17th International Conference on the Applications of Computer
Science and Mathematics in Architecture and Civil Engineering
K. Gürlebeck and C. Könke (eds.)
Weimar, Germany, 12–14 July 2006

# ENGINEERING PROCESS MODEL SPECIFICATION AND RESOURCE LEVELING

**A.B. Eygelaar*** **and G.C. van Rooyen**

*University of Stellenbosch
*Stellenbosch, South Africa*
E-mail: eygelaar@sun.ac.za

**Keywords:** Process Modeling, Model Specification, Tabu Search, Resource Leveling, Logical Sequence of Tasks.

**Abstract.** *The acceptance of process modeling in AEC industries do not exclusively depend on the results it can provide, but the ease at which these results can be attained. Specifying a complex AEC process model is a dynamic exercise that is characterized by many modifications over the process model's lifespan. This article looks at reducing specification complexity, reducing the probability for erroneous input and allowing consistent model modification.*

*Furthermore, the problem of resource leveling is discussed. Engineering projects are often executed with limited resources and determining the impact of such restrictions on the sequence of Tasks is important. Resource Leveling concerns itself with the restrictions caused by limited resources. This article looks at using Task-shifting strategies to find a near-optimal sequence of Tasks that guarantees consistent Dataset evolution while resolving resource restrictions.*

## 1 INTRODUCTION

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. However, the Architecture/Engineering/Construction (AEC) industries present unique challenges that complicate the modeling of their processes. A simple Engineering process model, based on the specification of *Tasks*, *Datasets*, *Persons* and *Tools*, and certain relations between them, has been developed, and its advantages over conventional techniques have been illustrated [1]. Graph theory is used as the mathematical foundation mapping *Tasks*, *Datasets*, *Persons* and *Tools* to vertices and the relations between them to edges forming a directed graph.

The acceptance of process modeling in AEC industries not only depends on the results it can provide, but the ease at which these results can be attained. Specifying a complex AEC process model is a dynamic exercise that is characterized by many modifications over the process model's lifespan. This article looks at reducing specification complexity, reducing the probability for erroneous input and allowing consistent model modification.

Furthermore, the problem of resource leveling is discussed. Engineering projects are often executed with limited resources and determining the impact of such restrictions on the sequence of *Tasks* is important. Resource Leveling concerns itself with the restrictions caused by limited resources. This article looks at using *Task*-shifting strategies to find a near-optimal sequence of *Tasks* that guarantees consistent *Dataset* evolution while resolving resource restrictions.

## 2 ENGINEERING PROCESS MODEL, COMPONENTS AND RELATIONS

Four sets of components have been identified as building blocks for the process model: set of *Tasks*, set of *Datasets*, set of *Persons* and a set of *Tools*. *Tasks* represent operations on *Datasets*, raising the *Data*set's status value in the process. *Tasks* are executed by *Persons* and *Datasets* are operated upon using *Tools*. There are 16 possible binary relations between the components, of which only 3 are user specified while the remaining 13 relations can be derived mathematically. The 3 user specified heterogeneous binary relations are the relation between the set of *Tasks* and the set of *Datasets* (*Task-Dataset* relation), the relation between the set of *Tasks* and the set of *Persons* (*Task-Person* relation), and the relation between the set of *Datasets* and the set of *Tools* (*Dataset-Tool* relation). Refer to figure 1 for an overview of the heterogeneous binary relations and their semantics.

| | Person | Task | Dataset | Tool |
|---|---|---|---|---|
| Person | | executes | access (read/write) | use |
| Task | is executed by | | access (read/write/ modify) | requires |
| Dataset | is accessed by (read/write) | is accessed by (read/ write/modify) | | can be edited by |
| Tool | is used by | is required by | can edit | |

*Highlighted binary relations are user specified.

Figure 1: Overview of heterogeneous binary relations

Of the four derived homogeneous binary relations, on the diagonal in figure 1, the relation in the set of *Tasks* is the most important:

$$\textit{Relation in the set of Tasks} :=$$

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

T Set of Tasks

The *Task-Dataset* relation and three predefined rules are used to determine the relation (1) in the set of *Tasks*, called the *Consistent Sequence of Tasks* graph (*CST*-graph) [1, 2]. Every edge in the *CST*-graph represents a relationship between its incident *Tasks* that must be honored in order to guarantee consistent development of *Datasets*. The *CST*-graph is topologically sorted to produce an optimal *Logical Sequence of Tasks* (*LST*). A graph and its *LST* are known as a *solution*. The result of topological sorting is a list of logical steps and each step contains *Tasks* that have to be executed before the *Tasks* in the following steps can be executed. The *CST*-graph and its *LST* are considered an optimal solution, because the least amount of logical steps is used and *Tasks* are assigned to the earliest possible logical step.

The remaining 3 homogenous binary relations, refer to table 1, can easily be derived using the *CST-solution* together with the other heterogeneous binary relations. For example, the relation in the set of *Persons* can be derived using the *CST*-graph and the user specified *Task-Person* relation.

| Homogeneous binary relation | Meaning of relation | Required information to derive relation |
|---|---|---|
| Relation in the set of *Tasks* (*CST*-graph) | Consistent sequence of *Tasks* | *Task-Dataset* relation and three predefined rules |
| Relation in the set of *Persons* (*Person-Person* relation) | *Person* loading (Which *Person* is utilized in what logical step) | *CST*-solution and *Task-Person* relation |
| Relation in the set of *Tools* (*Tool-Tool* relation) | *Tool* loading (Which *Tool* is utilized in what logical step) | *CST*-solution and *Task-Tool* relation |
| Relation in the set of *Datasets* (*Dataset-Dataset* relation) | *Dataset* evolution (What a *Dataset's* status is at the beginning and end of each logical step) | *CST*-solution and *Task-Dataset* relation |

Table 1: Overview of homogeneous binary relations

# 3 SPECIFICATION OF PROCESS MODEL

Although the amount of user specification has been significantly reduced to four component sets and three heterogeneous binary relations, AEC processes' inherent dynamic and complex nature compel the *Graphical User Interface* (*GUI*) to reduce user interaction complexity and erroneous input. The model should support easy and consistent model modification.

