The Algorithm Assembly Set of Plato

Christoph Lenzen[1], Maria Theresia Wörle[2], Falk Mrowka[3] and Andreas Spörl[4]
DLR / GSOC, Oberpfaffenhofen, Germany, 82234 Wessling

Rüdiger Klaehn[5]
Heavens-Above GmbH, München, 81377

Driven by the requirements of earth observing satellite missions, the mission planning team of the German Space Operations Center (GSOC) has improved its scheduling engine to allow automated timeline generation for multiple interacting satellites. Whereas the past work included extensions of the modeling language and improvements on the performance, current work focusses on the algorithm framework. In order to allow future missions' scheduling software to reuse generic algorithms, special attention is given to the way one can add new sub-algorithms and combine them with existing ones.
This ePoster demonstrates the algorithm framework of GSOC's mission planning software Plato, using its interactive GUI Pinta. Based upon a typical multiple satellite planning problem, a priority based generic algorithm is presented, which solves this problem. We show how this algorithm can be split up into small sub-algorithms, each of which can be used separately and all of which can be combined in arbitrary ways. We demonstrate how this flexibility can be used to create modifications on the overall algorithm or include mission specific sub-algorithms. Although all presented algorithms are based on simple heuristics, this mechanism supplies a straight forward way to incorporate more sophisticated optimization algorithms.
The techniques demonstrated in this paper will be shown by means of the OnCall planning project. This project is used by GSOC in order to schedule the on-call shift times of its staff in order to implement 24/7 support for all important satellite sub-systems.

Nomenclature

GSOC = German Space Operations Center
Pinta = program for interactive timeline analysis (software application developed at GSOC)
Plato = planning tool (software library developed at GSOC)
GUI = graphical user interface

Contents

[1] Mission Planning System Engineer, Mission Operations, DLR, Oberpfaffenhofen, 82234 Wessling, Germany
[2] Mission Planning System Engineer, Mission Operations, DLR, Oberpfaffenhofen, 82234 Wessling, Germany
[3] Mission Planning System Engineer, Mission Operations, DLR, Oberpfaffenhofen, 82234 Wessling, Germany
[4] Mission Planning System Engineer, Mission Operations, DLR, Oberpfaffenhofen, 82234 Wessling, Germany
[5] Software Engineer, Heavens-Above, Pfingstrosenstrasse 2, 81377 München, Germany

I. Introduction

FOR many years, GSOC has been developing its in-house scheduling software Pinta/Plato. According to the needs of the TerraSAR-X/TanDEM-X mission, recent developments included improvements of the performance and an extension of the modeling language. This included a generic way to configure the constraints of the model, which allows fast and therefore cheap implementation of modifications on the required constraints. For a detailed description about the modeling language, see [1].

However not all change requests refer to the constraints, some refer to the algorithm. These modifications still included expensive source code development. The current work focuses on how algorithms should be built up in order to allow configuring a proper algorithm and, in case special code is necessary, how to embed the mission specific sub-algorithm into the configurable algorithm framework.

II. XML techniques

A very useful way to define a configuration file is provided by the XML schema language. An XML schema allows defining the way an XML file may look like, e.g. which elements it has and of which type the values of the elements must be. This allows checking a given xml configuration file for correctness with respect to a certain schema.

The great benefit of this technique is that there exist tools which convert the XML schema into a source code class definition file. This class is serializable by the XML serializer, which generates and reads xml files that match the XML schema definition file. For C#, you can use the xsd.exe, a .NET framework tool provided by Microsoft, for the Java platform there exists the JAXB project.

