. Therefore the *Tabu-Search* algorithm is executed in its own thread. The GUI registers to numerous events that the *TabuSearchAlgorithm* class publishes. For instance, when a new best cCST and cCLSS is found the GUI is informed and the cCLSS is displayed. The results are also logged.

8.5 PERSISTENCE

For any application it is vital that the data is persisted in a robust format. In the case of PLEP it was desired that the persisted data is humanly readable. Therefore, serialization is automatically excluded. Another drawback of serialization is that it is very sensitive to code changes that can invalidate serialized data.

All data are persisted in XML format. Although XML is more verbose than serialization and comma-separated format, it is more robust against implementation changes. XML can easily be shared between different programming languages and is humanly readable.

Refer to *aecXML.pdf* in APPENDIX C: SOURCE CODE AND OTHER MATERIAL for an example of the XML format used in PLEP included on CD.

9 RESULTS

Two projects were modelled using the prototype PLEP application. The first project is a small fictitious AEC project and the second project is a larger real life AEC project sourced from an industry partner.

Four types of results were produced for each PLEP process model. Firstly, resource unconstrained results were produced. Secondly, *Task-Shuffling* was performed by itself. Thirdly, *Tabu-Search* was performed by itself, and fourthly, *Task-Shuffling* and *Tabu-Search* were performed in conjunction with each other. A better understanding of the behaviour of the strategies is possible if they are performed separately and in conjunction and the results compared.

For both process models the following *Tabu-Search* settings were used:

- *Tabu-Search* limited to 1000 iterations which means that 1000 cCSTs will be searched for.
- *Move* selection strategy set to Random. This means that *anchor-* and *drifter-Tasks* are picked randomly.
- The MRL was set to 300.

9.1 SMALL AEC PROJECT: FICTITIOUS PROJECT

The small fictitious AEC project is based on the typical design phase of a small scale concrete framed building. The user input of the traditional process model is compared to the PLEP process model in

Table 2. In Table 3 the results of the four analyses are shown.

Figure 40 shows the best cCLSS durations found at a specific *Tabu-Search* iteration.

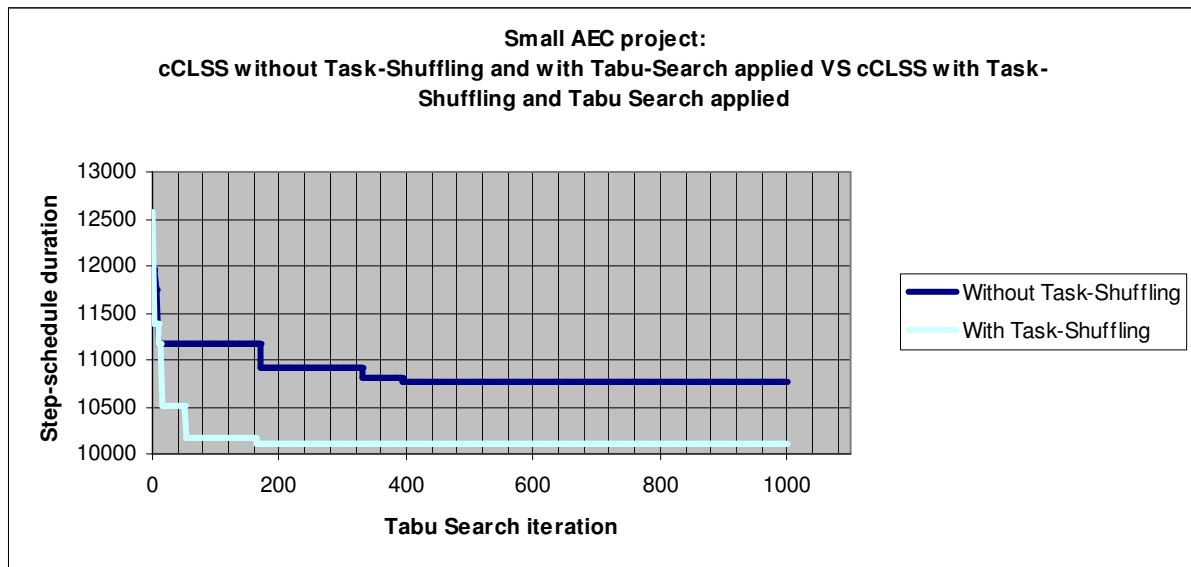
Table 2: Number of user specified components and relationships for small AEC project

		Traditional process modelling	PLEP process modelling
User specified components	Tasks	22	22
	Deliverables	n/a	22
	Persons	6	6
	Tools	4	4
	Statuses	n/a	4
	DEPs	n/a	3
		32	47
User specified relationships	Task-Deliverable	n/a	33
	Task-Person	27	27
	Deliverable-Tool	n/a	11
	Task-Tool	42	n/a (derived)
	Task-Task	35	n/a (derived)
		104	71
TOTAL		136	118

Table 3: Results of a small AEC project PLEP process model

	uCST	Task-Shuffling	Tabu-Search	Task-Shuffling And Tabu-Search
Steps	6	6	14	10
Duration [time unit]	7460	7460	10760	10035
Number of resource conflicts	23	6	0	0
Logical steps with resource conflicts	4	3	0	0
Number of moves	n/a	n/a	19	10
Average number of moves	n/a	n/a	19	14
Analysis duration [s]	< 1	< 1	144.15	153.77
Average time per move [ms]	n/a	n/a	7	10
Average time per cCLSS [ms]	n/a	n/a	144	153

Figure 40: Plots the best cCLSS found at each Tabu Search Iteration for the Small AEC project



9.2 LARGE AEC PROJECT: INDUSTRY PARTNER

The large AEC project was sourced from an industry partner in South Africa. The industry partner modelled the engineering planning phase of a real-life project using the prototype PLEP application. The user input of the traditional process model is compared to the PLEP process model in

Table 4. In Table 5 the results of the four analyses are shown. Figure 41 shows the best cCLSS durations found at a specific *Tabu-Search* iteration. In Figure 42 the distribution of cCLSS durations are shown for the case where *Task-Shuffling* and *Tabu-Search* are performed together.