## 3.1 Reducing user interaction complexity and erroneous user input

It is the responsibility of the user to specify the components and three heterogeneous binary relations of the process model. Each type of component or relationship requires certain attributes to be correctly specified before the component or relationship may be added to the model. For example: during the specification of a *Dataset* component the user must specify a unique name, a valid completion weight and one milestone class. How the user is prompted and enters the information into the *GUI* should be intuitive and simple. Verification of the user entered information is handled by the *GUI*, but some verification is delegated to the model. The following *Dataset* oriented verifications have to be completed before the *Dataset* component may be added to the model:

- Leading- and trailing whitespace are removed and the user specified name is checked for uniqueness.
- Completion weight is checked whether it is a valid number.
- The *GUI* enforces the selection of one and only one milestone class.

Each component's specification, may it be creating or modifying a component, is handled in *one* popup focusing the user's attention. If an extra popup is required, for example to specify a *Dataset's* attributes and milestone class in figure 2, the child popup is displayed over the parent popup to reduce visual complexity. The use of modal popups forces a user to complete a component's specification before advancing to the next undertaking.

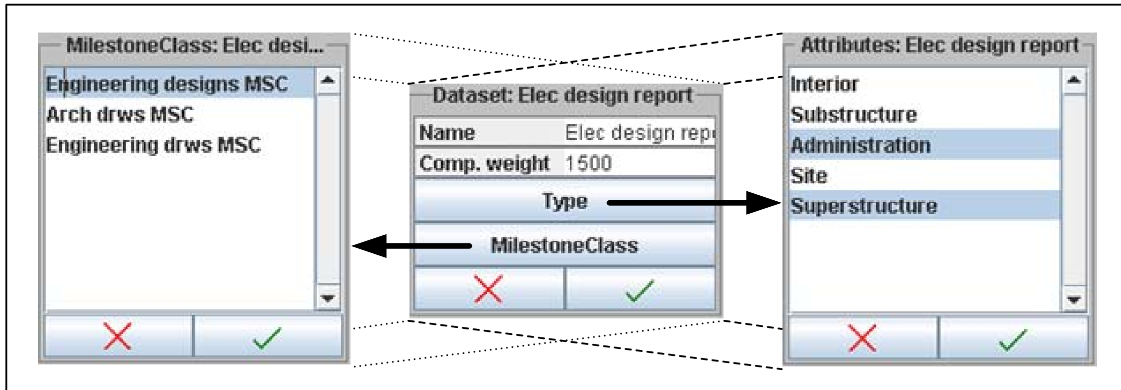


Figure 2: Specifying a Dataset component

Binary relations consist of ordered pairs and the relationship between the pair's two components is directed with a beginning and end component. Figure 3 shows the relationships a user must specify. The relation between the set of Tasks and the set of Datasets (Task-Dataset relation) is used as an example to explain the specification process. When specifying a Task-Dataset relationship for a specific Task (*Current Task*), any number of Datasets may have relationships with the *Current Task* as long as the Datasets are different. The user can specify three types of Task-Dataset relationships: read, modify or create. Only modify and create type Task-Dataset relationships involve specifying the dataset's status level after the execution of the *Current Task*. After investigating different specification procedures the *drag and drop* strategy performed the best.

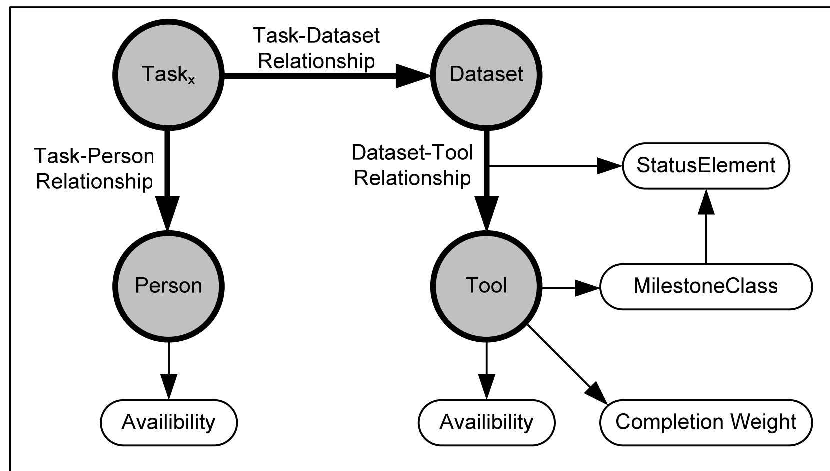


Figure 3: User specified relationships

In figure 4 the user is busy specifying a *Task-Data* relationship. "Create architectural design" is dragged from the *Task*-list and dropped into *Current Task*-list and automatically "Create architectural design" is set as the *Current Task*. Now the user drags "Architectural drawings" from the *Dataset*-list and drops it into the Create-list and automatically a *Task-Dataset* relationship (of type Create) is created between "Create architectural design" and "Architectural drawings". When the user selects "Architectural drawings" the user can choose a status level sourced from "Architectural drawings" milestone class.

Incorporating *drag and drop* functionality simplifies user interaction with the *GUI* as well as preventing erroneous input. The *drag and drop* process can be customized to guide the user while performing a *drag and drop* operation. For instance, the user's intention to copy or move the component being dragged is reflected in the shape of the cursor. Also, drop targets

can be configured to only accept certain types of components and components originating from certain drag sources reducing the probability for the user to specify inconsistent relationships.