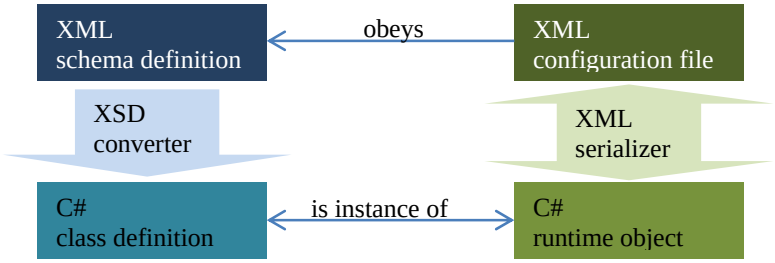

Figure 1. Interaction in between schema and code

Whereas this mechanism works straight forward with the constraint specifications, for the configuration of the algorithm, we need to provide a mechanism, which allows easily referencing a class, that might not yet be implemented during compile time of the Platon library. For example one might want to implement a special sub-algorithm, which shall be referenced by other algorithms, without recompiling the whole Platon library. For this purpose we have defined a schema containing an element, called *SequenceElement*, which allows specifying a class name and a configuration file for this class. We now can reference this schema inside an algorithm schema and use the SequenceElement in order to reference any class just by supplying its name. The idea behind is that the .NET SequenceElement object should provide the functionality to generate an object according to its class name and configuration file. This way the algorithm code can use its SequenceElement objects to dynamically generate the

specified object. It is not difficult to implement this code for a SequenceElement. However we faced a subtle challenge: the xsd.exe tool generates the C# files for each schema individually, thus for each XSD schema which uses the SequenceElement, an individual C# class 'SequenceElement' is generated. This doesn't affect serialization and deserialization; however one cannot write code for this class without the need to duplicate it for each SequenceElement definition.

We therefore implemented a small tool which works similar to the xsd.exe tool, except for the fact that each XML schema file is translated into its individual C# file and no classes are generated for elements of referenced schemata. Now we may need to reference certain base projects, which contain the definition of base schema classes, but we gain the possibility to write code for these base classes, which is accessible in all derived schema classes:

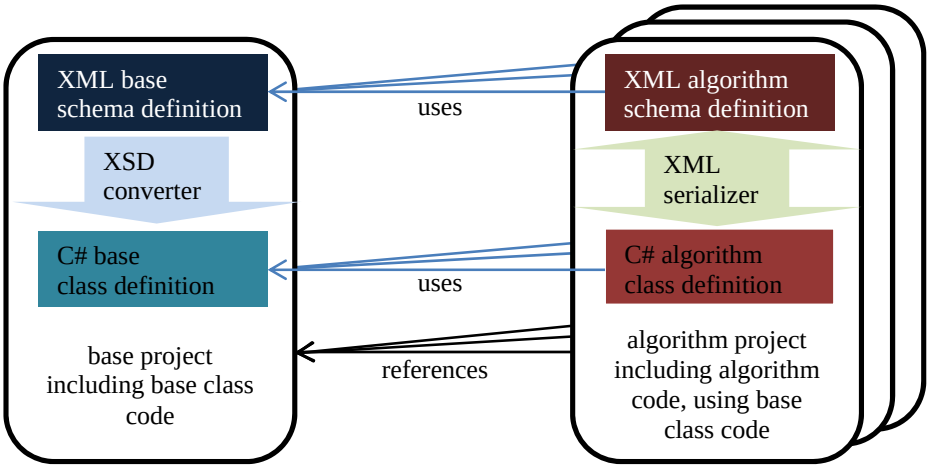

Figure 2. Interaction with base schema. *A referenced schema results in only one class, which means that code may be reused by all classes using this base schema class.*

Finally we implemented a little reflection magic inside the SequenceElement base class definition (including buffering for performance optimization) and our base XML type *SequenceElement* may be used by any algorithm schema definition in order to reference sub-algorithms or other special class types, together with their configuration files. Inside the code, an algorithm may use a *SequenceElement* property in order to generate a sub-algorithm object only by calling the respective factory method, which was defined inside the SequenceElement base class.

III. Generic Algorithms

In the following we describe several generic algorithms and algorithm fragments and how they can be combined to form complex heuristic algorithms.

A. Algorithms and Algorithm Fragments

The starting point of our development was the 'filter algorithm', which mainly consists of two decision steps:

1) Object selection:
   From all currently considered tasks and groups, select the one to process next .
   Remark: We don't call this variable selection, because the selected object can be either a task or a group.
2) Value selection:
   In case the object selection strategy has chosen a task, the value selection determines what timeline entry to create for this task.

In the original approach of algorithm configuration, we defined these two steps (and some further features) in one schema. However the more features we added to this algorithm, the more complex the schema grew. Additionally each time recompilation of the whole algorithm was necessary.

