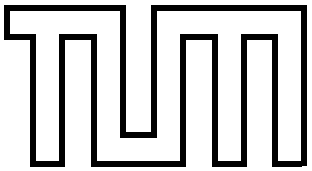


Technische Universtät München

Fakultät für Informatik

Diplomarbeit

A Modular Graphical User Interface for Interactive Planning

Autor: Daniel Weiss

Aufgabensteller: Univ.-Prof. Dr. Dr. h.c. Wilfried Brauer

Betreuer: Dr. Alexander Nareyek

Abgabedatum: 14.06.2004

Abstract

This diploma thesis deals with the development of a modular graphical user interface for the EXCALIBUR planning system. First the reader finds a short introduction about Action Planning and the concept of the EXCALIBUR-engine. Additionally, important properties for the graphical user interface are selected, and the goals of this thesis are defined. Different approaches of data visualization are gathered, and an own visualization technique is created.

The study describes the software engineering process of the graphical user interface, named the Plan-Presentation. First of all, the users had to be categorized corresponding to their background knowledge of artificial intelligence and of EXCALIBUR (Chapter *2.Requirements Elicitation*). Furthermore, the whole functionality is defined as use cases, and information about interface design is gathered and applied to the future graphical user interface. The objects and classes from the use cases and the existing data model of the EXCALIBUR-engine are all identified (Chapter *3. Object Analysis*).

Another important point is the discussion about architectural patterns and the decision for one of them as the basis for the system architecture (Chapter *4.System Components*). In the following, the implementation of the graphical user interface and all plugins are explained in detail (Chapter *5. Object Design and Implementation*).

Finally, an evaluation of the project is made. For that purpose, complications arisen during the implementation are listed, and it is explained how they were solved. Furthermore, the reasons for divergences from the specification are described. On the other hand, the targets fulfilled are rated. After all, it is described what the future work at the Plan-Presentation will be (Chapter *6. Evaluation*).

In the appendix, a *User and Developer Manual* helps new users working with the graphical user interface. Additionally, it explains developers how they can use the Plan-Presentation in their own EXCALIBUR planning scenario file. The document ends with a detailed *Plugin-Programming HOWTO* for developers who create a new plugin for the Plan-Presentation.

Contents

1. Introduction

This thesis describes the development of a graphical user interface, the Plan-Presentation application, based on the EXCALIBUR planning system. The user interface to be developed enables visualization and manipulation of a plan, created by the EXCALIBUR planning system, on the screen.

The intended audience of this thesis are readers with AI background knowledge who are interested in details about the development process of the graphical user interface for the EXCALIBUR engine. A short introduction about the EXCALIBUR concept is given, but it is recommended that readers interested in the concept of EXCALIBUR should study [Nar01] and [Nar00].

This chapter gives first an introduction of Action Planning systems in general and of the EXCALIBUR system in particular. After this, requirements for Real-World planning are gathered, and the goals of the graphical user interface and of this thesis are defined.

Furthermore, different approaches for the interactive visualization of plans are explained and one of them is chosen for the visualization of EXCALIBUR plans.

1.1. Action Planning

Planning systems choose, from a given set of predefined action types, the required ones, instantiate them to actions and align the actions so that the predefined goals are reached.

An agent is controlled by a planning system and is, for example, a robot or a visualization of a person in a virtual environment. However, there are many definitions for agents, for example, a program crawling the web and searching something for the user with AI technique can also be an agent. But for imagination purposes in this document, it is reasonable to use the vision of a robot. If such an agent wants to go through a doorway, it first has to open the door. For that purpose, an action is needed. An action has a precondition which tests if the action can be executed. For example, the action "open the door" has a *precondition* which tests if the door is closed at present. If the door is already open, this action cannot be applied. Additionally, the action has an *effect* representing the impact on the environment and in our example the effect is the changing of the door state from "closed" to "open".

The goals of the agent determine the targets of the planning process. The process was successful, if all goals were reached. A goal can be a state like "the door must be opened" or a numerical target like "get as much money as possible". The task of a planning system is to find the needed actions and to align them into the correct order. A more detailed overview about planning systems is given in "Artificial Intelligence : A Modern Approach." [NS03].

The base of this thesis is the EXCALIBUR-planning engine, which is built on top of the DragonBreath Constraint-Solver. Both systems are developed by Alexander Nareyek. In the following sections a short explanation about them is given.

1.1.1. The DragonBreath Engine

The DragonBreath engine is the base for the EXCALIBUR planning system, but it is not a planning system itself, so there is no planning context in this section.

The DragonBreath application is an optimization engine based on constraint programming and local search.

A cite of Dr. Roman Barták [Bar03] explains constraint programming: *Constraint programming is a technology for declarative description and solving of hard combinatorial problems. The user just states the constraints over the problem variables and the system finds a valuation of the variables satisfying the constraints.* A variable contains a value which is normally restricted by one or more constraints. A constraint can also restrict one or more variables at the same time.

Local search is an approach to find an optimal assignment by making local improvements. In the case of the DragonBreath engine this means if a constraint is violated, this constraint tries to find a better solution for its connected variables. This has, of course, an impact on the other constraints and might hurt them. However, this approach enables iterative improvements of the whole constraint-satisfaction-graph.

In practice, this approach is applied by the DRAGONRBEATH as follows:
The data structure of the DragonBreath engine consists of variable- and constraint-nodes in a graph. The variable-nodes are connected to one or more constraint-nodes.

The EQUAL-constraint[1] α has, for example, two variable-nodes A and B connected to it. If the value of A and B is not equal, the EQUAL-constraint α causes costs at the amount of $|A - B|$. As for example, $A = 5$ and $B = 3$ causes costs in the EQUAL-constraint amounting to 2. The constraint solver, named Global-Search-Control, would now request the EQUAL-constraint to improve itself. Therefore, the constraint sets the variable B to the value 5 and the costs are zero.

If there are many EQUAL-constraints in the same graph, the Global-Search-Control would choose the constraint with the highest costs and calls its improvement function. In our example the first improvement call sets the value of $B = 5$, this perhaps hurts the EQUAL-constraint β, which is connected to B and C.

The constraint-satisfaction-problem is solved, when all costs are eliminated. In the given example, this would be when $A = B = C$.

1.1.2. The EXCALIBUR Agent

The EXCALIBUR architecture uses the DragonBreath constraint-solver and adds special constraint types for planning purposes. Instead of the EQUAL-constraint higher level constraints like, for example, a State-Task-Object-Constraint, are used in the constraint graph, which describes the effect of opening the door. A set of variable-nodes, like the beginning-time, the duration, etc., are connected to this constraint which specify when the State-Task is executed.

The planning process is very complex, but in short words: The planning is realized by the insertion of constraints and variables which represent an action (e.g. Action-Object-Constraint) and some additional nodes (variables and other constraints) into the graph.
The following simple planning step is given as an example to show how the planning could take place.
The planning step begins when the State-Resource-Constraint causes costs, because a goal[2]

[1] In fact, the EQUAL-constraint is a SUM-constraint with one addend and one outcome.

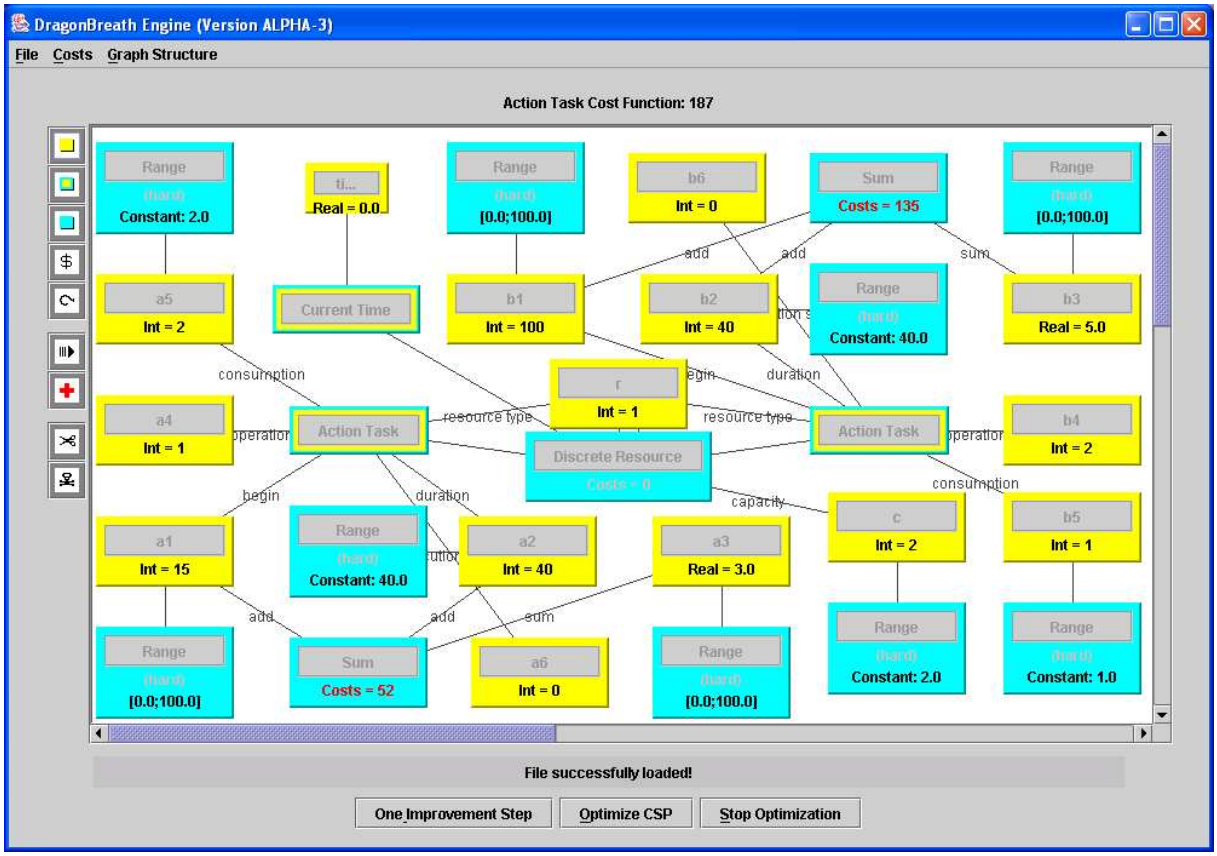


Figure 1.1.: The DragonBreath Engine representing two Action-Tasks. The yellow boxes represent variable-nodes, while the turquoise nodes are constraints.

connected to it is not fulfilled. For example, the state is ”door is closed” and the goal requires the state ”door is open”. Then the State-Resource-Constraint inserts a new State-Task node ”open the door” into the planning-graph so that the effect of this State-Task fulfills the goal hurt. However, this lonely State-Task causes structural errors, because no Action-Object-Constraint is connected to it, which is required by the specification. Therefore, the planning system inserts the missing Action-Object-Constraint and its corresponding additional nodes. One of the additional nodes is perhaps the precondition ”be in front of the door”, which is probably not fulfilled and the plan creation continues with this precondition.