Table 4: Number of user specified components and relationships for large AEC project

		Traditional process modelling	PLEP process modelling
User specified components	Tasks	79	79
	Deliverables	n/a	135
	Persons	17	17
	Tools	7	7
	Statuses	n/a	9
	DEPs	n/a	12
		103	259
User specified relationships	Task-Deliverable	n/a	414
	Task-Person	84	84
	Deliverable-Tool	n/a	137
	Task-Tool	181	n/a (derived)
	Task-Task	189	n/a (derived)
		454	635
TOTAL		557	894

Table 5: Results of a large AEC project PLEP process model

	uCST	Task-Shuffling	Tabu-Search	Task-Shuffling and Tabu-Search
Steps	10	10	27	12
Duration [time unit]	9747	9747	11980	10228
Number of resource conflicts	85	14	0	0
Logical steps with resource conflicts	9	5	0	0
Number of moves	n/a	n/a	48	10
Average number of moves	n/a	n/a	57	11
Analysis duration [s]	< 1	< 1	3652.74	937.52
Average time per move [ms]	n/a	n/a	63	82
Average time per cCLSS [ms]	n/a	n/a	3652	937

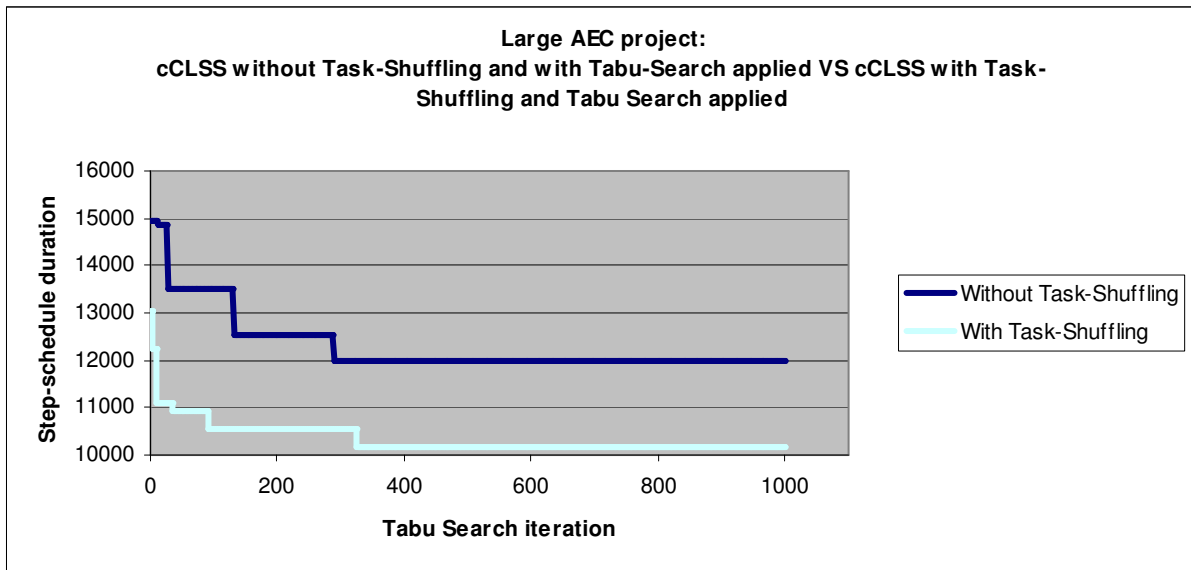


Figure 41: Plots the best cCLSS found at each Tabu Search Iteration for the Large AEC project