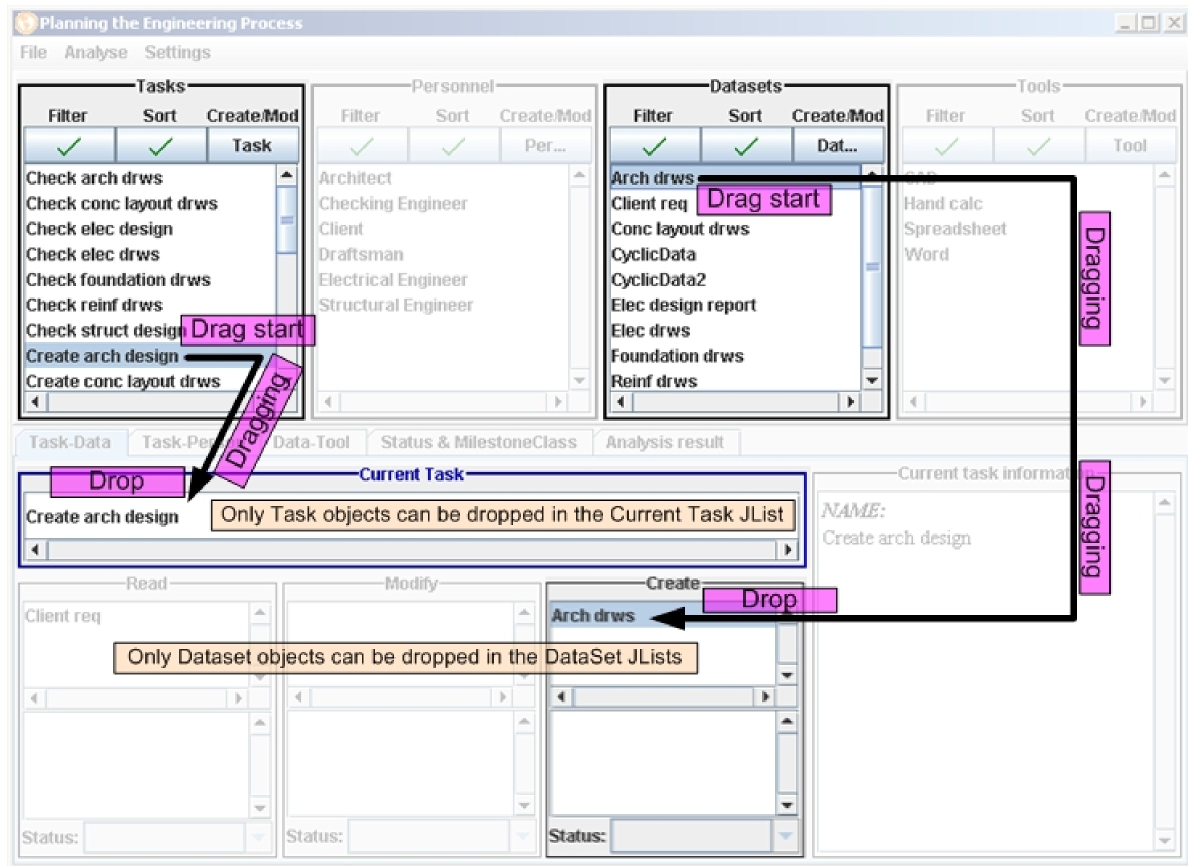

Figure 4: Drag and drop

### 3.2 Model modification

AEC projects can be extremely volatile undergoing constant modifications that complicate the modeling of such processes. Compounding this volatility are high levels of referencing between components caused by the use of graph structures. Therefore, model modification must be embraced while consistent referencing is guaranteed.

Herding resident objects into one model map reduces consistency issues and increase manageability of the resident objects. If a collection of objects of specific interest are required, it can be extracted from the model map using customizable filters. Using a single model map necessitates that all objects be uniquely identifiable. Each object has a persistent String identifier (*ID*) that is mapped to the object's reference inside the model map. Objects are not referenced at object level, but at *ID* level. Changing and modifying objects are easily facilitated with *ID* referencing. Two modification types have been identified:

reName:

An object's *ID* is modified from its *old ID* to a *new ID* (figure 5). The object's reference is not replaced with another object's reference, in other words, the object itself stays the same. The following steps explain how to execute this procedure to guarantee consistent *ID* referencing:

- Iterate over all the objects in the model map.
- If an object's *ID* matches *old ID* replace it with *new ID*.
- If an object references an *ID* that matches *old ID*, replace it with *new ID*.
- Remove map entry with Key = *old ID*.
- Insert map entry with Key = *new ID* and Value = **old** object reference.

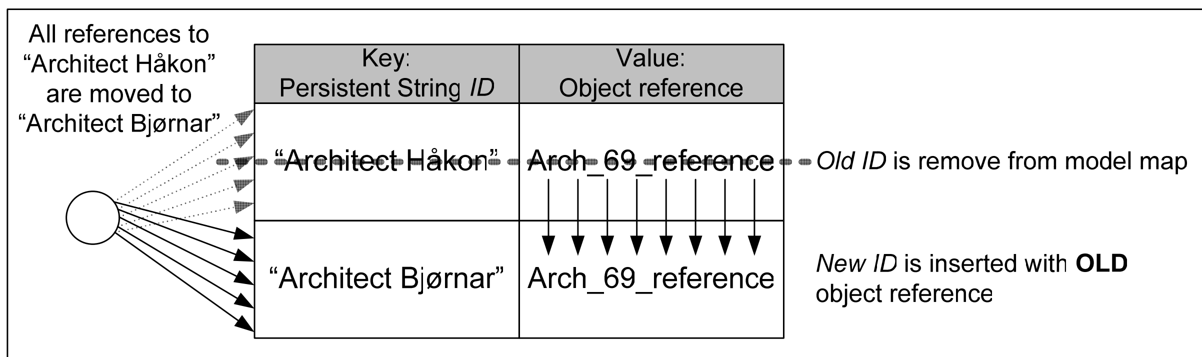


Figure 5: reName

reReference:

An object's *ID* is modified from its *old ID* to a *new ID* (figure 6). The object's reference is replaced with another object's reference, in other words, the object itself is also replaced. The following steps explain how to execute this procedure to guarantee consistent *ID* referencing:

- Iterate over all the objects in the model map.
- If an object's *ID* matches *old ID* replace it with *new ID*.
- If an object references an *ID* that matches *old ID*, replace it with *new ID*.
- Remove map entry with Key = *old ID*.
- Insert map entry with Key = *new ID* and Value = **new** object reference.

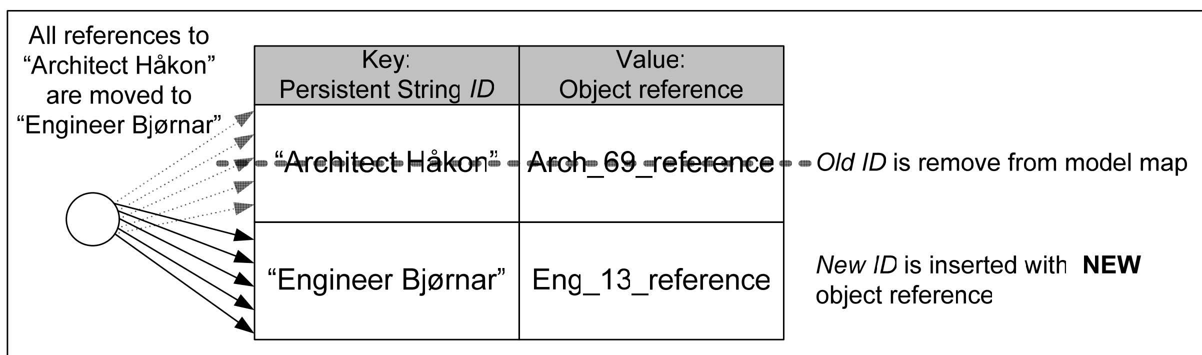


Figure 6: reReference

# 4 RESOURCE LEVELING

Resource leveling concerns itself with restrictions caused by limited resources. Engineering projects are often executed with limited resources and determining the impact of such restrictions is important. The focus here is to resolve resource conflicts in an optimal way.