The complete description of the planning process is available at [Nar00].

In fact the planning process is much more complex. There are the following constraint types:

- The precondition mentioned above is represented by a *Precondition-Task-Object-Constraint* and additional variables. These variables are the variable *temporal reference*, which describes the time, when the precondition must be fulfilled, and the variable *resource type*[3] to which this precondition refers and the variable *state* in which the resource must be so that the precondition is fulfilled.
  However, to every constraint a lot of variables are connected and the description of every variable in detail is not part of this thesis, which deals with the development of a graphical user interface. The papers about the data structure and the planning process are available at [Nar00] and [Nar01]. The description of the constraints as for now focuses only on the main conceptual parts of a constraint, but it should be remembered that every constraint

[3]See the next point ”State-Resource-Constraint” in the listing for information about resources.

is in a group of variables and other constraints.

- The *State-Resource-Constraint* represents a resource in the world. A resource can be a fuel level or the state of an open/closed door. The state is also projected over the time, that means the State-Resource-Constraint assumes the state of the resource for every tick in the time. This is needed so that the agent can plan into the future. For example, when the agent plans to open a closed door at the time tick 1000, the information is accessible that before tick 1000 the state of the door is closed and after tick 1000 the state of the door is open.

- The *State-Task-Object-Constraint* represents a contribution to a State-Resource-Constraint. That means a state task "add one liter" adds one liter of fuel into the tank. In context with the "open/closed door"-State-Resource-Constraint, a contribution can also be the state task "close the door", which changes the state of the door to "closed".

- While the State-Resource-Constraint is visible to all actors in the world, the *Action-Resource-Constraint*[4] represents a resource, which can be used only by the agent itself. An Action-Resource-Constraint is, for example, his arm or his leg. This resource is not usable for other agents in the world. The planning what an agent has to do takes place on this constraint.

- The *Action-Task-Object-Constraint* represents an allocation of an Action-Resource-Constraint at a given timespan. For example, while the agent is walking, the left leg is occupied with moving forward and backward. A visual example, what two Action-Task-Object-Constraints look like is visible in Figure 1.1.

- All the constraints ending with "Task-Object-Constraint" are representing a part of an action, are grouped around an *Action-Object-Constraint*. This means that one action (in the EXCALIBUR terminology) can consist of multiple single Action-Tasks, Precondition-Tasks and State-Tasks. If all connected preconditions are fulfilled, then all State-Tasks and Action-Tasks are executed. If one of the preconditions is not fulfilled, no task is executed.

- The *Transition-Constraint* represents external effects. It is very similar to the Action-Object-Constraint, but does not have any Action-Task-Object-Constraints connected to it. For example, if a washbasin contains too much water (the precondition), it would overflow and the floor would get wet (the State-Task). Transitions are always executed, if all preconditions are fulfilled. The agent does not decide which transition should be executed and which not.

1.2. Real-World Requirements for Planning

Planning systems that are used for a real world environment must fulfill a number of requirements:

- The planning system has not the time to find an optimal plan. For example, when driving a car, often a fast evasion manoeuvre is necessary. Therefore, *real-time reasoning* is required. The agent controlled by the EXCALIBUR planning system creates, in a short time, a simple plan which is incrementally improved. When more computing time is available, it computes a more optimal plan.

[4]The present name of the Action-Resource-Constraint is Discrete-Resource-Constraint and will be changed soon.

- *Changing sensor data* is another problem. If the sensor data change, the whole plan must be recomputed. The local-search approach enables the EXCALIBUR agent to make only minor changes to adapt the plan to the changes, if possible.
- The real world is not closed. It means that the agent can never know all entities in the world. There can be a person who comes through the entrance door and the agent has never seen it before. This is expressed as the *open-world assumption* or *incomplete knowledge*. However, the EXCALIBUR engine supports the adding of new resources during the planning process. For that purpose, only a State-Task-Constraint must be included into the graph.

These requirements are all supported by the EXCALIBUR engine. However, testing EXCALIBUR in a real world environment would be very expensive because of the required robotic elements. Fortunately, the application domain of today's third-dimension real time computer games is very similar to the real world so that the features of the EXCALIBUR agents will be demonstrated in this environment.

An easily to understand and concise graphical user interface is very helpful for the development of a planning system for a certain application domain (defining actions, preconditions, etc.). Additionally, it shall be used to support future development of the EXCALIBUR engine by admitting debugging of the plan. Another application domain is the usage of the Plan-Presentation as interactive planning interface for industry purposes, where the user observes the planning process and adjusts the plan, if necessary.

1.3. Important Properties for a Planning GUI

The important properties that are necessary for a planning GUI can be divided into features which are responsible for showing the plan and features to change the plan. Additionally, the user types can also be divided in:

- AI experts with knowledge of the EXCALIBUR engine and Java, who develop future planning scenarios or implement new AI techniques into the engine.
- The second user type are AI experts without special knowledge of EXCALIBUR. They know AI concepts like preconditions and actions. The Plan-Presentation shall be designed also for these users and must be therefore very intuitive.

At first, for showing the plan the graphical user interface should support:

1. An abstract, high level view of the whole planning process, which is easily to comprehend.
2. Additionally, a more detailed view is necessary enabling an expert to get all necessary data of the planning process to understand what is happening exactly.
3. Plans can be short, but they can also have a very long timespan. To observe all kinds of plans it is advantageously to have some sort of zooming available.
4. While complex problems take longer to compute and simple problems often not, therefore the planning speed control shall influence how fast the information appear on the screen.
5. Additionally, it shall be possible to use breakpoints to visualize complete improvement cycles or only atomic changes in the DragonBreath graph and everything between these two extremes.
6. Changes in the planning internals, due to the improvement of the plan, must be immediately displayed in the graphical user interface.

The next paragraph describes the basic functionality for changing the plan:

1. The graphical user interface should support the simulation of an open environment by the possibility to change sensor data.
2. For debugging purposes, the agent should be influenceable by changing its goals and by adding possible actions to his plan or by editing them.
3. All changes made in the planning interface must immediately be realized in the data model.

1.4. Goals of this Thesis

As the previous graphical user interface of the DragonBreath engine (see Figure 1.1) does not support the requested features listed in 1.3, it is the goal of this thesis to develop a new graphical user interface for Excalibur agents.

Besides all of the above mentioned features, which are also the goals of this thesis, there are many software engineering challenges:

- There should be as few dependencies from the Excalibur engine to the GUI as possible so that the engine can be started without the GUI.
- The GUI must be extremely extensible, because at present there are not all planning concepts implemented in the Excalibur engine and soon new features, like objects, will be added.
- Bi-directional update propagation that means, changes in the planning internals must immediately be reflected in the graphics, and changes at the graphical level must directly be realized by way of the planning internals.

1.5. Approaches

At first, the best graphical representation for plans must be found. For that purposes, different approaches of plan visualization are asserted in context to the Excalibur engine. The examples in the literature range from satellite scheduling to medical therapy plans in the third dimension.

1.5.1. Skeleton Plans

One approach is to visualize the plan as a graph, where two actions are linked, if one of the actions has to be executed before the other. This is done at SIPE-2 (System for Interactive planning and Execution) [WA96].

This technique shows best the relation between the actions and gives an expert a very detailed view ”why” an action is needed. Unfortunately, the Excalibur data model does not contain the information, which action must happen before another action. It would be also very expensive to compute this.

1.5.2. Plan Hierarchies

Taking into consideration the third-dimension gives an additional possibility to show more data intuitively. The SIPE-2 technology supports hierarchical decomposition planning, this means