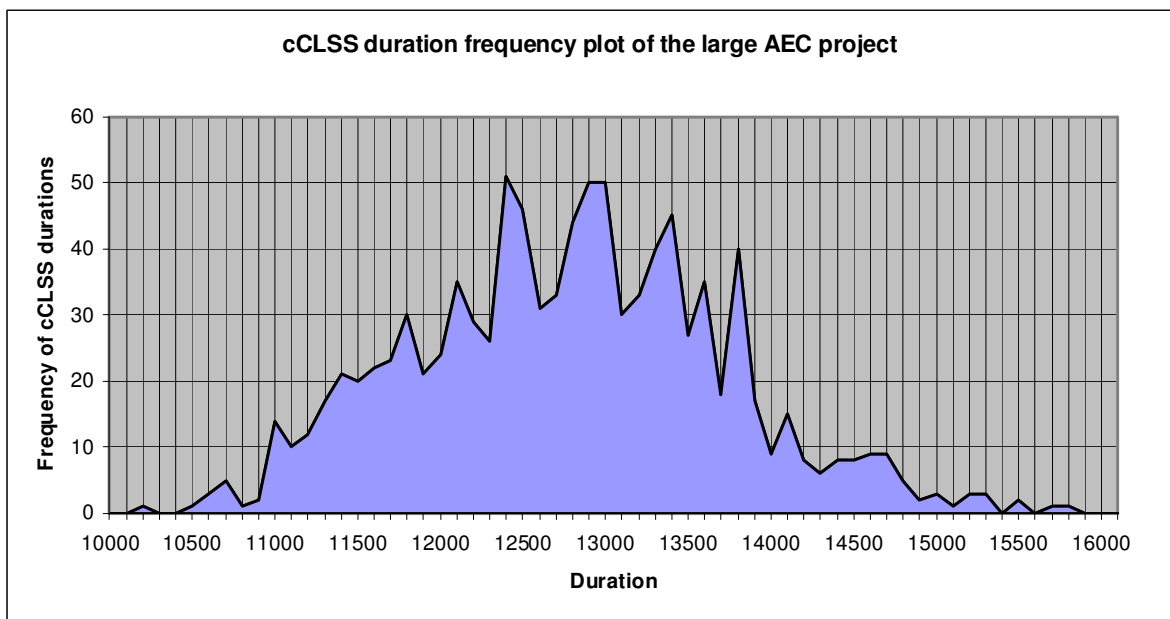


Figure 42: cCLSS duration frequency plot of the large AEC project

9.3 INTERPRETATION OF RESULTS

The two PLEP process models produced excellent results and the application behaved as expected. The small AEC project is rather small and some of the results may be misleading. Even so, the results are consistent and can be fully explained. The results of the large process model were as expected. When speculating about how the algorithms and application will scale, these results should be extrapolated.

9.3.1 LOGICAL STEP SCHEDULES

In APPENDIX A: Small AEC project (fictitious project) and APPENDIX B: Large AEC project (industry partner) the uCLSS for both projects are presented and is correct in terms of consistent Deliverable evolution. The situation where a Task is delayed because it requires input Deliverables that are not ready will never arise if the Tasks are executed as sequenced in the uCLSS. The uCLSS is optimal in terms of the number of logical steps and the execution rank of the Tasks. It is not possible to place any Tasks in an earlier logical step without causing a Deliverable evolution conflict.

Therefore, compared to the uCLSS all cCLSSs will contain equal or more logical steps and all Tasks will have an equal or higher execution rank. The cCLSSs in APPENDIX A: Small AEC project (fictitious project) and APPENDIX B: Large AEC project (industry partner) confirm this statement.

9.3.2 DURATIONS

Although the uCLSS is optimal in terms of the number of logical steps and execution ranks of Tasks it is not optimal in terms of duration. It is possible to produce a cCLSS with a shorter duration than the uCLSS as shown in a small example in Figure 43. However, this scenario does not occur often. The PLEP application could not find cCLSSs with shorter durations compared to the uCLSS.

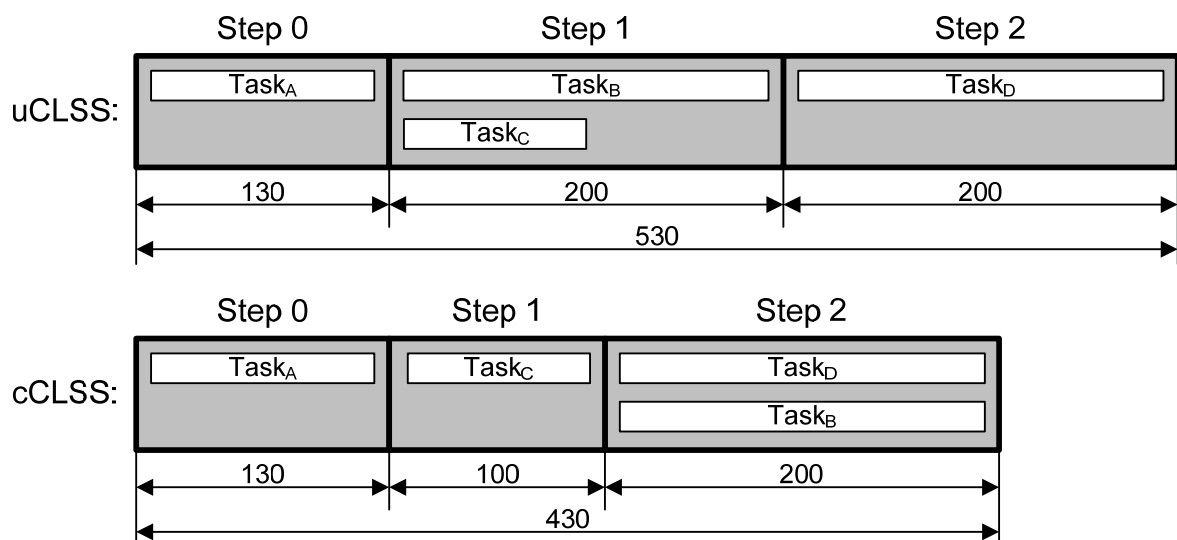


Figure 43: A cCLSS with a shorter duration than the uCLSS

Task-Shuffling is a non-intrusive resource loading reduction strategy. It moves the Tasks around inside a logical step much like playing a puzzle and it has no effect on the duration of the logical step. The duration of a logical step is fixed by the Task with the longest duration inside the logical step. *Task-Shuffling* never artificially increases the duration of steps. Therefore, performing *Task-Shuffling* by itself has no effect on the duration of a logical step schedule.

The *Tabu-Search* is an intrusive resource conflict resolution strategy. It shifts Tasks over the boundaries of logical steps and therefore has the power to alter the durations of logical steps. If the Task with the longest duration is shifted away from a logical step, the duration of the step will decrease to the Task that had the second longest duration. Similarly, if a Task is shifted into a logical step and has a duration longer than the duration of the step, the duration of the step will increase. The *Tabu-Search* always increases the execution rank of Tasks that are shifted. Therefore, as the *Tabu-Search* shifts Tasks in its quest to resolve all resource conflicts, the logical step schedule will get drawn out. In other words, the logical step schedule will become serialized. Usually this leads to cCLSSs with longer durations compared to the uCLSS as seen in Table 3 and Table 5. In Figure 43 it was shown that it is theoretically possible to produce a cCLSS with a shorter duration than an uCLSS, but that seldom happens.