A better approach therefore is to split up the algorithm into its most atomic parts and to allow referencing sub-algorithms in order to combine them:

1. Object Selection

The object selection algorithm snippet is given a set of groups and tasks. It applies a list of object filters in order to select the next element to consider. In case this element is a group, it selects the first matching group algorithm and applies it to the elements of the group. In case this element is a task, it selects the first matching task algorithm

and applies it to the task. Note that at this point you are free to choose any sub-algorithm, not only the Value Selection. For example, you may reference an *Algorithm Competition* (see below) in order to try different strategies and choose the best result.
The standard set of object filters currently includes:

- *[Groups/Tasks] First*
  As long as groups resp. tasks are left, this filter ignores tasks resp. groups.
- *Lexical Order [Ascending/Descending]*
  This filter chooses the task or group with smallest resp. greatest name w.r.t. lexical ordering.
- *[Maximum/Minimum] Priority [Max/Sum] OfElements*
  This filter chooses the task or group with greatest/least priority. [Max/Sum] specifies how to derive the priority of a group from its elements.
- *Model Filter* (ListOfAlternative[Name/Parameter]Filters selection)
  This filter lets pass only those tasks and groups, whose names match one of the name resp. parameter filters.
- *Conflicting Objects*
  This filter lets pass all tasks, which have conflicting timeline entries and all groups which contain at least one conflicting element.
- *First In Time*
  This filter analyses the time dependencies and lets pass those tasks and groups, which need to be scheduled next in order to build up the timeline entry graph from earliest to latest tasks.
- *Reject*
  This filter doesn't let pass any object.
- *Reject When All Parents Scheduled [At All/As Desired/Maximal]*
  This filter lets pass tasks and groups as long as at least one parent group of this object exists, which indicates that further scheduling is required.
- *Random Object*
  This filter selects a randomly chosen element from the remaining objects (a cryptographically strong random function is used here to assure that different planning runs result in different planning results).
- *Reject When Scheduling Fails*
  This filter doesn't let pass a task resp. group in case a previous try to schedule this element did not modify the timeline.
- *Reject When Considered Multiple Times* (int maximumNumberOfTimes)
  This filter doesn't let pass an element in case it has already been considered maximumNumberOfTimes.
- *Reject After Considering Multiple Elements* (int maximumNumberOfElements)
  This filter rejects all elements as soon as maximumNumberOfElements have been considered.
- *Omit Check For Existence Of RejectFilter*
  Usually not specifying a reject filter results in an endless loop. Therefore it is checked that there exists at least one reject filter. This check can be disabled using this 'filter'.
- *Tasks With Least Timeline Entries*
  From all tasks which are left to consider, this filter chooses those which have the least number of timeline entries.
- *Tasks With Minimum Scheduled Duration*
  From all tasks which are left to consider,this filter selects those whose sum of timeline entry duration is minimal.

2. *Ancestor Object Selection*

The ancestor object selection algorithm snippet is given a set of groups and tasks. These objects' ancestors are searched for matching groups of minimum distance to any of these considered objects. The resulting set of groups is handed over to the sub-algorithm.

The same object filters apply here as for the object selection algorithm, however only those applicable for groups are meaningful.

3. *Scheduled Task Selection*

The scheduled task selection allows selecting tasks which are scheduled, independently of the object hierarchy

4. *Value Selection*

The value selection algorithm snippet is given only one task and a timeline entry range to which the algorithm shall restrict its timeline entry search. It applies a list of value filters in order to select the most appropriate timeline

entry for this task. In case such a timeline entry exists, this timeline entry is added to the timeline. A sub-algorithm can be defined. This sub-algorithm will be executed on the task in case such a timeline entry has been created.