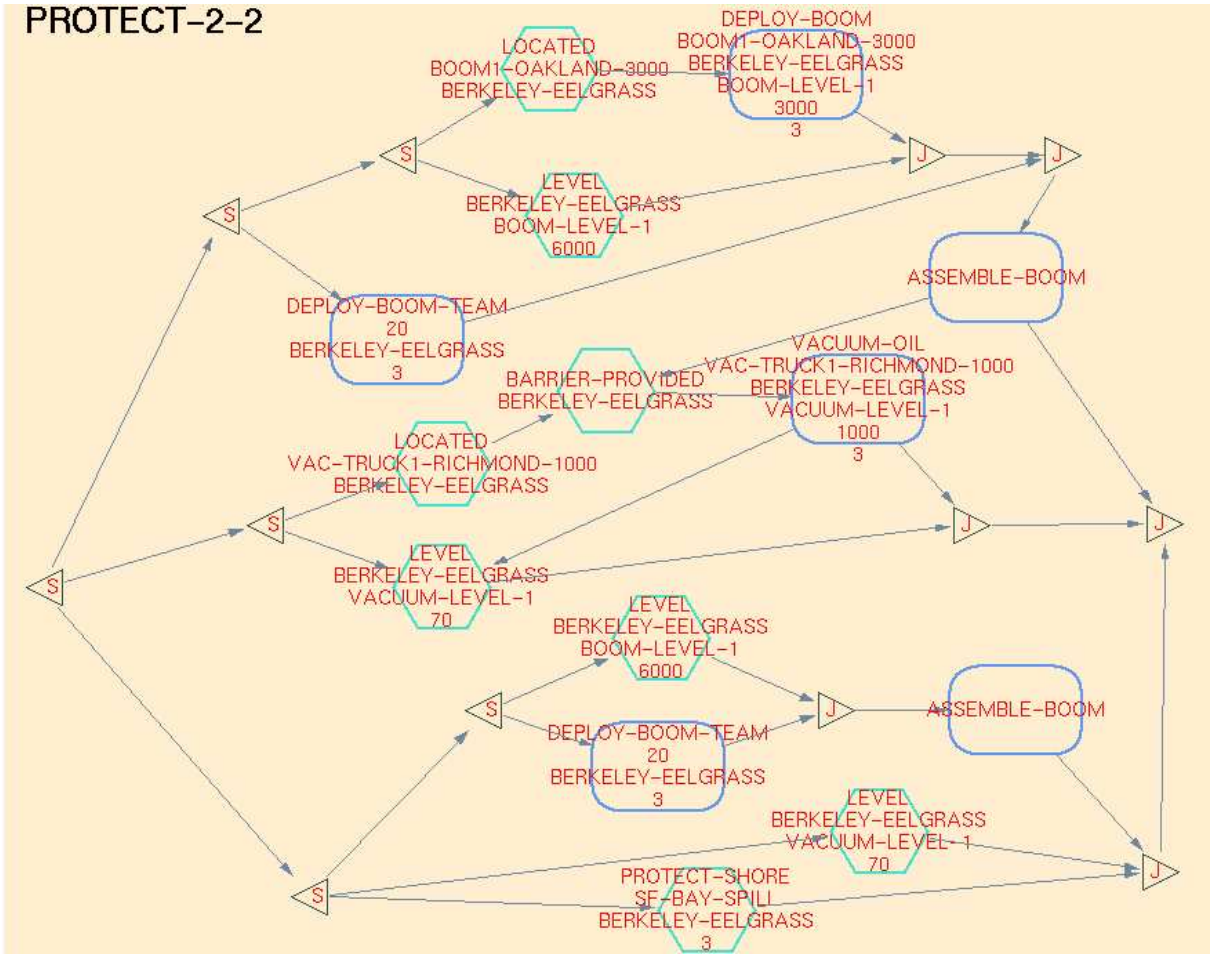


Figure 1.2.: SIPE-2 Oil Spill Response Planning.

one large action represents many smaller actions. A large action is represented as a three dimensional box, where many smaller boxes are inside or above (see Figure 1.3). The user can move through the 3D world as in a flight simulator. If he crosses the border of a 3D box, he sees the actions inside of this box. This concept is developed by Kishalay Kundu and Chad Sessions (see [KSdR02]).

However, the planning process is very dynamical so that it would be very difficult for the user to orientate, while the plan and the 3D environment is changing fast. While EXCALIBUR is supporting hierarchical decomposition planning (see [Nar02]), it is not the focus of the application and not interesting in this context.

1.5.3. Timeline Approaches

Timelines, plots where the x-axis variable is time, are a very effective method of displaying trends, building on the human perception of time as linear and taking advantage of the human ability to infer closeness of relation of two items from their spatial proximity[MYWA02].

Shinkansen Graphical-Timetable

The graphical timetable of the Shinkansen [Lam99] train makes the advantage of an graphical representation clear in comparison with tabulated numerical data. The timeline is the x-axis

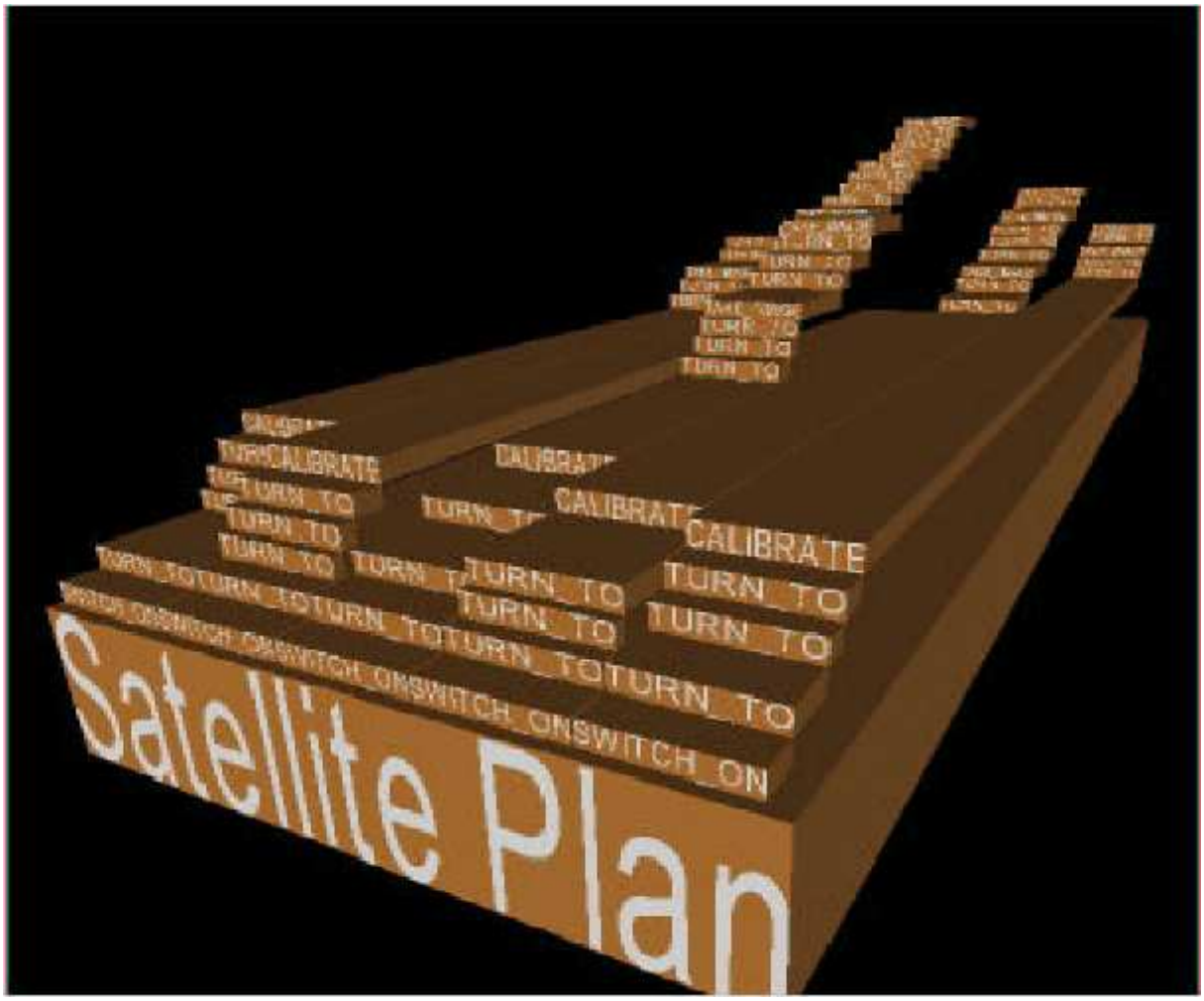


Figure 1.3.: Hierarchical decomposition viewer for satellite scheduling in the third dimension.

and the stations are listed on the y-axis. A train going into a certain direction is represented by a diagonal line from the upper left to the lower right, and in the opposite direction it is represented by a diagonal line from the lower left to the upper right.

This graphical representation is very good for some-order plans, where some actions have an order, that means they must be after another actions, and some do not, where an action is independent from other actions. In this case, actions having required other actions before, for example, as driving from station B to C requires first the action ”driving from A to B”, are visualized by a line. Whereas two actions without any order, for example, the train at 15 h driving from A to B has no visual connection to the train driving at 21 h from A to B.

However, this approach leaks in the visualization of different actions and actions having no ordering. In this case, all lines are trains, but at different times, whereas in the Excalibur planning system, actions can be completely different. For example, one action represents the walking from one place to another, while another action represents the manufacturing of something. Furthermore, there is no order of actions available in the Excalibur planning system, while in the Shinkansen approach it is toughly defined that before driving the track B to C the train must have driven track A to B. The result would be that there are no long lines, but instead a lot of short lines in the visualization. Additionally, at the Excalibur-engine the same action can be in parallel, while the contribution of a railtrack is always exclusive.

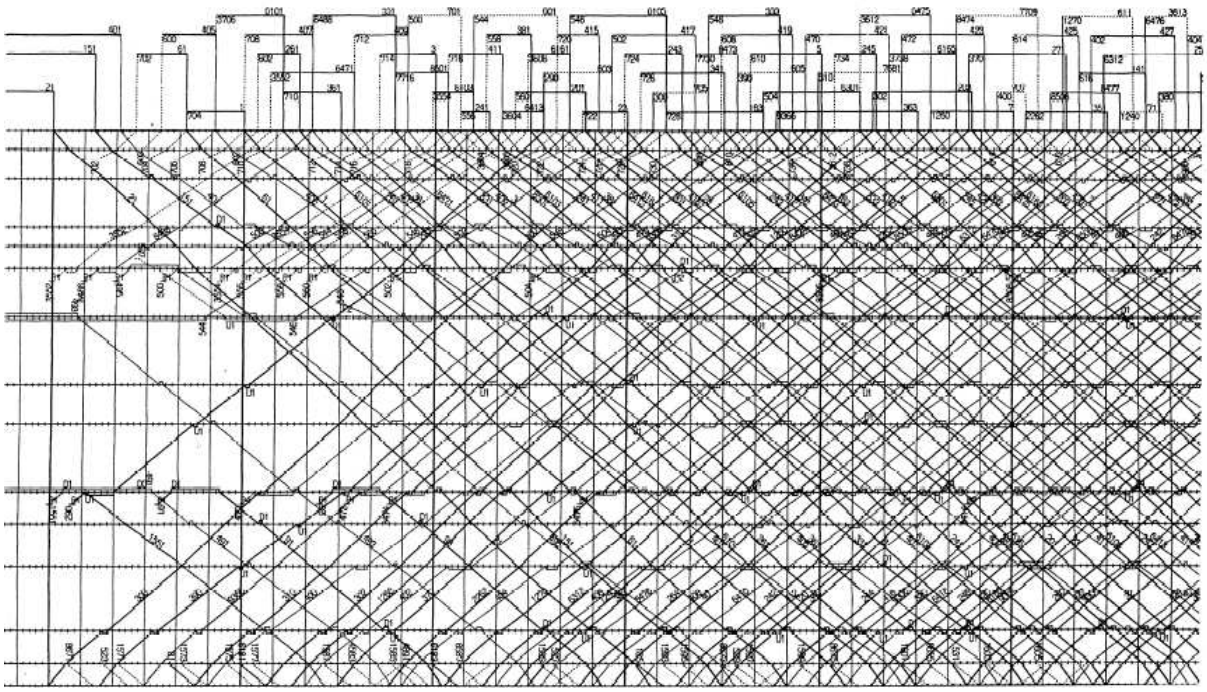


Figure 1.4.: The Shinkansen Graphical-Timetable.