Performing *Task-Shuffling* and *Tabu-Search* together produce cCLSSs with shorter durations compared with when *Tabu-Search* was performed by itself. *Task-Shuffling* reduces the amount of resource conflicts the *Tabu-Search* must resolve and therefore fewer Tasks have to be shifted. Since fewer Tasks have to be shifted the cCLSSs are less serialized, leading to shorter overall durations.

9.3.3 RESOURCE CONFLICTS

Given its simplicity, *Task-Shuffling* was surprisingly efficient. *Task-Shuffling* cannot produce inferior results due to its non-intrusive nature and therefore should always be performed. Unfortunately *Task-Shuffling* cannot guarantee the resolution of all resource conflicts. *Task-Shuffling* reduces resource loading and is not focused upon resolving all the resource conflicts.

The *Tabu-Search* strategy is aggressive and focuses exclusively on resolving resource conflicts. Tasks can never generate a resource loading of more than one for a specific resource. Therefore, if the availability of all resources is at least one, it is guaranteed that the *Tabu-Search* will find a cCLSS.

9.3.4 NUMBER OF MOVES

Task-Shuffling reduces the number of resource conflicts *Tabu-Search* must resolve. Therefore it is expected that performing *Tabu-Search* together with *Task-Shuffling* will require less *moves* compared to when *Tabu-Search* is performed by itself.

9.3.5 AVERAGE TIME PER MOVE

It may seem like an anomaly that the average time spent on performing a *move* is longer when *Tabu-Search* and *Task-Shuffling* is performed together compared to when *Tabu-Search* is performed by itself. The reason for this behaviour stems from the fact that when *Tabu-Search* and *Task-Shuffling* is performed together the Tasks are less serialized. In other words there are more Tasks per logical step. More Tasks per logical step leads to a larger *neighbourhood of moves*, which in turn leads to more *moves* that must be evaluated to pick which *move* to perform. Therefore the average time spent on performing a *move* is longer when *Tabu-Search* is performing in conjunction with *Task-Shuffling*.

9.3.6 AVERAGE TIME PER cCLSS

Tabu-Search is computationally much more expensive than *Task-Shuffling*. Therefore it is expected that the average time spent to find a cCLSS is longer when *Tabu-Search* is performed by itself compared to when the *Tabu-Search* is perform in conjunction with *Task-Shuffling*, because *Tabu-Search* must resolve more resource conflicts.

In the small AEC project this was not the case (see Table 3). When *Tabu-Search* was performed in conjunction with *Task-Shuffling* it took on average more time to reach a cCLSS compared to when *Tabu-Search* was performed by itself. This is due to the small size of that project. The overhead of performing *Task-Shuffling* is relatively large compared to the total overhead of reaching a cCLSS.

In the large AEC project the average time to reach a cCLSS performing *Tabu-Search* in conjunction with *Task-Shuffling* was significantly shorter compared to when *Tabu-Search* was performed by itself (see Table 5). *Task-Shuffling* reduced the amount of resource conflicts the computationally expensive *Tabu-Search* had to resolve and this lead to an overall reduced computation time to reach a cCLSS.

9.3.7 EFFECTIVE SEARCHING OF SOLUTION SPACE BY TABU SEACH

It is important to confirm that *Tabu-Search* is performing a thorough search of the solution space. Figure 42 shows the frequency plot of the durations of cCLSSs found when *Tabu-Search* was performed in conjunction with *Task-Shuffling* on the large AEC project. The durations of the cCLSSs found, shows a good distribution. Interestingly the mean cCLSS duration is 13266 and the median cCLSS duration is 12740. Therefore the chances of finding a cCLSS with a shorter duration are more than finding a cCLSS with a longer duration.

Tabu-Search finds good cCLSSs rather quickly (

Figure 40 and Figure 41). In both analyses *Tabu-Search* found its best cCLSS in approximately the first twenty to forty percent of total *Tabu-Search Iterations*. In larger projects where it may be computationally expensive to search for cCLSSs the *Tabu-Search* may be stopped after the initial surge of good cCLSSs has been found.

9.3.8 USER INPUT

User input effort can be measured in several ways of which the amount of user input is one. The small AEC project is probably too small to deliver conclusive results, but the large AEC project delivered expected results.

A PLEP process model will always require more component user input, because in addition to the traditional process model's components the Deliverables, Statuses, and DEPs must be specified.

It is not clear whether a PLEP process model will require more or less user input than an AEC traditional process model. This depends mainly on the size of the Task-Deliverable, Deliverable-Tool, Task-Tool, and Task-Task relations. The PLEP process model will require more user input in terms of relations if the Task-Deliverable and Deliverable-Tool relations are larger than the Task-Tool and Task-Task relation.

In the case of the small AEC project the traditional process model required more relational user input, but in the case of the large AEC project the PLEP process model required more relational user input.

When measuring whether a PLEP process model requires more relational user input than a traditional AEC process model the complexity as well as the size of the relations should be compared. As discussed in this study the specification of the Task-Task relation is complex and inconsistent and should be avoided. Also, the number of Task-Task relationships that must be specified for the large AEC project, listed in

Table 4 as being 189 for the large project, is somewhat misleading. The listed number is the actual number of “has to be executed before” relationships discovered by the PLEP process model. However, a traditional project planner would have had no way of knowing these beforehand. He would have had to *consider* and *evaluate* a much larger number of possible Task-Task relationships, thereby increasing the specification effort significantly.