When deriving the *CST-solution*, as described in section 2, only the consistent evolution of *Datasets* is taken into consideration. The impact of limited resources is not considered and indirectly the assumption is made that resources have an unlimited availability. Ignoring the impact of limited resources, the *LST* holds the least amount of logical steps to execute *Tasks* as early as possible. Thus, the *CST-solution* is the unique optimum solution with unlimited resources. Assuming there is a limit on the availability of resources creates the possibility that resources may be overutilized. Resources are utilized when *Tasks* are executed which mean that concurrent execution of *Tasks* requires concurrent utilization of resources. If the concurrent utilization is greater than the availability of resources the problem of resource overutilization is born. In theory, *Tasks* responsible for resource overutilization can only be executed if the availability of resources is increased or the concurrent utilization of these resources decreased sufficiently to resolve the resource overutilization. Increasing the availability of overutilized resources is a simple strategy, but not always an efficient or even possible strategy. This turns the focus to reducing resource overutilization by shifting the *Tasks* responsible away from areas of concurrent execution. Two strategies are used to reduce the concurrent utilization of resources namely shuffling and a tabu search technique.

## 4.1 Shuffling

The sequence of *Tasks* inside a logical step has thus far enjoyed no attention and there are no restrictions on this sequence either. *Shuffling*, as graphically explained in figure 7, is the shifting and stacking of *Tasks* inside the logical step's boundaries with the aim of reducing concurrent execution of *Tasks*. *Shuffling* does not change *Tasks'* step numbers, because the *Tasks* remain in the same logical step. *Shuffling* can only reduce or at its worst have no effect on resource consumption; therefore shuffling will always be employed.

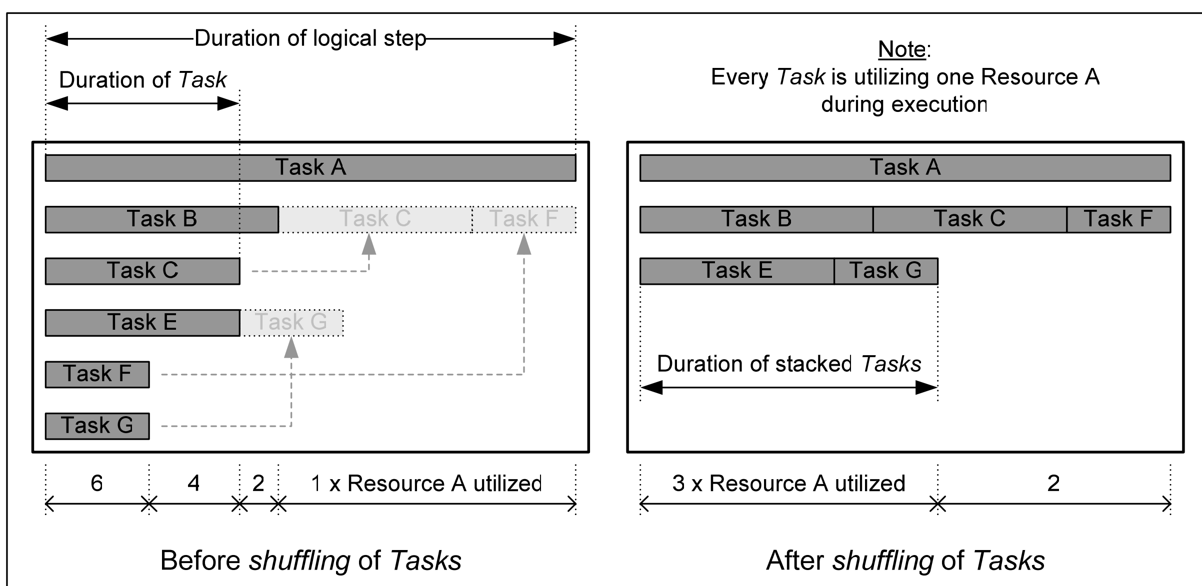


Figure 7: Shuffling

By default a logical step's duration will be set to the minimum and this lower bound is set by the step's longest *Task* duration. A logical step's duration does not have an upper bound. Longer step durations offer the opportunity for a higher degree of *shuffling*, but the increase in step duration has to be weighed up against the reduction of resource utilization. Also, *shuffling* is highly sensitive to the differences in a step's *Task* durations. If a step's *Task* durations are equal, no *shuffling* is possible without increasing the step's duration. However, if adequate differences in a step's *Task* durations are present *shuffling* is possible even with the step's duration set at its minimum.

## 4.2 Tabu Search

The *shuffling* strategy has limited impact and to achieve further reduction of concurrent execution of *Tasks* responsible for overutilized resources necessitates the shifting of *Tasks* across the boundaries of logical steps. This strategy's inherent explosive complexity compels the use of heuristic techniques to determine alternative solutions that account for resource restrictions. *Tabu search* is investigated as a searching strategy.

Tabu search definition:

*Tabu search* is a form of local neighborhood search. The current solution (*S*) is used as a starting point for the search. Its local neighborhood 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*. Recent *moves* are recorded in a tabu list to force the search away from visited solutions, because *moves* in the tabu list are not allowed to be executed. *Moves* are evaluated 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.

Application of TS in process model:

The *CST-solution* is a unique optimum *solution*, but it is based on the assumption of unlimited resources. This implies that the remaining *solution* space, including the *solutions* that satisfy limited resources, consist only of *solutions* where combinations of *Tasks* are delayed (shifted to or executed in later logical step). Thus, using the *CST-solution* as the starting *solution* a *move* can only be defined as shifting a *Task* to a later logical step. Shifting a *Task* is achieved by *edge insertion* into the current *solution*.