The standard set of value filters currently includes:

- *No Conflict*
  This filter lets pass all timeline entries which don't cause a conflict on a non-omitted constraint.
- *Project Horizon*
  This filter lets pass timeline entries which do not exceed the project horizon.
- *Time Selection*
  This filter restricts the timeline entries to certain time intervals.
- *[Earliest/Last] Start*
  This filter lets pass those timeline entries which have earliest resp. last start time.
- *[Min/Max] Duration*
  This filter lets pass those timeline entries which have min resp. max duration.
- *FirstInGrid*(Time gridBase, Duration gridSize)
  This filter lets pass all timeline entries whose start time is the earliest time inside the configured time grid.
- *Max Duration In Grid*
  This filter lets pass all timeline entries whose duration is equal to the maximum available duration in specified duration grid.
- *Existing Timeline Entries*
  This filter lets pass all timeline entries of this task which currently are defined
- *Conflicting Timeline Entries*
  This filter lets pass all timeline entries of this task which cause or contribute to a conflict
- *Max Duration Best Suitability / Multiple Assignments Max Suitability*
  These filters let pass the timeline entries which maximize a resource value during the timeline entry
- *Deconflict*
  This filter lets pass all timeline entries which may help to reduce the most severe conflict, which can be reduced by the considered task.
- *Reject*
  This filter doesn't let pass any timeline entry.
- Ongoing: *Resource Propagation*
  Depending on the resource usage and the estimated criticalities of the resources, this filter selects the timeline entry which is expected to cause the least restriction to other tasks.

5. *Locking*

With this algorithm, you can mark parts of timeline entries as being locked, i.e. they will not be removed from the timeline.

6. *Unschedule*

Given exactly one task, all or a specified number of timeline entries of this task, matching a specified list of value filters, are removed from the timeline. Locked parts of timeline entries will be omitted by the unschedule mechanism. A sub-algorithm may be specified in order to react on timeline entry removals.

7. *Algorithm Sequence*

The algorithm sequence allows executing multiple algorithms one by one, each continuing from the result of the preceding algorithm.

8. *Algorithm Competition*

The algorithm competition allows executing multiple algorithms independently in parallel. A list of valuation strategies is specified in order to determine the best result, which is taken over as the result of the algorithm competition call. In case multiple valuation strategies are defined, the n-th strategy is only applied to the best results according to the first n-1 strategies.

The standard set of valuation strategies currently include:

- *Conflict Evaluation*
  This strategy prefers the timeline with least conflicts.
- *Resource Evaluation*
  This strategy calculates the integral over a certain resource's profile. It prefers the timeline which maximizes this integral.

- *Resource Product Evaluation*
  This strategy calculates the integral over the product of two resource profiles, where
  (f*g)(t) := f(t)*g(t)
  It prefers the timeline which maximizes this integral.
- *Timeline Entry Evaluation*
  This strategy calculates the integral over a certain resource during certain tasks' timeline entries. It prefers the timeline which maximizes this integral.
- *Valuation Assembly*
  This strategy combines the results of other valuation strategies into one common value, using the following operations and functions:
  +, -, *, /, Average, Max, Min, Abs, Ceiling, Exp, Floor, Log, Log_e, Pow, Round, Sign, Truncate

9. Repair on Exit

This algorithm is not yet decomposed into its atomic snippets, and it is a good example of an algorithm, which grew too complex to configure. Nevertheless its functionality is very valuable, since it saves a lot of coding for a standard problem. The basic idea is the following (one of three use cases inside the TerraSAR-X/TanDEM-X scheduler):

An activity task 'payload-data-downlink' must be scheduled together with a preparation task 'ground antenna booked', e.g. the downlink from a satellite can only be scheduled in case the ground antenna is booked for this satellite. We cannot schedule the antenna booking first, because we do not yet know where the downlink can be scheduled. Besides we cannot schedule the downlink first, because we do not yet know where the antenna is available for booking.

The solution proposed by this algorithm is the following:

a) determine where the antenna may be booked
b) derive the times where the downlink may be scheduled according to a)
c) schedule the downlink, ignoring the constraint 'downlink needs antenna booking'
d) try to repair conflicts on the constraint 'downlink needs antenna booking' by extending existing or creating new timeline entries of the 'antenna booking' task
e) in case d) succeeds, accept, otherwise reject (here we use Platon's backtracking mechanism)

10. Static Scheduling Rules

This algorithm is not decomposed into its atomic snippets; however it proved to be useful in deriving timeline entries from ground station opportunities, which are already booked. We use this for scheduling start- and stop-times for housekeeping data transmission in the TerraSAR-X/TanDEM-X mission. Basically this algorithm searches for opportunity start- and stop-times (resource profile values) and adds offsets in order to generate timeline entries. Several parameters exist, which allow merging opportunities, ignoring short opportunities, avoiding and resolving conflicts etc.