10 CONCLUSIONS AND RECOMMENDATIONS

The main problem identified with traditional process modelling is that it requires the user to directly manipulate the Task-Task relation which is complex and may become inconsistent. PLEP process modelling introduces the concepts of Deliverables and the Task-Deliverable relation and avoids the direct manipulation of the Task-Task relation. The user specifies the input and output Deliverables for each Task after which the Task-Task relation is derived from the specified information. Deriving the Task-Task relation from the Task-Deliverable relation is less complex and its consistency is guaranteed.

This study had a two-fold focus. First a usable PLEP application was implemented to create and maintain AEC process models, produce measurable results, and be used as a research platform. Initially the application could only produce the uCST and uCLSS and although the uCST and uCLSS are valuable information it did not incorporate the impact of resource constraints.

Secondly, resource constrained results were produced using the uCST and uCLSS as a starting point. The impact of resource constraints on the uCST and uCLSS was investigated and a solution was implemented on top of the PLEP application.

In this chapter conclusions will be drawn on PLEP, the prototype application implementation, and the results. Recommendations will also be made as to what research may be conducted in the future.

10.1 RESOURCE UNCONSTRAINED RESULTS

Traditional scheduling focuses on Tasks and does not model the impact Deliverables have on the logical sequence of Tasks. However, it is the Deliverables that are the most important components of a process model, because the Deliverables are the actual products of a project. Tasks are merely the entities that must be performed in a specific sequence to produce the Deliverables at the required status. Therefore, in a PLEP process model the Deliverables and the Task-Deliverable relation are the core information from which the Task-Task relation is derived. Deliverable-focused process modelling requires the participants to have a clear, but

nevertheless simple understanding of the underlying AEC process. This improves the quality and thoroughness of a PLEP process model. The meaning of each derived Task-Task relationship is apparent, because it can be traced back to the Task-Deliverable relationships responsible for its existence. Whereas in traditional process modelling there is no way to gain access to the meaning of a user specified Task-Task relationship, other than if the users documented their intent; which seldom happens in real projects.

PLEP process modelling requires the user to input more information compared to traditional process modelling, but the information is bundled into smaller packages that are considerably simpler. These small packages of input information also make it possible for project participants to work on the process model collaboratively. For instance, each participant knows best what Tasks they have to perform and the required input and output Deliverables. Therefore each person can specify their own Tasks, Deliverables and Task-Deliverable relationships. The input of the participants is combined and a logical sequence of Tasks for the whole project is derived using PLEP. When participants add or modify any information an updated logical sequence of Tasks can instantly be derived. This is in stark contrast with traditional AEC process modelling which is expensive, time consuming, and specialists are required to create and maintain the model.

The logical sequence of Tasks, also called the uCST in this study, produced by PLEP does not incorporate resource constraints. Although it is important to model the impact of resource constraints it does not render the uCST worthless. On the contrary, the capability of deriving the uCST is what sets PLEP apart from traditional process modelling. An array of powerful project management applications is available that require as input the logical sequence of Tasks. Instead of feeding these applications with traditional user specified logical sequence of Tasks the PLEP derived uCST can be used. A conclusion can be drawn that PLEP is an excellent tool for producing the uCST and is superior in terms of model consistency and reusability compared to a traditional user specified logical sequence of Tasks.

10.2 RESOURCE CONSTRAINED RESULTS

The main goal of this study was to use the PLEP framework and prototype application to produce a cCST and cCLSS from the uCST and uCLSS. The resource constrained results were produced in the domain of logical step schedules and although logical step schedules do

not reflect real-life scenarios as good as time-based schedules it still provided valuable insight into resource constrained modelling and scheduling. The knowledge gained from this study can be transferred to time-based schedules without any fundamental changes.

Concluding from the results the two strategies, *Task-Shuffling* and *Tabu-Search*, proved to be very successful in solving resource conflicts in the context of logical step schedules.

10.3 RECOMMENDATIONS

The focus of this study was limited due to time constraints, but there is considerable scope for future PLEP research. The current version of the PLEP framework and prototype application can be improved and several topics are obvious candidates for future research.

10.3.1 IMPROVING THE CURRENT VERSION OF PLEP

Several assumptions were made to simplify the problem domain and process model. Strictly speaking these assumptions are sound, but introduce inefficiencies to the algorithms and subsequent results. Focusing on these assumptions will improve the current version of PLEP and not incur large changes to the prototype application.

EFFORT-DRIVEN RESOURCE ALLOCATION

The assumption that only the responsibility of performing a Task is assigned by the Task-Person relationship means that assigning more Persons to a Task will not reduce the duration of the Task. It is recommended to allow effort driven resource allocation, because the concept of resource levelling can be better modelled and Tasks can be “rushed”. Resource levelling is the process of resolving resource conflict and also ensuring balanced use of resources. Too little resources are considered a resource conflict and this situation was dealt with in this thesis. However, having too many resources in certain parts of a schedule is wasteful. In these instances it is beneficial if idle resources can be assigned to Tasks.

GENERIC RESOURCE ALLOCATION

Currently a specific Person can only be assigned to a specific Task once. If a Task requires ten engineers then ten different engineers must be modelled and each one

assigned to this Task. It is recommended that generic resources are introduced to PLEP. Generic resources model the role of a Person, but not a specific instance of a Person. If a Task requires ten engineers only one generic engineering Person can be assigned to the Task. If the availability of the generic engineering Person is less than ten a resource conflict will be present. At a later stage generic resources may be replaced with specific resource instances, but the specific resource instances must be able to perform the role of the generic resources it replaces.

VARIABLE AVAILABILITY OF RESOURCES

The assumption that resources have a constant availability during the execution of the project is a by-product of using logical step schedules as well as that it simplifies algorithms concerning resource availability. Variable resource availability should be introduced when time-based schedules instead of logical step schedules are used. Each resource can have its own timeline with variable availability.