Medical Therapy 3D View

Another approach is a complex 3D view of a medical therapy plan (see [KM99] and Figure 1.5). It was developed by Robert Kosara and Silvia Miksch. There can be many actions in parallel and the preconditions are expressed as hurdles.

In a variation of this approach, another interesting feature is the usage of known metaphors like traffic lights for preconditions. This helps the reader of the plan to understand the semantic very fast.

However, in our planning process we have a lot of preconditions which check a completely different resource from that which the action will change. So it is difficult to express which precondition checks which resource. Setting a fixed color for every resource is not possible, because there are many resources and the user should normally see only some of them. The problem is that the color of one resource should always be the same, but this is not possible because if the resource is not in the view area (because it is scrolled away) and the color is given to another, now visible, resource. The alternative would be to give every action another color, which is fixed for ever. Than due to the big amount of actions, the screen would be very colorful and two colors of different action would differ only at a minimal color nuance. Therefore, it is reasonable to use fewer colors, but in a more semantic manner. This view technique transmits a lot of information, but it needs some time to understand the symbols and the representation. A more simple type of view is required.

Gantt Chart

Another approach is the visualization of actions as simple boxes. They are aligned on a timeline, the x-axis. Actions that are in parallel to other actions are shifted on the y-axis. This is very similar to a Gantt chart. A Gantt chart is a diagram used in project management and scheduling, where the x-axis is the time and the y-axis shows tasks to be performed to complete the project.

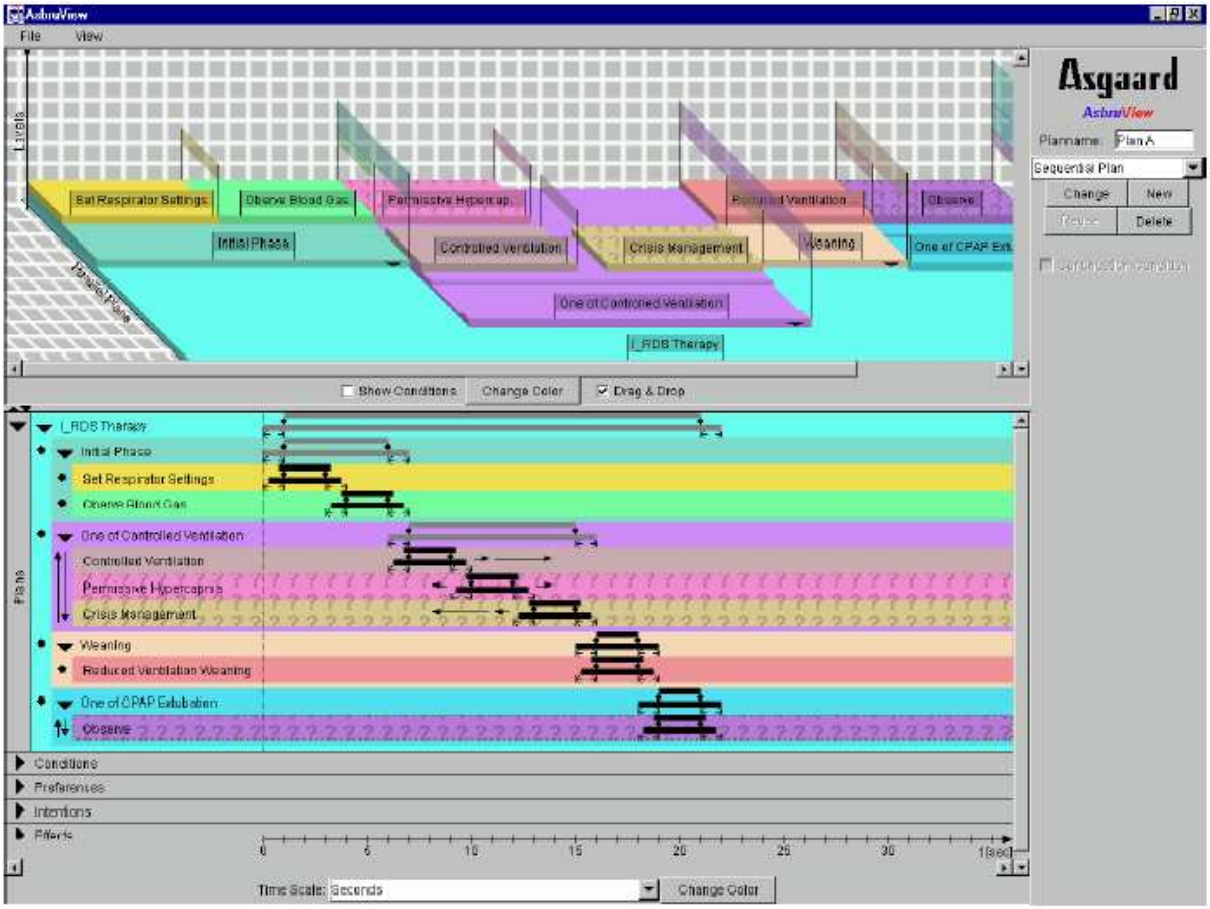


Figure 1.5.: The medical therapy 3D view developed by Robert Kosara and Silvia Miksch.

Every task is displayed as a horizontal bar spanning the time period during which it is expected to take place. The Gantt chart was developed by Charles Gantt in 1917. It is widely spread (for example in Microsoft Project) and therefore most users, especially in the scheduling application domain, feel comfortable with it. Additionally, this is complaisant to the EXCALIBUR data structure, because the actions inside the data model are aligned also to a timeline and all necessary data are available.

Visualization Decision

In contrast to the other visualized views, the Gantt chart is simpler and less specialized and therefore the user can recognize changes in the plan faster. This is the reason why an variation of a Gantt chart is used for the graphical user interface.

However, the Gantt chart is still simplified, because it is not displayed which action must be before another. This information is not available in the data model.

Of course, there is also additional information to be visualized, like the state of a resource or if an action can be really executed. The states of a resource are intervals with a beginning and an ending time. This is similar to actions and will be also represented like them in a separate area, with one exception: no shifting on the y-axis is required, because the resource can only be in one state at a time.

Additionally, coloring will be used to show, whether an action can be really executed. For that purpose, there are traffic lights showing the color green for an executable action and a red color

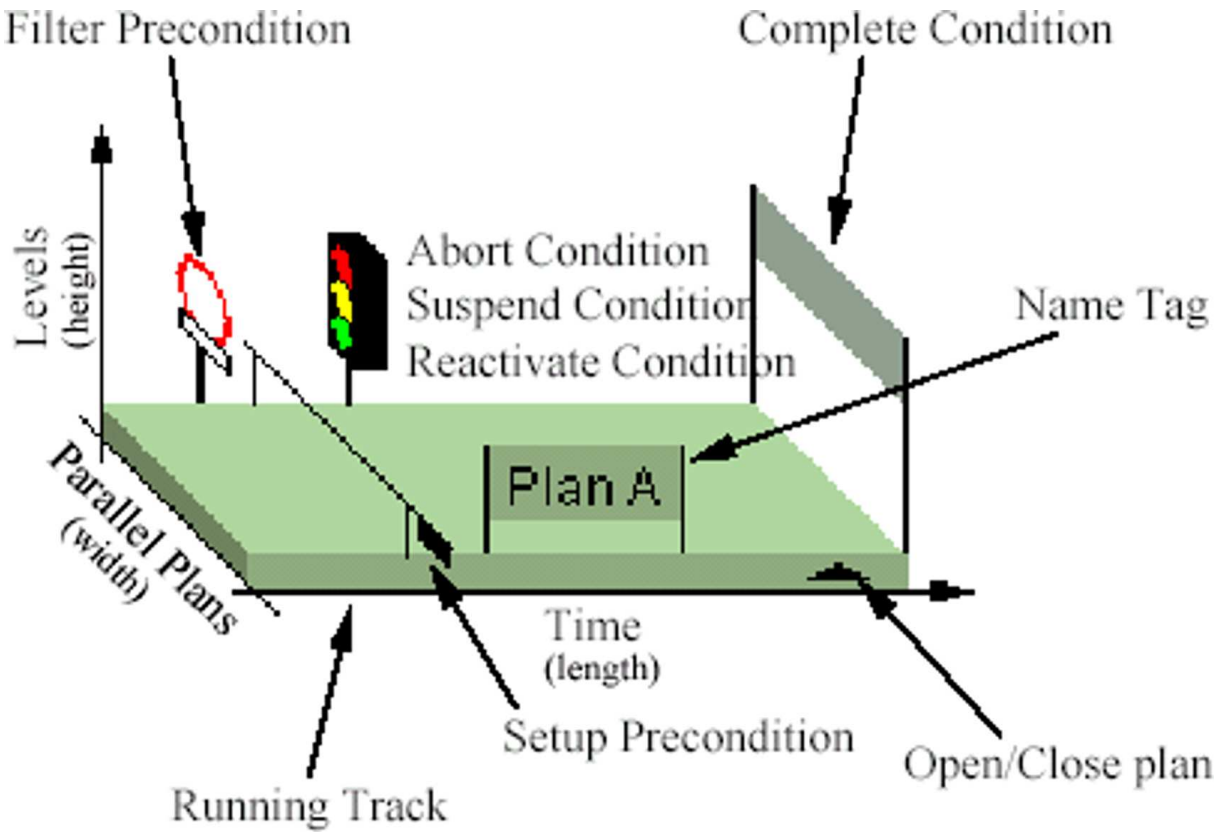


Figure 1.6.: A traffic light indicates if the precondition for the next therapy step is fulfilled.

for an action, where problems occur, which is very intuitive. The idea is taken from [KM99].

To show the relation between actions and the used resources, a line is painted between them. More exactly there is a small rectangle which represents the State-Task which contributes to this resource below the states view of the resource.

This technique enables also the other features demanded to be realized. The abstract view is possible, if only the actions are displayed. If an expert needs more information, he can additionally expand a detailed view and look what exactly happens to a resource.