Edge insertion:

A *move*, and thereby shifting a *Task* to a later logical step, is realized by *edge insertion*. The edge is inserted from the anchor *Task* to the chosen drifter *Task*. The anchor *Task* is either a *Task* that is not responsible for resource overutilization or the *Task* least desirable to shift. The drifter *Task* is the most desirable *Task* to shift. In figure 8 *Task* B is the anchor and Task C the drifter. *S** is reached from *S* by *edge insertion* and topological sorting *S** reveals the impact of the *move*.

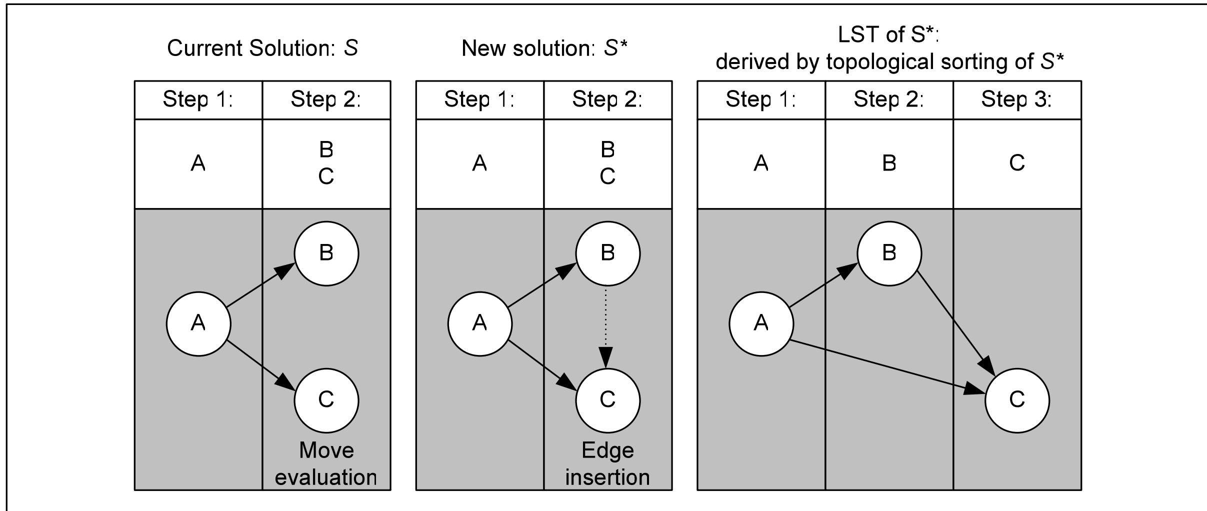


Figure 8: Edge insertion

Move evaluation:

During the execution of a *tabu search*, *moves* are only performed if it will shift *Tasks* responsible for resource overutilization. *Moves* are evaluated, using a value function, before the best *move* can be chosen. Two types of criteria are used namely value and disqualification criteria. A value criterion is used by the value function to determine a *move's* value while a disqualification criterion determines if the *move* is acceptable or not. The value function (2) consists of the summation of weighted value criteria where each criterion's weight determines the influence the criterion will have on the value function:

$$\text{Value of } move = \sum (\text{criterion weight})_i \text{ x } (\text{criterion value})_i \text{ for all criteria } I \qquad (2)$$

*Task slack* and *Task resource utilization* are used as value criteria and *Minimum Task resource utilization* as a disqualification criterion:

*Task slack*:

*Task slack* is the number of logical steps a *Task* can be shifted without increasing the overall number of logical steps. *Tasks* on the critical path will have zero slack. A *move* with higher slack is more desirable.

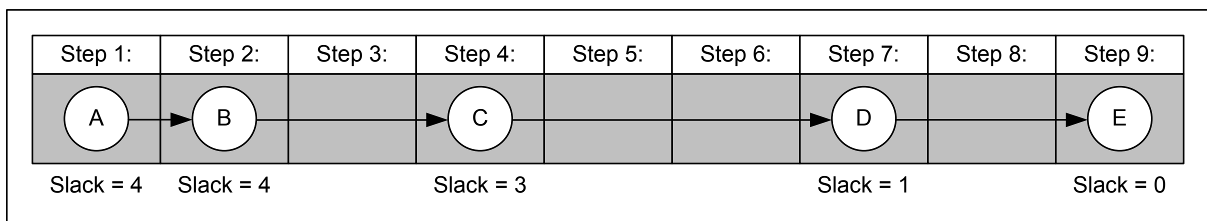


Figure 9: Task slack

*Task resource utilization*:

Shifting a *Task* that is responsible for $k$ overutilized resources is more attractive than shifting a *Task* that is responsible for $j$ overutilized resource if $k > j$.

*Minimum Task resource utilization*:

A *Task* utilizes a resource $k$ times. The availability of the resource is $j$. If $k > j$ then the *Task* is responsible for $k - j = w$ over utilizations of the resource.

Shifting the *Task* to a subsequent logical step will not resolve the overutilization of the resource, because the *Task* is still associated *k* times with the resource and the availability of the resource remains *j*. A *move* where $k > j$ is not useful and is therefore disqualified.

<u>Solution evaluation:</u>

A *Solution* consists of a graph and the graph's LST. *Solutions* must be comparable with each other to be able to choose the best *solution* as the tabu seach explores the *solution* space. A value function is used to evaluate a *solution* to determine a value. Any combination of a *solution's* properties can be incorporated. Chosen *solution* properties used in the value function are weighed according to its importance. If a user decides that the number of logical steps must be the dominant property the value function will assign high values to *solutions* with a lower number of logical steps.

*Solutions* are evaluated, using a value function, before the best *solution* can be chosen. Two types of criteria are used namely value and disqualification criteria. A value criterion is used by the value function to determine a *solution's* value while a disqualification criterion determines if the *solution* is acceptable or not. The value function (3) consists of the summation of weighted value criteria where the each criterion's weight determines the influence the criterion will have on the value function:

$$\textit{Value of solution} = \sum (\textit{criterion weight})_i \times (\textit{criterion value})_i \textit{ for all criteria } i \qquad (3)$$

*Solution duration* is used as value criterion and *Overutilized resources* as a disqualification criterion:

<u>*Solution duration*</u>:

Refer to figure 10 for an explanation of a logical step's contribution to the *Solution duration*.