TASK DURATION: A TASK OPERATES ON ADJACENT DELIVERABLES IN PARALLEL, SERIES AND A COMBINATION OF THE TWO

The duration of Task is calculated using the incident output Task-Deliverable relationships. It is assumed that a Task will always operate on its adjacent Deliverables in series, because additional Person-Deliverable information is required to be able to determine if a Task can operate on any of its adjacent Deliverables in parallel. This causes Tasks to be assigned maximal durations which is obviously not efficient.

It is recommended that when a Task operates on more than one Deliverable the necessary Person-Deliverable information is specified, because this will enable PLEP to determine which Deliverables the adjacent Task can operate on in parallel.

CONTRACTED TASKS ARE PERFORMED IN SERIES

The natural assumption is that all the Tasks in a cycle must be performed in parallel, because the cyclic Tasks depend on each other's input and output Deliverables. However, the incident Task-Person relationships determine whether cyclic Tasks can be performed in parallel, series or a combination of the two. For example, if the same Person with an availability of one is assigned to all the Tasks in a cycle it is easy to

realize that the cyclic Tasks must be performed in series, because the Person can only work on one Task at a time. To avoid this complexity a conservative assumption was made that Tasks in a cycle are always performed in series and therefore the duration of a contracted Task is always the sum of the cyclic Tasks. It is recommended that the Task-Person relationships are taken into account in a more detailed manner and that cyclic Tasks that can be performed in parallel are modelled as such.

10.3.2 EXTENDING PLEP

There are several topics that are good candidates for future research. Some topics reside within PLEP while other topics will latch on to the concepts of Deliverables.

SCALABILITY

Very large real-life AEC process models can contain more than ten thousand Tasks and such large process models will expose any inefficiency with the input of the process model, keeping the model up to date, and the algorithms. The next step will be for an industry partner to thoroughly test the prototype application to see if the concepts of PLEP and the application scales well. The heuristic algorithms will require much more research effort to increase their efficiency, but it is probably the GUI that will demand the most attention.

GRAPHICAL USER INTERFACE

It was difficult to visualize and explore the process model, even though the investigated process models were relatively small. Real-life AEC projects can contain over 10,000 Tasks and it would be inefficient to view such large process models as lists and Gantt charts. It would be possible to focus on a small part of the process model, but then its contextual setting is lost. If the whole process model is viewed it is not possible to focus on specific areas. It is important to be able to have a focused and contextual view of the process model at the same time. A FOCUS+CONTEXT view [7] as seen in Figure 44 should be used to explore and interact with the AEC process model.

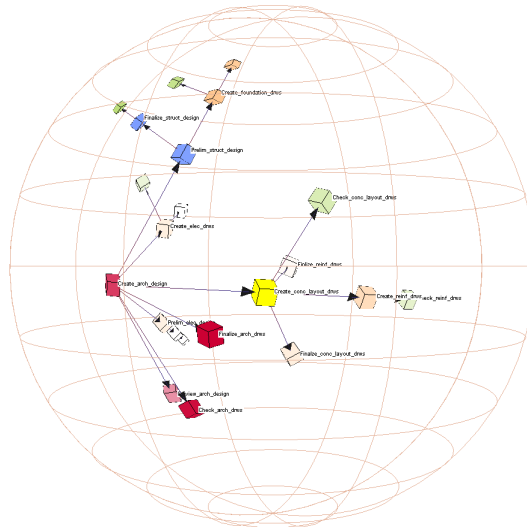


Figure 44: Hyperbolic graph view of an AEC process model

USER ASSISTANCE

PLEP reduces the complexity of specifying and maintaining AEC process models compared to traditional methods, but it remains a difficult job due to the sheer volume of information and the graph environment associated with all the relationships. It should be easy to navigate the process model graph, find errors, and see the impact of certain user acts. For example, if a Status is to be removed the user should be able to see which DEPs and output Task-Deliverable relationships are using this Status.

Reusing information will decrease specification effort. Process models of similar AEC projects will contain many components and relationships that are also similar. For example, *Structural Steel Design* projects will share many of the same Statuses, DEPs, Tasks, Deliverables, Persons, and Tools. It may be speculated that many of the obvious relationships are shared as well. The common information can easily be extracted into templates and the templates reused between projects of similar nature.

Deliverable Breakdown Structures (DBS) and Work Breakdown Structures (WBS) will enable collaborative specification of Deliverables and Tasks, ensure that no overlapping Deliverables and Tasks are specified, ensure that Tasks are specified at a constant level of detail, and the hierarchical information in the WBS and DBS can be utilized in “rolling-up” Tasks and Deliverables. It is recommended that DBS and WBS are investigated and introduced to the PLEP framework.

INCONSISTENT MODEL STATE

When a PLEP process model is specified not all the information may be readily available. In those cases it will be beneficial to leave the model in an inconsistent state until the necessary information is available. For example, a Task modifies a Deliverable, but the output Status is not known. This Task-Deliverables relationship is then entered into the PLEP process model without an assigned output Status, but it will always be highlighted in some way to remind the user that this relationship is in an inconsistent state. If a process model is allowed to be in an inconsistent state it is very important that the causes of the inconsistencies can easily be located and corrected when the required information is available.

TIME-BASED PLEP AND RESOURCE CONSTRAINT FUNCTIONALITY

PLEP is only concerned with logic between components. Time is not incorporated into PLEP. This study focused on introducing resource constraints on top of the PLEP framework and prototype application, but in the domain of logical step schedules. Time-based schedules add the extra dimension of time to a logical schedule and are more complex to model. It is recommended that PLEP and the work of this study are transferred to a time-based framework which is a better reflection of reality and will be more useful in real life. *Task-Shuffling* will not be applicable in a time-based schedule and the *Tabu-Search* algorithm will require more and higher quality metrics to effectively steer the search in a less constrained domain.