11. Constraint Generation

New constraints can be defined by calling this algorithm.

Usually constraint generation should be finished before entering the main algorithm. However there exist situations where certain constraints shall be generated dynamically inside the algorithm. One example may be that a constraint should be considered if possible, so we define an algorithm competition with one algorithm defining this constraint and one not defining this constraint. In case we succeed with the algorithm defining the constraint, this one's result will be chosen, otherwise the other one's result will be chosen.

B. Combining algorithms

Most algorithms supply a sub-algorithm property which can be used to specify further algorithms to be executed 'inside' the current algorithm. In order to execute multiple algorithms one after another or in parallel, you can use the algorithm sequence resp. algorithm competition. Even if one needs some project specific sub-algorithm, the generic algorithm snippets can still be used, because you can reference the project specific code from the generic algorithm snippets.

The following pictures illustrate how a typical algorithm snippet configuration looks like, using the grid view of an XML editor:

Example: Object Selection

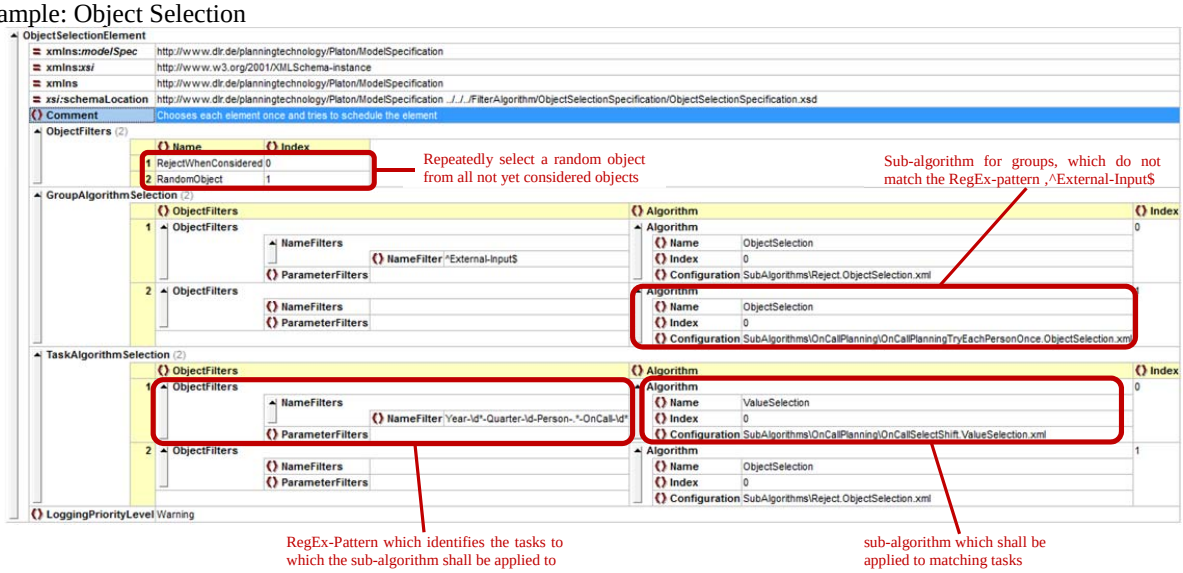

Figure 3. Example configuration file of the Object Selection algorithm snippet.