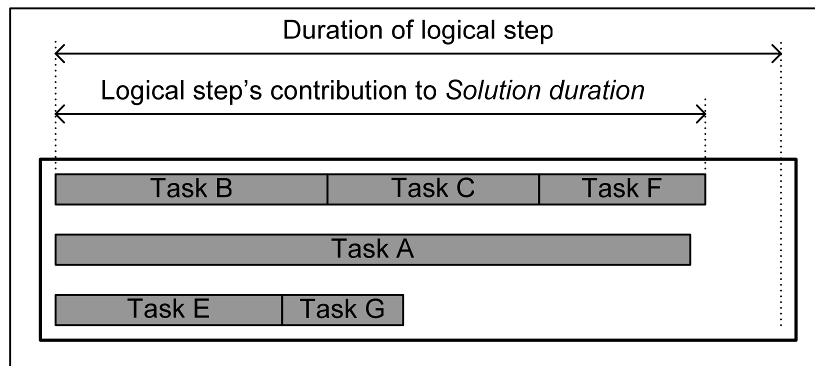


Figure 10: Logical step's contribution to solution duration

<u>*Overutilized resources*</u>:

A limit can be specified on the number of overutilized resources that is acceptable. Let this limit be *k*. If a *solution* contains *j* overutilized resources and $j \leq k$ the *solution* is acceptable, otherwise it is not acceptable.

Tabu list:

The major idea of the *tabu list* is to classify certain search directions as forbidden (tabu) to prevent returning to previously visited *solutions*. The *tabu list* acts as short-term memory forcing the search away from recently visited *solutions*. To reach *S** from *S* the best *move* is performed and placed into the *tabu list*. Only *moves* not present in the *tabu list* may be performed.

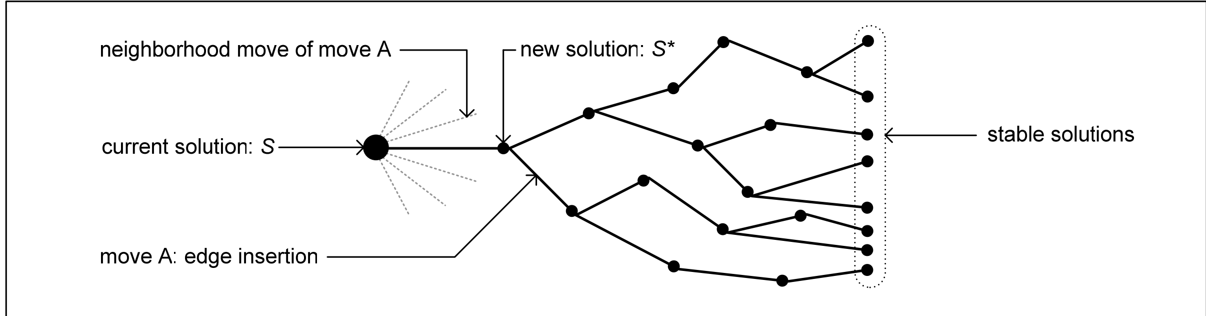


Figure 11: Searching the solution space

To strictly enforce the *tabu list* can severely restrict the tabu search. To reach a *solution* requires a unique combination of *moves*. Referring to figure 11 it is easy to see that to reach any *solution* requires traversing a unique path (combination of *moves*). If the first *move* from the current *solution* in figure 11 is placed in the *tabu list* and then strictly forbidden, only one of the indicated *stable solutions* can be reached. That is clearly not a desirable situation. *Aspiration* criteria allow *moves* in the *tabu list* to be performed.

One *aspiration* criterion is used that allows a tabu *move* to be performed *m* times before it is strictly forbidden and is called the Tabu Redo Limit. Let $t_A$ be the number of times *move* A has been performed and $N_A$ the set of neighborhood *moves* of move A. Then *move* A can only be performed if:

$$t_A < m \ \wedge \ t_A \leq min\{ t_i \ \forall i \in N_A\} \qquad (4)$$

Tabu search-algorithm:

*Tabu search* is a searching technique that is adapted to the particular characteristics of the *solution* space being searched. In this case the *tabu search algorithm* has distinct steps it follows while searching the *solution* space.

Tabu search-run:

A *tabu search-run* starts with a current *solution S*. Investigating the current solution's *LST* provide information like resource utilizations. While iterating over the *LST's* logical steps they are checked for resource overutilizations. If such a step is located an attempt is made to perform a *move* and if the attempt fails the iteration continues. If the attempt is successful the *move* is performed, *S** is reached, and the iteration terminates. However, if the iteration ends without performing a *move* the *tabu search-run* is terminated and *S* is considered a *stable solution*. A *stable solution* is when no more *moves* can or should be performed and the structure of the *solution's* graph remains stable. Refer to figure 12 for a graphical representation of the *tabu search-run* algorithm.