The handling is also very intuitive because of the timeline approach. Scrolling to the right means viewing the plan at a later point of time. The zoom slider in context with the scrolling function allows to view a small part of the plan or the whole plan on screen.

1.6. Overview

The thesis will document the whole software engineering process of the Plan-Presentation application:

The elicitation of requirements (Chapter *2.Requirements Elicitation*) detecting which objects and classes are necessary (Chapter *3.Object Analysis*) and describing the creation of a system architecture (Chapter *4.System Components*).

Chapter *5 Object Design and Implementation* describes the objects in detail and explains the difficulties during the implementation process.

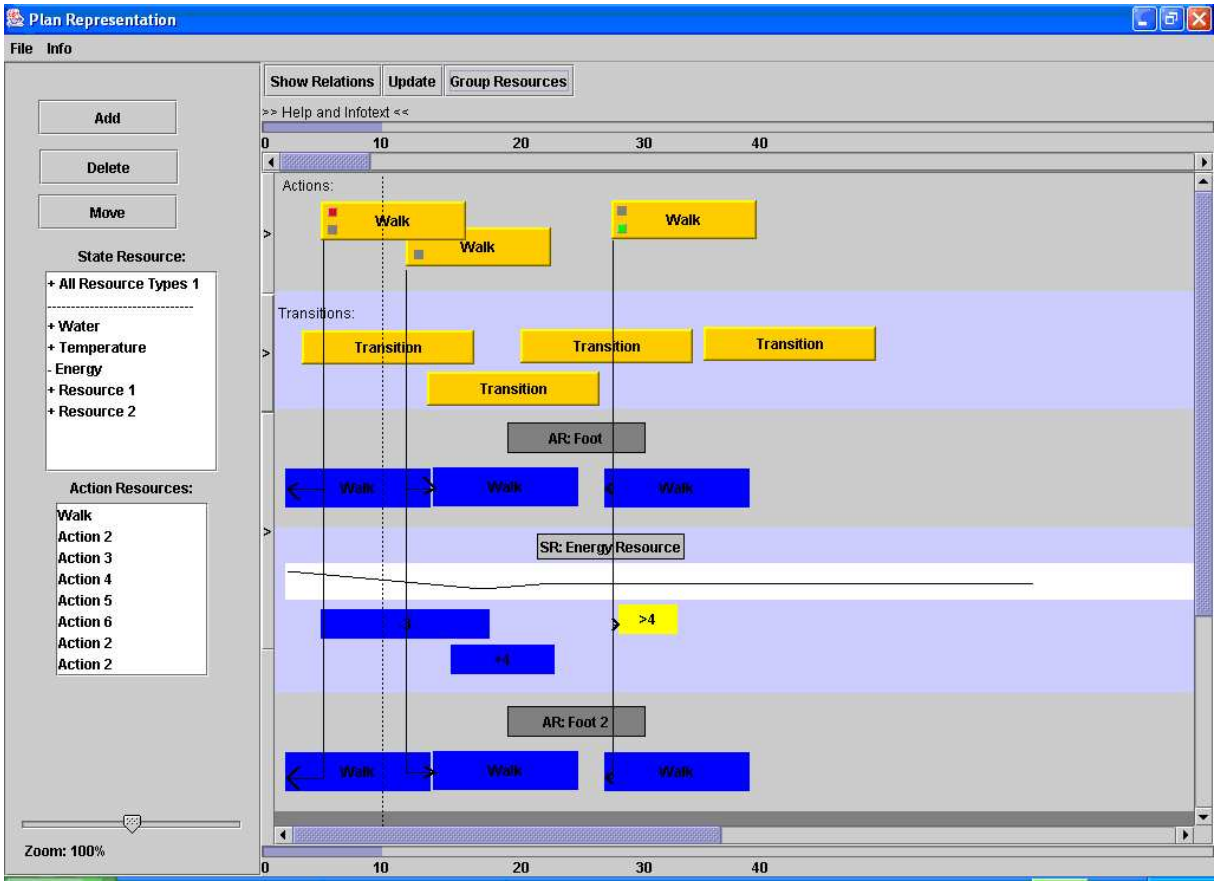


Figure 1.7.: The prototype of the Plan-Presentation window.

The last Chapter *6 Evaluation* assesses the implementation and the specification. While using the application few complications arise, which were not predictable in the specification process.

In the appendix, there is a *User and Developer Manual* and a special HOWTO for plugin programmers.

The structure of the software engineering process is based on [BD00].

2. Requirements Elicitation

The following four chapters, from *2. Requirements Elicitation* to *5. Object Design and Implementation*, deal with the software engineering process building the Plan-Presentation graphical user interface. This detailed description is particularly for readers interested in developing an extension for the Plan-Presentation or readers interested in concepts about observing and visualization of complex data structures. The software engineering process is orientated towards the book "Object-Oriented Software Engineering" [BD00].

In this chapter, the actors are categorized so that an estimation of the background knowledge of graphical user interfaces and the application domain is possible. Then the use cases define the whole functionality of the Plan-Presentation application. After this, a short introduction into interface design is given and the derivations for the Plan-Presentation graphical user interface are explained. At last, nonfunctional requirements are identified. These nonfunctional requirements describe specifications, which are not directly visible to the end-user, but are challenges in the development process.

2.1. Identifying Actors

The identifying of the actor types is necessary to gather information about their background knowledge of graphical user interfaces and planning systems. This supports the later decision what elements in the user interface are necessary. For example, if a step-by-step dialog, a wizard or only shortcuts are needed.

It is to assume that all users working with this application have expert skills as end-user and know concepts like drag&drop, scrolling and zooming.

To create a scenario and define which action types are available an AI expert with deep knowledge of the Excalibur-engine and Java is necessary. However, defining new action types is not part of the Plan-Presentation application and not part of this thesis.

The second user type is a computer scientist with AI knowledge, who is familiar with preconditions, actions and effects. At first, during the orientation phase, the abstract view of actions and later the detailed view is interesting.

Another user type is a computer scientist working in a production environment, observing the planning process and adjusting the plan. This user type is similar to the AI expert without deep knowledge of the Excalibur concept.

Abstracting these points leads to the following categories:

- AI experts with knowledge of the Excalibur engine and Java, who develop future planning scenarios or implement new AI techniques into the engine.
- The second user type is an AI expert without special knowledge of Excalibur. He knows AI concepts like preconditions and actions. The Plan-Presentation shall be designed also for these users and must therefore be very intuitive.

Other actors like machines do not exist, because the application does not communicate with other servers or programs.

2.2. Identifying Use Cases

The following use cases define the whole functionality of the application. The use cases were defined in iterative improvement cycles between Dr. Nareyek and the author of this thesis. They are based on the requirements defined in Chapter *2 Requirements Elicitation*.

For every section, it is to assume that the Plan-Presentation has already been started. This means the main program with the planning-scenario definition is running and has instantiated the Plan-Presentation graphical user interface, which after that is visible as a window on the screen.

The following sections contain a very detailed description of the functionality. The description is based on the use case template in [BD00] page 111.

2.2.1. Viewing the Plan

- *Participating actors*: All
- *Entry condition*: The application has been started.
- *Flow of events*:
  1. The user has started the Plan-Presentation application and sees the planning window in the pause mode. There are no actions visible.
  2. The user presses the play button so the engine tries to solve the planning problem. New actions are shown on the screen and disappear again, if the planning engine decides that this action was wrong.
  3. If the planner is in the online mode, the user sees a dashed vertical line moving from left to right. This is the current time for the agent.
  4. The user can scroll to the left or right to see the plan at an earlier or later point of time.
  5. Optionally, he can use a zoom slider to see a bigger or smaller clipping of the plan on the screen.
- *Exit condition*: Clicking on the menu item $File -> Exit$.
- *Special requirements*: None

2.2.2. Viewing a Resource

- *Participating actors*: AI expert
- *Entry condition*: There are resources available in the overview lists.
- *Flow of events*:
  1. The user clicks either on an Action-Resource or a State-Resource (see Chapter *1.1.2.* EXCALIBUR-*Agent* for details about the technical terms of the planning system) in

one of the overview lists. A State-Resource can be a Numerical-Heap-State-Resource or a Symbolic-State-Resource.

2. An additional view[1] appears in the main window. Depending on what he clicked, the content of the view is different.
3. If there are too many views for the screen height in the main window, a vertical scroll bar on the right side appears to scroll to the desired view.
4. A user no longer interested in this view can remove it by clicking again on the resource in one of the overview lists.

- *Exit condition*: Clicking again on the view in the overview lists.
- *Special requirements*: The Symbolic-State-Resource and the Numerical-Heap-State-Resource are both listed in the State-Resource overview list.

2.2.3. Retracting a View

- *Participating actors*: AI expert
- *Entry condition*: A view is expanded.
- *Flow of events*:
  1. If the user presses a special "retract button", the view height gets very narrow.
  2. Now the view is masked and only the view's name appears on the narrowed view.
- *Exit condition*: If the user clicks again on the narrowed view, it will expand. The second possibility is to click again on the resource in the overview list, then the complete view will disappear.
- *Special requirements*: None.

2.2.4. Showing Transitions

- *Participating actors*: AI expert
- *Entry condition*: Clicking on the menu entry $View -> Show\ Transitions$
- *Flow of events*: The transition view is added to the main window. The usage is similar to the action view.
- *Exit condition*: Clicking on the menu entry $View -> Hide\ Transitions$
- *Special requirements*: None

2.2.5. Activating Swimlanes

- *Participating actors*: AI expert
- *Entry condition*: There are at least two views visible (the action view and at least one resource view). The user clicks on the menu entry $View -> Show\ Swimlanes$

[1] A view is a part of the planning area in the window. It is realized by a JPanel.

- *Flow of events*: The swimlanes appear on screen and connect actions with their corresponding tasks, which contribute to a resource.
- *Exit condition*: The user clicks on the menu entry $View -> Hide\ Swimlanes$
- *Special requirements*: Additionally, a swimlane will appear, when the user moves the mouse cursor over an action.

2.2.6. Changing the Sorting Mode

- *Participating actors*: AI expert
- *Entry condition*: More than two resource views are visible. The sorting mode[2] is "Update".
- *Flow of events*:
  1. When the sorting mode is "Update", the resources with the latest access made by a task are on top (straight below the action view).
  2. The user clicks on the sorting mode "Grouped" and now all Action-Resources are grouped together on top while all State-Resources are beneath.
- *Exit condition*: None.
- *Special requirements*: As soon as an action is clicked on, all resources accessed will be sorted on top.