Example: Algorithm Sequence

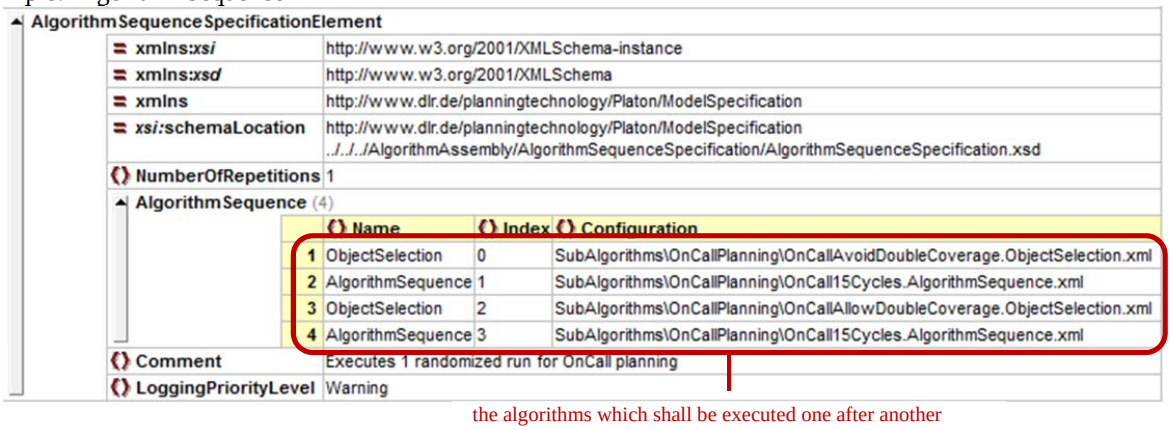

Figure 4. Example configuration file of the Algorithm Sequence algorithm snippet

As you can see, the configuration files of different algorithm snippets are completely different. In fact, each algorithm must define a dedicated XML schema, which defines what elements have to be defined in a configuration file of this algorithm. However these different XML schemata may share common elements. In the above examples, both algorithms allow to specify sub-algorithms. These are specified by a *Name* which identifies the algorithm, an *Index* which in the first case has no meaning and in the second case indicates the order of the algorithms, and a *Configuration* which points to a configuration file of the proper format for this algorithm. This pattern to specify an algorithm is defined by the *SequenceElement* mentioned above and may also be used for any other configurable item. In fact, it is also used for the *ObjectFilters* in the Object Selection example above, but as these two filters don't need configuration, only the *Name* and the *Index* are supplied.

In the following an example is shown that was implemented in order to schedule the on-call shifts for 2012 at GSOC. It depicts how to combine different algorithms:

Example 'On Call Planning'

The following pictures illustrate the OnCall planning algorithm:

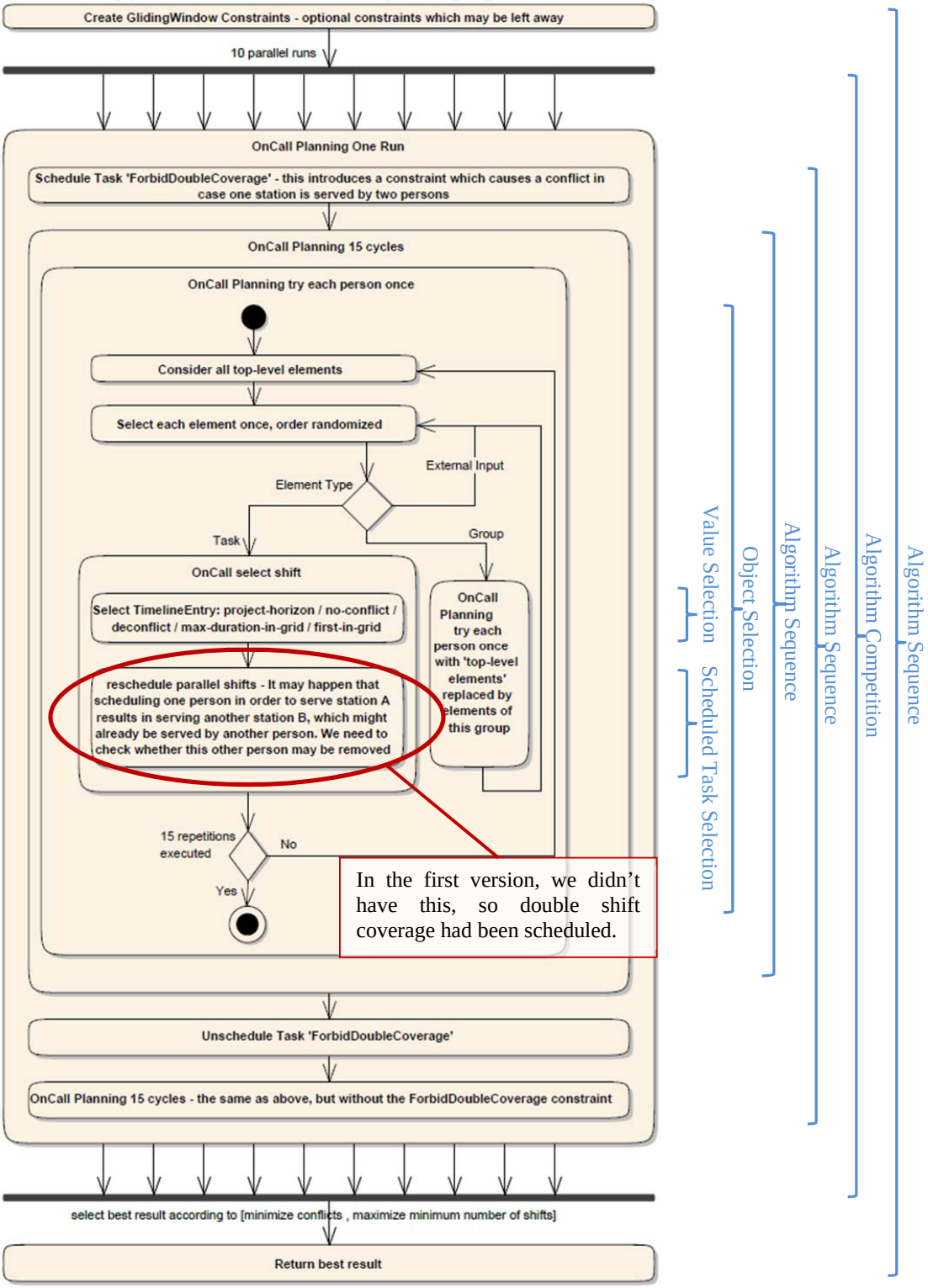

Figure 5. OnCall algorithm.