Several excellent time-based scheduling applications are readily available and the question may arise why PLEP should not simply export its results in a format these applications can interpret. This certainly is a valid point, but unfortunately no readily available time-based scheduling application incorporates the concept of Deliverables as part of their actual scheduling routines as PLEP does. If the results of PLEP are exported to these readily available applications a lot of valuable information is thrown away. Therefore, it seems be worth the effort to research and implement a time-based PLEP framework and application.

FOLLOW-UP SYSTEMS

Deliverables and how they are used in PLEP is a powerful concept that can be used in many follow-up systems. For example, an Earned Value Management system requires an up-to-date planned schedule to measure progress against. Obviously PLEP provides an excellent planned schedule (although logic-based at this point in time), but PLEP also provides the Deliverables and Statuses against which progress can easily be measured.

10.4 FINAL REMARKS

This study looked at the theory of PLEP, introduced resource constraints, implemented a prototype application, and produced good results. Several conclusions can be drawn from this study, but most importantly it can be concluded that excellent resource constrained results were produced which was the main aim of this study. Another important conclusion is that PLEP is a definite improvement over traditional scheduling methods and therefore it makes sense to implement resource constraint functionality into the PLEP framework. Many improvements are required to elevate the quality of PLEP to the level of a real-life usable application, but the usefulness of PLEP has already been proved.

APPENDIX A: Small AEC project (fictitious project)

Step 0	Step 1	Step 2	Step 3	Step 4	Step 5
A	B C D	E [F,G,H] I J	K [L,M,N,O,P,Q]	R [S,T,U]	V

Figure A.1: Small AEC project's uCLSS with Task-Shuffling applied

Step 0	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7	Step 8	Step 9	Step 10	Step 11	Step 12	Step 13
A	B C	E F	D G	L	H T	N O	I J	K	P S	Q R	V	M	U

Figure A.2: Small AEC project's cCLSS without Task-Shuffling and with Tabu-Search applied

Step 0	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7	Step 8	Step 9
A	B D	I J	K [C,Q]	E [F,S,P]	O H	M [N,G]	L U	R T	V

Figure A.3: Small AEC project's cCLSS with Task-Shuffling and Tabu-Search applied

- A. Create arch design
- B. Review arch design
- C. Prelim struct design
- D. Prelim elec design
- E. Finalize struct design
- F. Finalize arch drws
- G. Create foundation drws
- H. Create elec drws
- I. Finalize elec design
- J. Create conc layout drws
- K. Finalize conc layout drws
- L. Finalize foundation drws
- M. Finalize elec drws
- N. Check struct design
- O. Check arch drws
- P. Create reinf drws
- Q. Check elec design
- R. Finalize reinf drws
- S. Check conc layout drws
- T. Check foundation drws
- U. Check elec drws
- V. Check reinf drws

See electronic PLEP model for details **APPENDIX C: SOURCE CODE AND OTHER MATERIAL.**

APPENDIX B: Large AEC project (industry partner)

Step 0	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7	Step 8	Step 9
A1	B1 [C1,D1,E1,F1,G1,H1,I1,J1,K1,L1,M1,N1,O1]	P1 [Q1,R1,S1,T1,U1,V1] [W1,X1,Y1,Z1,A2,B2]	C2 [D2,E2,F2] O2 P2	Q2 [R2,S2] [T2,U2] [V2,W2] [X2,Y2,Z2]	A3 [B3,C3,D3] [E3,F3,G3,H3]	I3 J3 [K3,L3] M3 [N3,O3] P3	Q3 R3 [S3,T3,U3]	V3 [W3,X3]	Y3 [Z3,A4]

Figure B.1: Large AEC project's uCLSS with Task-Shuffling applied

Step 0	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7	Step 8	Step 9	Step 10	Step 11	Step 12	Step 13	Step 14	Step 15	Step 16	Step 17	Step 18	Step 19	Step 20	Step 21	Step 22	Step 23	Step 24	Step 25	Step 26
A1	F1 I1	S1 N1	H1 K2	B1 J1 E2	D1 Z1	W1 M1 L1	P1 C2 X1	T2 L2	G3 V1 G1	U1 F2 Y1	M2 K1 O1	C1 A2	R1 E1	T1 J2	D2 O1 I2	G2 O2 F2	Q2 U2 E2 Z2 N2	R2 V2 H2 H3	E3 X2 W2 D3	A3 B3 I3 J3 K3 F3	M3 R3 N3 F3 C3	Q3 V3 O3 T3 L3	W3 S3 U3	Y3 Z3	X3	A4

Figure B.2: Large AEC project's cCLSS without Task-Shuffling and with Tabu-Search applied

Step 0	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7	Step 8	Step 9	Step 10	Step 11
A1	B1 [C1,D1,F1,E1,G1,H1,I1,J1,M1,N1,O1,L1,K1]	P1 [Q1,R1,S1,U1,W1,X1,Y1,Z1,B2]	C2 [G2,T1,V1,K2,J2,L2,M2] [P2,A2]	D2 [O2,E2,I2] [Q2,T2,F2] [H2,N2]	R2 [U2,S2] [W2,Y2,Z2]	V2 E3 X2 H3	A3 [B3,C3,D3] [I3,F3,L3] J3 K3	M3 R3 [N3,G3,U3] O3 P3	Q3 [S3,T3,X3]	V3 [W3,A4]	Y3 Z3