2.2.7. Entering Sensor Data

- *Participating actors*: AI expert
- *Entry condition*: A state resource view is expanded.
- *Flow of events*:
  1. The user clicks on the "edit" button of this state resource view.
  2. A popup window opens so the user can either select a state, if it is a Symbolic-State-Resource, or he can enter a numerical number, if it is a Numerical-Heap-State-Resource.
- *Exit condition*: Clicking on the "ok" button to set this value (state or number) as the present sensor data. After clicking on the "ok" or "cancel" button, the popup window disappears.
- *Special requirements*: None.

2.2.8. Adding a Goal

- *Participating actors*: AI expert
- *Entry condition*: A state resource view is expanded.
- *Flow of events*:
  1. The user clicks on the "add goal" button of this state resource view.

[2]The sorting mode defines the sorting of the views in the planning window from top to bottom.

2. A popup window opens so the user can select a state, which should be the goal of the planning process.
3. Additionally, he must enter the time at which the goal has to be reached.

- *Exit condition*: Clicking on the "ok" button to set this value state as the goal for this resource. After clicking on the "ok" or "cancel" button, the popup window disappears.
- *Special requirements*: None.

2.2.9. Moving an Action

- *Participating actors*: AI expert
- *Entry condition*: Visible action in the action view.
- *Flow of events*:
  1. The user drags (clicks on the action and holds the mouse button pressed) the action and moves it in the action view at the timeline.
  2. At the destination time, the user releases the mouse button so the action is moved to this place.
- *Exit condition*: After releasing the mouse button, the action is aligned at the y-axis so that no overlaps of actions occur.
- *Special requirements*: When the action is moved to the left or right border, the view must scroll in the corresponding direction.

2.2.10. Inserting an Action

- *Participating actors*: AI expert
- *Entry condition*: Clicking on the menu item $Add -> Action$
- *Flow of events*:
  1. After a click on the $Add -> Action$ menu item, the mouse cursor changes to a crosshair. The user moves the mouse in the action view to the place desired on the timeline.
  2. He presses the left mouse button so a popup window opens.
  3. In the popup window, the user chooses the desired action type from a list. The time, when the action shall be executed, is already filled in.
- *Exit condition*: The user clicks on the "ok" button to insert this action at the given time. After clicking on the "ok" or "cancel" button, the popup window disappears.
- *Special requirements*: Because of the complexness of this operation, there is a help text instructing the user.

2.2.11. Removing an Action

- *Participating actors*: AI expert
- *Entry condition*: Clicking on the menu item $Remove -> Action$

- *Flow of events*:
  1. After a click on the *Remove − > Action* menu item, the mouse cursor changes to a crosshair. The user clicks on the action, which shall be deleted.
- *Exit condition*: After clicking on the action, the cursors turns to the default cursor icon and the action disappears.
- *Special requirements*: Because of the complexness of this operation, there is a help text instructing the user.

2.2.12. Moving a Precondition/State Task between Resources

- *Participating actors*: AI expert
- *Entry condition*: At least two resource views are visible.
- *Flow of events*:
  1. The user drags a task so the mouse cursor changes to a drag&drop icon.
  2. While moving the mouse over a resource, the mouse cursor changes to a drop-allowed or a drop-denied icon.
  3. The user releases the left mouse button and, if it is possible, the task is inserted to the resource below the mouse cursor. In addition, the drop position gives the information at which time the task should be executed.
- *Exit condition*: Releasing the mouse button.
- *Special requirements*: Moving a task inside a resource on the timeline is possible, too.

2.3. Interface Design Requirements

The application should orientate at the *Ten Usability Heuristics* by Jakob Nielsen[Nie94]. They are:

1. *Visibility of system status: The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.*
2. *Match between system and the real world: The system should speak the users' language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.*
3. *User control and freedom: Users often choose system functions by mistake and will need a clearly marked "'emergency exit"' to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.*
4. *Consistency and standards : Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.*
5. *Error prevention: Even better than good error messages is a careful design which prevents a problem from occurring in the first place.*
6. *Recognition rather than recall: Make objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.*

7. *Flexibility and efficiency of use: Accelerators – unseen by the novice user – may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.*
8. *Aesthetic and minimalist design : Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility.*
9. *Help users recognize, diagnose, and recover from errors: Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.*
10. *Help and documentation: Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.*

The main concept is the representation of the time in an areal visualization. This is done by aligning the actions on the x-axis, which is the time-axis. Additionally, the objects-actions model by Ben Shneiderman [Shn02] is used. The objects are the AI actions and tasks from the agent's plan, and the interface objects are the visible GUI elements, like a box on the screen. The actions of the Shneiderman model are the tasks, like moving an AI action. However, a deflection from the Shneiderman model is that the AI actions consisting of tasks are not displayed in a hierarchical decomposition manner. Because of the continuous changing of the actions, it is not possible to add a lot of information in this abstract display form. Instead, the AI actions are represented very abstract, while the detailed information is accessible below in a separate area. The connection between the abstract and the detailed information is realized by a painted line between them.

Additionally, the concept of direct manipulation [Shn02] is used. This means that the AI actions on the screen can be dragged and dropped with mouse cursor. For this operation, it is not necessary to put numerical data into an input field.

Besides the described techniques (direct manipulation), additional techniques are used:

- A very simple form of hierarchical decomposition without recursion. For example, information of the resources are available in a very abstract manner: Only the name is visible. Only if the user clicks on this name in the list, a more detailed view is expanded in the planning window.
- Furthermore, the concepts and elements of the Java Foundation Classes (JFC) and the Java Look and Feel are used, which are familiar to the user. This supports a fast familiarization with the GUI.
- The input fields all support error avoidance techniques, like the blocking of letters in numerical input fields.
- For power users, shortcuts to functions often used, like play and stop, are available.
- The entering of sensor data or the changing of the sorting mode is realized by icons and buttons.
- The menu tree of the Plan-Presentation looks as follows:
  - File
    - Exit - To exit the application.
  - View

    - Show Swimlanes / Hide Swimlanes - to show or hide the connection lines between objects.
    - Sortmode grouped - to group the action resources and the state resources.
    - Sortmode update - to mix the action and state resources with the rule that the resource with the latest update is sorted on top.
    - Show Transitions / Hide Transitions - to show or hide the transition view.
  - Add
    - Action - to add an action to the plan.
  - Remove
    - Action - to remove an action from the plan.

  With the shortcuts and the intuitive menu tree, typeahead [Shn02] is supported. That means the user can enter commands like Alt + V + A to add an action without waiting on the menu expansion.

Due to the amount of information, which is represented during the planning process, a color coding is required. The color coding concept is orientated towards [Shn02] and does not use colors which are on the opposite side of the color spectrum,like blue and red. Instead, natural colors are used to relax the viewer's eye according to Luke Wroblewski[3]. The colors are selected by a color harmonies calculator[4] choosing the colors depending on the distance to the center of the color spectrum.

As a result of the inserting and removing and the additional scrolling of the actions on the screen, it is not possible to give every action a different color. Instead, every action has the same color as have the connected action and state tasks,because an action is associated with the effect (represented as action or state tasks). Only the precondition tasks have a color different from the actions.

The actions in the Excalibur model are a group of tasks, therefore the actions are painted without any border to represent the group property. Instead, the atomic tasks are painted with a border to represent the closed property.

The evaluation of the concepts is done by expert reviews and by formal usability inspections. The expert reviews are done, with a cognitive walkthrough, by Dr. Nareyek while testing the user interface of the Plan-Presentation. Additionally, a formal usability inspection is realized by the author during a netmeeting presentation with other team members. The entire functionality is explained and commentaries are noted down.

2.4. Identifying Nonfunctional Requirements

The basic nonfunctional requirement for graphical user interfaces is that they should not be blocked while the data model is being computed. In addition, it should be oriented towards the MS Windows style.

A special feature of this thesis is the necessity for the Plan-Presentation to be extremely extensible, because additional AI techniques will be added to the data model soon. For example, at present a resource can be only one variable, like the state of a door. But it is not possible to

[3] http://www.boxesandarrows.com/archives/natural_selections_colors _found_in_nature_and_interface_design.php
[4] http://www.easyrgb.com/harmonies.php

combine resources, like a car with one variable for speed, one for location and one for the fuel level.

Furthermore, the data model should run further on as a standalone application without the graphical user interface. This means there should be as few dependencies as possible. The CPU and the memory usage shall also not be worsened when in stand alone mode.

In addition, the new graphical user interface Plan-Presentation shall not get in conflict with the existing graphical user interface of the DragonBreath engine.

3. Object Analysis

In this chapter the many objects are categorized to enable some sort of a divide and conquer approach. After that, in Section *3.2. Mapping Use Cases to Objects*, it is checked, if every name appearing in the use case list is mapped to an object so that no objects are missing. In Section *3.3. Modeling Generalization*, common parts are joined together to one object.

3.1. Identifying Entity-, Boundary-, Control-Objects

The division of the objects in this section is supporting the architecture commitment in the next chapter. The objects are divided in entity-, boundary-, and control-objects.

- *entity-objects* are objects inside of the data model, holding the data.
- *boundary-objects* are the objects visible to the user (or to other applications, but this is not the case in the Plan-Presentation).
- *control-objects* are the objects describing processes and handle the user input.

3.1.1. The Entity-Objects

From the theoretical description of the planning system in [Nar01] and the existing data model we derive the following entity-objects:

- *Action-Object-Constraint* representing an action
- *Action-Task-Object-Constraint* representing an action task
- *State-Task-Object-Constraint* representing a state task
- *Precondition-Task-Object-Constraint* representing a precondition task
- *State-Resource-Constraint* representing the general form of a State-Resource-Constraint and the more specialized:
  - *Numerical-Heap-SRC* representing a Numerical-Heap-State-Resource, which contains numerical values like a fuel level,
  - *Symbolic-SRC* representing a Symbolic-State-Resource of which the values consist of predefined states. It can only be in state A, B or C etc., but not between A and B.
- *Transition-Constraint* representing the external effects known as the transitions
- *Current-Time-Object-Constraint* representing the present current time of the data model
- *Task-Constraint* controlling the correct structure of an action and some additional things. This object is not a control-object, because from the Plan-Presentation point of view it is a simple data model object
- *Global-Search-Control* representing the engine interface but being, like the Task-Constraint, no control-object

- *Real-Time-Control* representing the speed control of the engine

The Plan-Presentation is a graphical user interface to the EXCALIBUR data model, so there are no own entity-objects for the Plan-Presentation application. Of course, there will occur some entity-objects in the object design chapter, when they are needed for the low-level implementation.