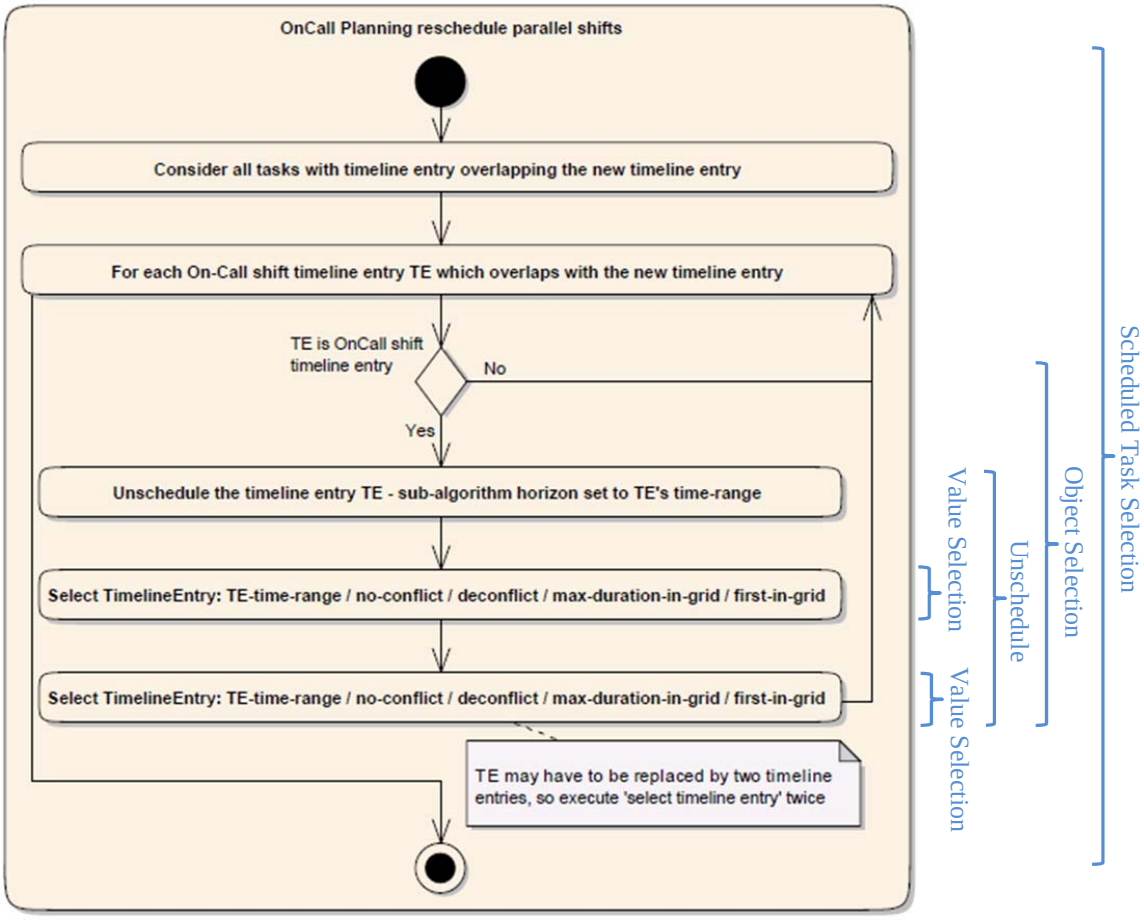

Figure 6. OnCall algorithm, reschedule parallel shifts

In this OnCall planning algorithm, we schedule the shifts of the staff by selecting all persons one by one and for each person we schedule one on call shift. As we will not succeed to fill the timeline by scheduling each person once, we repeat this 15 times.

In order to achieve a good result, we randomize the order in which we select the persons. We execute many (e.g. 100) runs and select the best result.

However we found that we scheduled unnecessary shifts. This happened as follows:

- Person A, who can serve the Power-Thermal station, has been scheduled from 1.3. – 14.3.
- Person B, who can serve both the Power-Thermal station and the Attitude and Control station, has been scheduled from 4.3. – 11.3., because of the Attitude and Control station.
- Obviously we have a superfluously scheduled shift from 4.3. - 11.3. for the Power-Thermal station.

In order to solve this problem, the straight forward solution would be that whenever we add a new on-call shift timeline entry, we need to check all parallel scheduled on-call shift timeline entries. For those timeline entries which are at least partly no longer useful, we need to un-schedule the superfluous parts.

Exactly this is what we have introduced by adding the step ‘reschedule parallel shifts’, which could be implemented just by inserting further configuration and referencing this as sub-algorithm inside the ‘OnCall select shift’ algorithm snippet. The step ‘remove superfluous parts’ has been implemented by

- un-schedule timeline entry in question (unschedule person A during 1.3. – 14.3.)
- schedule during unscheduled time range, where useful (schedule person A during 1.3. – 4.3.)
- schedule during unscheduled time range, where useful (schedule person A during 11.3. – 14.3.)

Remark: Similar to the *Create GlidingWindow constraints* steps in the OnCall Planning example you may introduce further pseudo-algorithms whose purpose is to modify the model. The benefit of this approach is that such a pseudo-algorithm can be incorporated into the overall algorithm. The backtracking support of Plato allows to roll

back even such model modifications in case the respective algorithm path is rejected later on (e.g. because an outer algorithm competition chooses a different result).

Conclusion

The main focus in the Pinta/Plato software has always been to find a good solution to our problems, without spending too much implementation effort in specialized solutions. However we noticed that most new problems incorporate new challenges which require further implementation and adaptation of the existing generic algorithms. With the new algorithm assembly set we put an end to modifying existing algorithms, because new features usually result in new algorithm snippets. However the main benefit is not the fact that we do not need to recompile the existing algorithms, but the ability to freely combine and reuse the snippets, which makes the system much more flexible. Additionally the complexity of the snippets is reduced dramatically and so not only the Plato experts may understand the algorithm but also other less experienced developers and users.

References

[1]Lenzen, Mrowka, Spörl, Klaehn, "Scheduling Formatins and Constellations", *SpaceOps 2010 Conference*, AIAA-2010-2002