Figure B: Large AEC project's cCLSS with Task-Shuffling and Tabu-Search applied

- A1. Sign Off Layouts to Milestone 1
- B1. Create Civil Package Technical Specification
- C1. Create Dewatering Screens Mechanical Specification
- D1. Design Dewatering Feed Conveyor
- E1. Create Pumps Mechanical Specification
- F1. Create Float Cells & Blowers Mechanical Specification
- G1. Create Sampler Mechanical Specification
- H1. Create SMP Package Technical Specification
- I1. Create E & I Installation Package Technical Specification
- J1. Layout Dewatering Feed Conveyor
- K1. Create Instrumentation Specification
- L1. Create PLC Specification
- M1. Create UPS Specification
- N1. Create Belt Scales Mechanical Specification
- O1. Create Valves Mechanical Specification
- P1. Procure Instrumentation
- Q1. Procure Valves
- R1. Procure Dewatering Screens
- S1. Procure Pumps
- T1. Procure Float Cells & Blowers
- U1. Procure Sampler
- V1. Procure Belt Scales
- W1. Model Dewatering Feed Conveyor
- X1. Procure PLC
- Y1. Procure E & I Installation Package
- Z1. Procure Civil Package
- A2. Procure UPS
- B2. Procure SMP Package
- C2. Create Dewatering Feed Conveyor Drawings
- D2. Size Sub-Station
- E2. Layout Flotation Support & Access Structure
- F2. Model Belt Scales

- G2. Model Valves
- H2. Layout Dewatering Support & Access Structure
- I2. Model Float Cells & Blowers
- J2. Model Dewatering Screens
- K2. Model Pumps
- L2. Model Instrumentation
- M2. Model Sampler
- N2. Create MCC & Lighting Specification
- O2. Design Flotation Support & Access Structure
- P2. Design Dewatering Support & Access Structure
- Q2. Design Dewatering Civils
- R2. Design Flotation Civils
- S2. Design Sub-Station Civils
- T2. Check Dewatering Feed Conveyor Drawings
- U2. Layout Flotation Civils
- V2. Finalise Flotation Support & Access Structure CAD Model
- W2. Layout Dewatering Civils
- X2. Finalise Dewatering Support & Access Structure CAD Model
- Y2. Procure MCC & Lighting
- Z2. Layout Sub-Station Civils
- A3. Create Dewatering Support & Access Structure GAs
- B3. Create Flotation Support & Access Structure Gas
- C3. Extract Dewatering Structural MTOs
- D3. Extract Flotation Structural MTOs
- E3. Finalise Flotation Civil CAD Model
- F3. Finalise Dewatering Civil CAD Model
- G3. Issue Dewatering Feed Conveyor Drawings
- H3. Finalise Sub-Station CAD Model
- I3. Update Sub-Station Layouts
- J3. Create Flotation Civil Drawings
- K3. Update Flotation Layouts
- L3. Create Sub-Station Civil Drawings
- M3. Update Dewatering Layouts
- N3. Create Dewatering Civil Drawings
- O3. Check Dewatering Support & Access Structure Drawings
- P3. Check Flotation Support & Access Structure Drawings
- Q3. Create Dewatering Rebar Drawings
- R3. Create Flotation Rebar Drawings
- S3. Issue Dewatering Support & Access Structure Drawings
- T3. Issue Flotation Support & Access Structure Drawings
- U3. Create Sub-Station Rebar Drawings
- V3. Check Flotation Civil Drawings
- W3. Check Dewatering Civil Drawings
- X3. Check Sub-Station Civil Drawings
- Y3. Issue Flotation Area Civils
- Z3. Issue Dewatering Area Civils
- A4. Issue Sub-Station Area Civils

See electronic PLEP model for details ***APPENDIX C: SOURCE CODE AND OTHER MATERIAL.***

APPENDIX C: SOURCE CODE AND OTHER MATERIAL

BIBLIOGRAPHY

- [1] See Wikipedia, *Graph (mathematics)*,
http://en.wikipedia.org/wiki/Graph_%28mathematics%29 (as of Jan. 8, 2008, 22:35 GMT).
- [2] Huhnt, W. (2004): Progress Measurement in Planning Processes on the Base of Process Models, *Xth International Conference on Computing in Civil and Building Engineering*, June 02-04. Weimar, Germany.
- [3] Eygelaar, A. (2004): *Modeling the Engineering Process*. Final year project, University of Stellenbosch, Stellenbosch, South Africa.
- [4] Pahl, P.J., Damrath, R. (2001): *Mathematical Foundations of Computational Engineering*. Springer-Verlag Berlin Heidelberg, Germany.
- [5] See Wikipedia, *Strongly connected component*,
http://en.wikipedia.org/wiki/Strongly_connected_component (as of Jan. 8, 2008, 22:36 GMT).
- [6] See Wikipedia, *Generics in Java*, http://en.wikipedia.org/wiki/Generics_in_Java (as of Jan. 8, 2008, 22:37 GMT).
- [7] Tamara Munzner, *Interactive Visualization of Large Graphs and Networks*,
http://graphics.stanford.edu/papers/munzner_thesis/ (May 2007).
- [8] Michel Gendreau, An Introduction to Tabu Search,
http://www.ifi.uio.no/infheur/Bakgrunn/Intro_to_TS_Gendreau.htm (May 2007).
- [9] Sait, S.M., Youssef, H. (1999): *Iterative Computer Algorithms in Engineering*. Wiley - IEEE Computer Society Press, California, USA.
- [10] Boost C++ Libraries, *Boost Graph Library*,
http://boost.org/libs/graph/doc/table_of_contents.html (May 2007).