3.1.2. The Boundary-Objects

Most of the new objects are boundary-objects, because the main focus of the Plan-Presentation is the interface to the user.

First the entity-objects need a representation in the graphical user interface:

- *Action-Semantic-Graphic* representing one Action-Object-Constraint
- *Action-Task-Semantic-Graphic* representing one Action-Task-Object-Constraint
- *State-Task-Semantic-Graphic* representing one State-Task-Object-Constraint
- *Precondition-Task-Semantic-Graphic* representing one Precondition-Task-Object-Constraint
- *Numerical-SRC-View* representing a whole Numerical-Heap-SRC
- *Symbolic-State-Resource-View* representing a whole Symbolic-SRC
- *Transition-Semantic-Graphic* representing also an Action-Object-Constraint, but one which was created by the Transition-Constraint
- *Swimlane* representing one line which connects two or more objects in the graphical user interface
- *Help-Text* representing a string which is displayed on top of the planning area in the window

Secondly, there are additional objects required, which support the representation of the entity-objects:

- Planning Window named as *PlanJFrame* because of its implementation
- *Action-Coordinator-View* containing all Action-Semantic-Graphics
- *Transition-Coordinator-View* containing all Transition-Semantic-Graphics

Not visible supporting objects are classified as control-objects in the next section.

At last, there are boundary-objects admitting the data entry, for example forms:

- *Add-Action-Dialog* to add an action to the planning system
- *Edit-Sensor-Dialog* to change sensor data
- *Breakpoint-Dialog* to de/activate the breakpoints in the data model
- *Add-Goal-Dialog* to add a goal to a Symbolic-State-Resource-View.

3.1.3. The Control-Objects

To complete the supporting objects of the boundary section, there are the following control-objects:

- *Action-Resource-Coordinator* coordinating all action resources. It is represented as a list of all action resources.
- *State-Resource-Coordinator* coordinating all state resources.

Additional control-objects are:

- *Zoomslider* to control the zoom level
- *Update-Sortmode* and *Grouped-Sortmode* buttons
- *Play*, *Pause*, *Fastforward* and *Stepping* buttons to control the speed of the planning process
- *Add-Goal* and *Edit-Sensor-Data* buttons to open the corresponding dialog windows
- *Retract Button* to retract and to expand a view

After committing to an architecture[1] in Chapter *4.2 Designing an Initial Subsystem Decomposition* the following control-objects appear, which connect the model and the presentation:

- *Action-Resource-Controller* connecting an Action-Resource-View with an Action-Resource-Object-Constraint
- *Numerical-SRC-Control* connecting a Numerical-SRC-View with a Numerical-Heap-SRC
- *Symbolic-State-Resource-Control* connecting a Symbolic-State-Resource-View with a Symbolic-SRC

3.2. Mapping Use Cases to Objects

The following sections check, whether all objects are identified by comparing the object names of the use cases with those of Section *3.1 Identifying Entity-, Boundary-, Control-Objects*. The software engineering process does not allow that important objects, which are not defined in the *3.1 Identifying Entity-, Boundary-, Control-Objects* section, appear in this section.

3.2.1. Viewing the Plan (Use Case 2.2.1)

There are the following nouns:

- *Planning-Window* - corresponds to the PlanJFrame boundary object
- *Play-Button* - the Play-Button
- *Actions* - appear on the screen as Action-Semantic-Graphic objects
- *Current-Time* - represented in the data model by the Current-Time-Object-Constraint and in the graphical user interface by the Current-Time-Semantic-Graphic
- *Zoom Slider* - corresponds to the zoom slider control object

3.2.2. Viewing a Resource (Use Case 2.2.2)

- *Resources in the overview lists* - the overview lists are the Action-Resource-Coordinator and the State-Resource-Coordinator. The items inside a list are the Symbolic-State-Resource-Control, the Numerical-SRC-Control and the Action-Resource-Controller.

[1] The software engineering process is iterative, so the author of this thesis has to decide which information from the last iteration of the process has to be brought forward.

- *Action Resource* - the entity object Action-Resource-Object-Constraint represented by the Action-Resource-View. Both are connected by the Action-Resource-Controller.
- *View appears* - a Symbolic-State-Resource-View, a Numerical-SRC-View or an Action-Resource-View appears in the planning window.
- *Symbolic-State-Resource* - of course the State-Resource-Object-Constraint and its more specialized subclass Symbolic-SRC
- *Numerical-Heap-State-Resource* - the State-Resource-Object-Constraint and its more specialized subclass Numerical-Heap-SRC

3.2.3. Retracting a View (Use Case 2.2.3)

- *Retract-Button* - relates to the Retract-Button control object

3.2.4. Showing Transitions (Use Case 2.2.4)

- *Transition-View* - defined as the Transition-Coordinator-View
- *Action-View* - defined as the Action-Coordinator-View

3.2.5. Activating Swimlanes (Use Case 2.2.5)

- *Action-View* - defined as the Action-Coordinator-View
- One *Resource-View* - either a Symbolic-State-Resource-View, a Numerical-SRC-View or an Action-Resource-View
- *Swimlane* - represented by the Swimlane boundary-object
- *Action* - in this context it means Action-Semantic-Graphic
- *Tasks* - either a Precondition-Task-Semantic-Graphic, a State-Task-Semantic-Graphic or an Action-Task-Semantic-Graphic

3.2.6. Changing the Sorting Mode (Use Case 2.2.6)

- Two visible *Resources* - At least two visible resources are either of the type Action-Resource-View, Symbolic-State-Resource-View or Numerical-SRC-View.
- Latest accessed by a *task* - The task can be any task of the types Precondition-Task-Object-Constraint, State-Task-Object-Constraint or Action-Task-Object-Constraint
- *Sortmode grouped* - The sortmode is an attribute of the Real-Time-Control.
- *Action-Resources* - of course the Action-Resource-View
- *State-Resources* - either a Symbolic-State-Resource-View or a Numerical-SRC-View

3.2.7. Entering Sensor Data (Use Case 2.2.7)

- *Edit-Button* - the Edit-Sensor-Data button as defined in the control-objects
- *State-Resource-View* - either a Symbolic-State-Resource-View or a Numerical-SRC-View
- *Popup-window* opens - This corresponds to the Edit-Sensor-Dialog or Edit-Numerical-Sensor-Dialog.
- To *select a state* - means a SymSRCState, which is part of the data model and not declared in the analysis model.
- *Ok-button* and *cancel-button* - simple gui (graphical user interface) buttons with minimal functionality

3.2.8. Adding a Goal (Use Case 2.2.8)

- *State-Resource-View* - a Symbolic-State-Resource-View
- *Add-Goal-button* - the Add-Goal button
- *Popup-window* opens - means the Add-Goal-Dialog
- *Select a state* - The state means a Sym-SRC-State which is part of the data model and not declared in the analysis model.
- *Ok-button* and *cancel-button* - simple gui buttons with minimal functionality

3.2.9. Moving an Action (Use Case 2.2.9)

- *Action-View* - defined as the Action-Coordinator-View
- The user drags an *Action* - means an Action-Semantic-Graphic.

3.2.10. Inserting an Action (Use Case 2.2.10)

- *Action-View* - defined as the Action-Coordinator-View
- *Popup-window* opens - means the Add-Action-Dialog
- List of *action-types* - means a TCActionType which is part of the data model and not declared in the analysis model.
- *Ok-button* and *cancel-button* - simple gui buttons with minimal functionality
- *Help text* - defined as the Help-Text boundary-object

3.2.11. Removing an Action (Use Case 2.2.11)

- *Action - means the Action-Semantic-Graphic visible on screen.*
- *Help text* - defined as the Help-Text boundary-object

3.2.12. Moving a Precondition/State Task between Resources (Use Case 2.2.12)

- Two *Resources* - There are at least two resources of the same type visible, either of the type Symbolic-State-Resource-View or Numerical-SRC-View.
- The user drags a *task* - means either a Sym-SRC-Task-Semantic-Graphic or a Numerical-SRC-Task-Semantic-Graphic.

3.3. Modeling Generalization

All views have the same retract functionality. Therefore it is reasonable that all views must extend the class *RetractableView*.

Apart from that, the views *Action-Coordinator-View*, *Transition-Coordinator-View* and *Action-Resource-View* are very similar. They all show some sort of tasks which can be executed in parallel and all have timespans. The generalized class is named *Multi-Level-View*, the three other classes are no longer necessary.

4. System Components

In this Chapter *4*. System Components design goals are derived by the nonfunctional requirements. After this, architectural patterns are discussed and a system architecture is built.

4.1. Identifying Design Goals from Nonfunctional Requirements

From the nonfunctional requirement "GUI must not be blocked while the engine thread is computing" an extra GUI-thread is derived.

Additionally, the Plan-Presentation application shall be very extensible. This is realized by a plugin system. This means that new artificial intelligence methods like objects with more than one variable can be included in the GUI with an extra view which is loaded by the application as plugin.

From the requirement that the data model shall work furtheron as standalone application, too, and that the Plan-Presentation application should not get in conflict with the existing DragonBreath GUI, it is deduced that the graphical user interface is docked to the data model by the observer pattern.

At last, the graphical user interface shall be oriented at standard windows applications. Unfortunately, Java supports the MS Windows Look and Feel only for the MS Windows platform, but it is to assume that the users of other platforms work much that intuitively with their native platform Look and Feel as windows users do. Therefore, the Plan-Presentation shall always use the native platform Look and Feel, which is default at this platform.

4.2. Designing an Initial Subsystem Decomposition

The challenge at the system design is to find the best suitable architectural pattern and adapt it to the special needs of the application. Patterns shall only give an orientation for the system architect. So first the patterns in the shortlist are explained.

4.2.1. Architectural Patterns Discussion

The following three patterns were put in the short-list, because they all provide an abstraction mechanism and are laid out for graphical user interfaces.

Layer-Pattern

The layer-pattern is a good example, how architectural patterns work. The 7 layer OSI model helps the developers to divide the work between different teams.