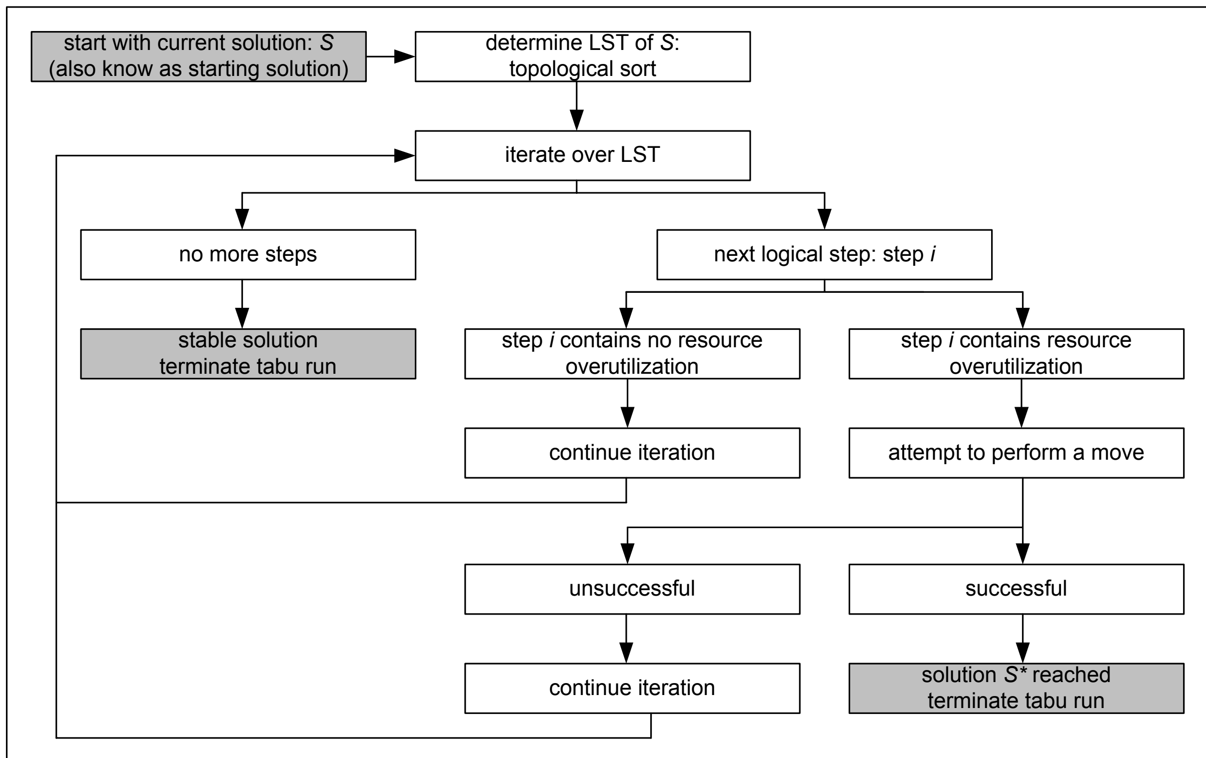


Figure 12: Tabu Search-run diagram

Tabu Search-relay:

A *tabu search-relay* consists of several *tabu search-runs*, but the *tabu search-runs* always start with the *CST-solution* as its starting *solution*. Each *tabu search-relay* will continue to perform *tabu searh-runs* until a *stable solution* is reached. When a *stable solution* is reached the *tabu search-relay* terminates. In figure 13 the highlighted path is a *tabu search-relay*. Each segment of the path is a *tabu search-run*. When *S** is reach from *S* another segment is added to the path until *S** is stable.

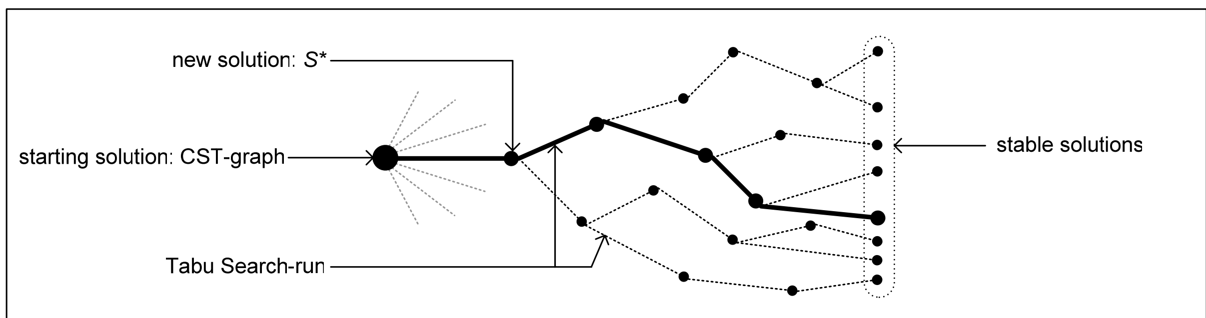


Figure 13: Tabu Search-relay

Tabu Search-race:

A *tabu search-race* consists of several *tabu search-relays*. Each *tabu search-relay* produces a *stable solution* that is evaluated and the best *solution* is stored. The best *solution* may or may not be an optimum *solution* for the given resource availability, but it is the best *solution* the *tabu search-algorithm* could find.

# 5 APPLICATION AND RESULTS

A Java application is implemented and a small AEC project modeled. The project consist of 22 *Tasks*, 8 *Datasets*, 6 *Persons* and 4 *Tools*. The *Tools* and *Persons* are the resources of the project. The specified parameters for the *tabu search-algorithm* were:

- Step duration available: minimum (a logical step's longest *Task* duration)
- Number of *tabu search-runs*: 100
- *Solution* value function criterion: based only total duration
- *Solution* disqualification criterion: only *solutions* with zero resource overutilizations are eligible for best solution
- *Move* value function criteria: *Task slack* weighted at 0.8 and *Task resource utilization* reduction weighted at 0.5
- *Move* disqualification criterion: *Minimum Task resource utilization*

The *tabu search-algorithm* searches for *solutions* by delaying *Tasks* from their original positions in the *CST-solution* (starting *solution*). Since delaying *Tasks* are the only *moves* allowed it explains why the best *solution* (best in terms of *solution evaluation*) found by *tabu search-algorithm* has a longer duration compared to the *CST-solution's* duration. This increase in duration is reflected in the reduction of the overutilized resources.

The amount by which the *solution's* duration increases depends on how much resource overutilization and *Task slack* were present in the *CST-solution*. *Shuffling* thrives if a logical step has many *Tasks* with a good spread in duration. As the tabu search algorithm performs *moves* in an attempt to decrease the concurrent execution of *Tasks*, *shuffling* becomes less successful.

Although the process being modeled only consists of 22 *Tasks* meaningful results could be attained. Refer to table 2 for the *tabu search* results.

| Solution | *CST-solution* | Best *tabu search solution* | Worst *tabu search solution* |
|---|---|---|---|
| Steps | 6 | 10 | 13 |
| Duration | 7460 | 9860 | 12335 |
| Resource overutilizations | 4 | 0 | 0 |
| *Shuffling*: Concurrent task execution reductions | 9 | 3 | 2 |
| *Shuffling*: Resource utilization reductions | 25 | 7 | 4 |
| Specified step duration | minimum | minimum | minimum |
| *Moves* | n/a | 11 | 19 |
| *Tabu search-runs* | n/a | 1217 | 1217 |
| *Tabu search-relays* | n/a | 100 | 100 |
| Time to compute | n/a | 100 seconds | 100 seconds |

Table 2: Tabu Search results

Figure 14 shows how the best and worst *solution's moves* affected the duration and resource overutilization. It can be seen that performing a best *move* does not necessarily reach an improved *solution* $S^*$ from $S$, in terms of resource overutilizations. This characteristic allows the *tabu search-algorithm* to escape local optima.

Figure 14: Effects of *moves* on duration and resource overutilization

Specifying step durations longer than the minimum step duration increases the efficiency of *shuffling* and consequently reduces the starting *solution's* resource overutilizations. Since only *Tasks* responsible for overutilized resources are allowed to be shifted the amount of *moves* available will also reduce. As the *tabu search-algorithm* search through the *solution* space its reduced choice of *moves* does not allow it to reach the same best *solution* as with the minimum specified step durations. That is why table 3 shows an increase in s*olution duration*, a decrease in *moves*, decrease computation time (direct result of decrease in moves) and increase in *shuffling* efficiency as the specified step duration increases.

| Solution | Best *tabu search solution* with following amounts available for step duration: | | | |
|---|---|---|---|---|
| | minimum | 1500 | 2500 | 4000 |
| Steps | 10 | 7 | 7 | 6 |
| Duration | 9860 | 10010 | 11000 | 12250 |
| Resource overutilizations | 0 | 0 | 0 | 0 |
| *Shuffling*: Concurrent task execution reductions | 3 | 12 | 12 | 14 |
| *Shuffling*: Resource utilization reductions | 7 | 11 | 28 | 33 |
| Specified step duration | minimum | 1500 | 2750 | 4000 |
| *Moves* | 11 | 8 | 4 | 3 |
| *Tabu search-runs* | 1217 | 1142 | 385 | 285 |
| *Tabu search-relays* | 100 | 100 | 100 | 100 |
| Time to compute | 100 seconds | 95 | 38 | 30 |

Table 3: Influence of step duration specification

# 6 CONCLUSION

The implemented *Graphical User Interface* and model constitute a substantial improvement over existing implementations. Implementing the *GUI* exposed several areas for future development and improvement. For example, the *drag and drop* process can be improved to provide stricter guidance and more feedback to the user. Essentially, however, *GUI* development is an ongoing exercise.

A *solution*, especially its *LST*, encapsulates large amounts of information that the user need to visualize in a rational manner. Apart from visualizing the *LST* the user also need to be able to interact with the *LST*. Displaying a *LST* is graphically intensive and will require future development. The development of a proper interactive graphical representation will introduce new opportunities of user interaction with the *LST* like *drag and drop*.

The provisional results attained by the *tabu search-algorithm* are very encouraging and future benchmarking will include larger AEC process models. Although the tested AEC process model was relatively small in size, only twenty two *Tasks*, a human will most

probably not have been capable of solving it in an efficient way. Larger models can only be solved computationally.

*Shuffling* proved to be surprisingly efficient and justifies future development. *Shuffling Tasks* is a combinatorial problem, but currently the algorithm only calculates one solution. The drawbacks of *shuffling* are that it is sensitive to the number of *Tasks* present in a logical step and the spread of the *Tasks'* durations. Unfortunately the impact of these drawbacks cannot be significantly reduced with improving *shuffling* algorithm.

Improving the *tabu search-algorithm* can be achieved on many fronts. The *aspiration-*, *move evaluation-* and *solution evaluation* criteria can be expanded in order to increase the level of complexity at which these evaluations operate. The quality of a decision is only as good as its preceding evaluations. It may be assumed that for larger models computational limitations will decrease the searchable area of the solution space. In this case the necessity of performing the best *move* becomes more critical.

Currently a *move* is only associated with *edge insertion*, but future development may focus on introducing removal of inserted edges as possible *moves*. Implementing such a strategy will be more complex, but the resulting search may be more thorough.

The computational complexity of the *tabu search-algorithm* is high. Searching for one hundred solutions in a *solution* space containing only twenty two *Tasks* already consumes time of the order of a minute, using a Intel Pentium M 1.8GHz processor with 1GB of RAM. Computational complexity was not the primary focus during the development of the pilot implementation and future improvements are possible.

The biggest limitation this project has identified is the reliance on the *Logical Sequence of Tasks*. Topologically sorting the *CST*-graph compartmentalizes the *Tasks* into logical steps, which reduces the complexity of the problem, but in doing so introduce scheduling limitations. Specifically, step logic enforces jumps in the starting times of *Tasks*, whereas the *CST*-graph only enforces a *Task's* starting time, as shown in figure 15. It is proposed that future developments focus on using the *CST*-graph directly.

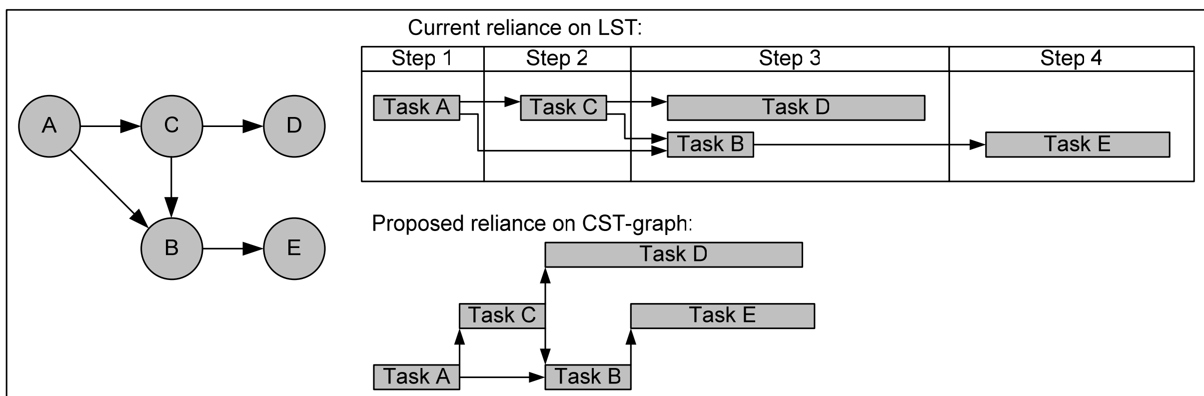


Figure 15: Proposed reliance on CST-graph

**REFERENCES**

[1] Huhnt, W. (2004): Progress Measurement in Planning Processes on the Base of Process Models, *$X^{th}$ International Conference on Computing in Civil and Building Engineering*, June 02-04. Weimar, Germany.

[2] Eygelaar, A. (2004): *Modeling the Engineering Process*. Final year project, University of Stellenbosch, Stellenbosch, South Africa.

Pahl, P.J., Damrath, R. (2001): *Mathematical Foundations of Computational Engineering*. Springer-Verlag Berlin HeidelBerg, Germany.

Sait, S.M., Youssef, H. (1999): Iterative Computer Algorithms in Engineering. Wiley - IEEE Computer Society Press, California, USA.