Between two layers, there are defined interfaces where developers can put on their work. The interfaces change from layer to layer, but the concept of defined function calls is always the same. This repeating helps the developer to understand the global architecture of the application.

However, a specialized version of the layer-pattern is usable for graphical user interfaces, the *Four-Layer-Architecture*.

The application is separated in four layers, which are described at http://c2.com/cgi/wiki?FourLayerArchitecture as follows:

- *The View layer. This is the layer where the physical window and widget objects live. It may also contain Controller classes as in classical MVC. Any new user interface widgets developed for this application are put in this layer. In most cases today this layer is completely generated by a window-builder tool.*
- *The Application Model layer. This layer mediates between the various user interface components on a GUI screen and translates the messages that they understand into messages understood by the objects in the domain model. It is responsible for the flow of the application and controls navigation from window to window. This layer is often partially generated by a window-builder and partially coded by the developer.*
- *The Domain Model layer. This is the layer where most objects found in an OO analysis and design will reside. Examples of the types of objects found in this layer may be Orders, Employees, Sensors, or whatever is appropriate to the problem domain.*
- *The Infrastructure layer. This is where the objects that represent connections to entities outside the application (specifically those outside the object world) reside. Examples of objects in this layer would include SQLTables,[...]*

In our application domain, the *infrastructure layer* would be the DragonBreath constraint solver, while the *domain model layer* would be the Excalibur data structure.

The *application model layer* would be a set of classes which preprocesses the Excalibur data to show them in the *view layer*, for example, to compute all action task durations to show the duration of an action.

The advantage is that every developer understands this simple architecture at once.

The disadvantage is that the *application layer* grows very complex. This layer is a monolith and the extensible-goal cannot be satisfied. All preprocessing functionality must be combined in this layer. However, there are different levels of preprocessing: While the Action-Task is represented in the view-layer, this object is also needed to compute the duration of an action with multiple Action-Tasks. Additionally, for every object in the view there must be a corresponding object, which informs the view-object about changes in the data model (the domain model layer), in the application layer.

MVC

The pattern most commonly used for graphical user interfaces is the Model-View-Controller (MVC) pattern. It consists of:

- *Model* - The model is a data model containing the data to represent. This is often part of the application domain. In our Plan-Presentation program, it is both the DragonBreath and the Excalibur engine.
- *View* - The view represents the data on the screen. This can be one window or many parts (GUI containers) in a window.

- *Controller* - The main function of the controller is to take the user input (often caught by the view) and make a model update. In addition, the controller creates the views.

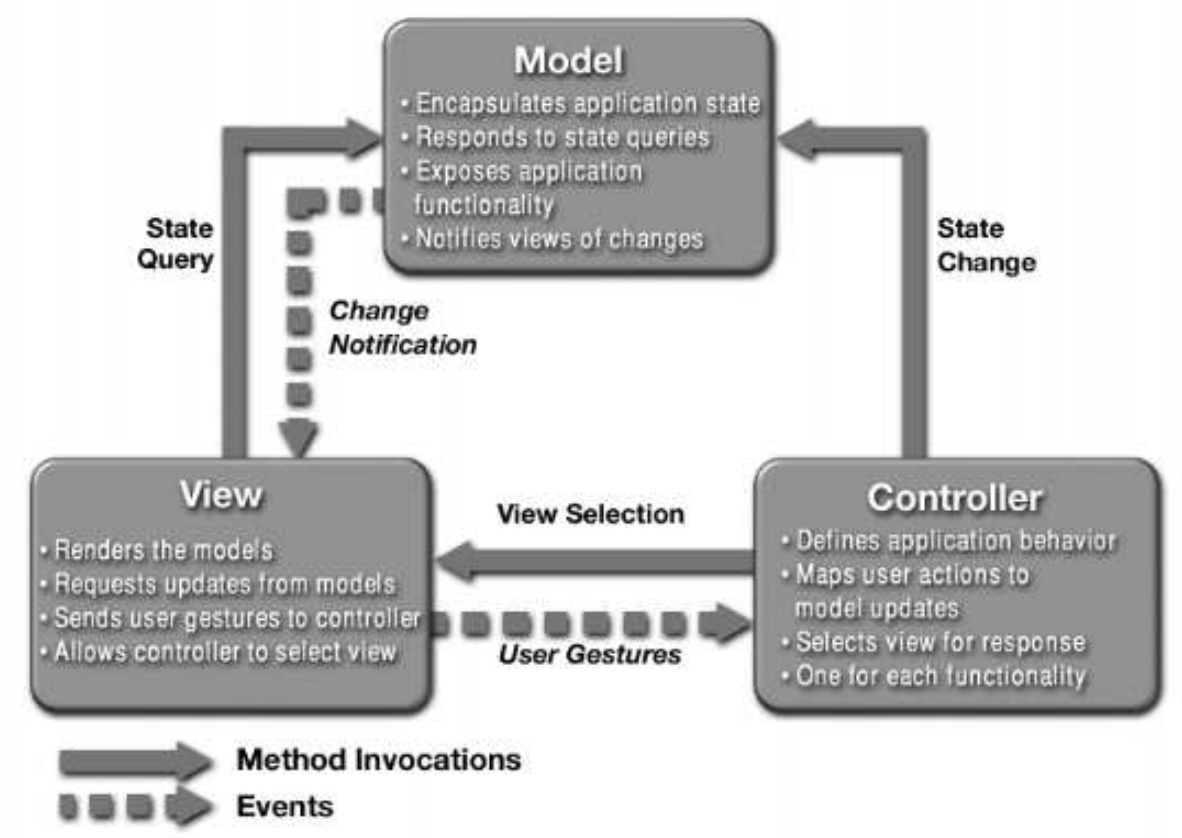


Figure 4.1.: The Model-View-Controller explained by Vijay Ramachandran at developer.Java.sun.com

This breakdown is very useful, because the representation is separated from the data model. The whole business logic is encapsulated in the model, and therefore the view can be exchanged if another design is required. Furthermore, this architecture allows different views for one data model.

This pattern can therefore be used for the Plan-Presentation application, but it also has a disadvantage:

The application shall show a list of all state resources available, but this is already a view. When clicking on a list item, it will open a more detailed view in the main window. This scenario can be described as a view (the detailed part) ”inside” of another view (the resource list). It would be very unstructured, if every controller of a list item and the controller of the list are connected to the data model. There is something like a tree structure needed.

PAC

The Presentation-Abstraction-Control architectural pattern (PAC) defines a structure for interactive software systems in the form of a hierarchy of cooperating agents. Every agent[1] is responsible for a specific aspect of the application's functionality and consists of three components:

[1] In this context ”agent” has another meaning than in the artificial intelligence environment.

presentation, abstraction, and control. This subdivision separates the human-computer interaction of the agent from its functional core and its communication with other agents. [BMR+96]

This PAC pattern allows a hierarchical structure, which is missing in the MVC pattern. The hierarchy consists of

- A *top-level-agent* is connected to the data source and creates the agents on the layer below.
- The *intermediate-level-agents* coordinate the agents below them. They preprocess data for the bottom-level-agents.
- The *bottom-level-agents* are mainly visible on screen. They represent a small part of the data model in different ways.

See Figure 4.2 for a visualization of the PAC tree structure.

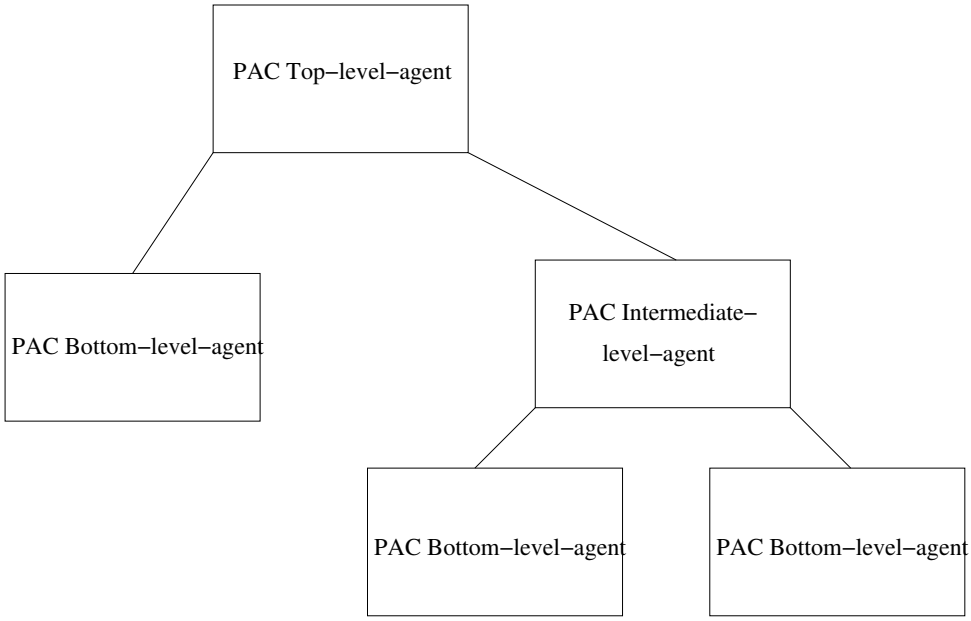


Figure 4.2.: Overview on the Presentation-Abstraction-Control architectural pattern.

In the Plan-Presentation application the top-level-agent is the application framework, while the intermediate-level-agent is a *State-Resource-Coordinator*. This intermediate-level-agent coordinates the single state resources. More concretely the intermediate-level-agent is represented as a list of state resources. Every list entry is a bottom-level-agent. When clicking on this entry, the bottom-level-agent expands its view in the main window to allow a more detailed view.

Every agent in the pac pattern consists of four classes:

- The *Presentation* represents the data on screen and is similar to the view in the MVC pattern.
- The *Abstraction* is comparable to the data model.
- The *Controller* gets the user input like in the MVC pattern and also connects the Abstraction with the Presentation. The advantage is that, because of the strict separation, the single parts of an agent can easily be exchanged.
- At last, there is a class *PACAgent* working as an interface to other agents. This is mainly a facade[2] of the agent.

[2]The Facade-Pattern provides a unified interface to a set of interfaces in a subsystem.

In Figure 4.3, a class diagram of such a PAC-Agent is provided.

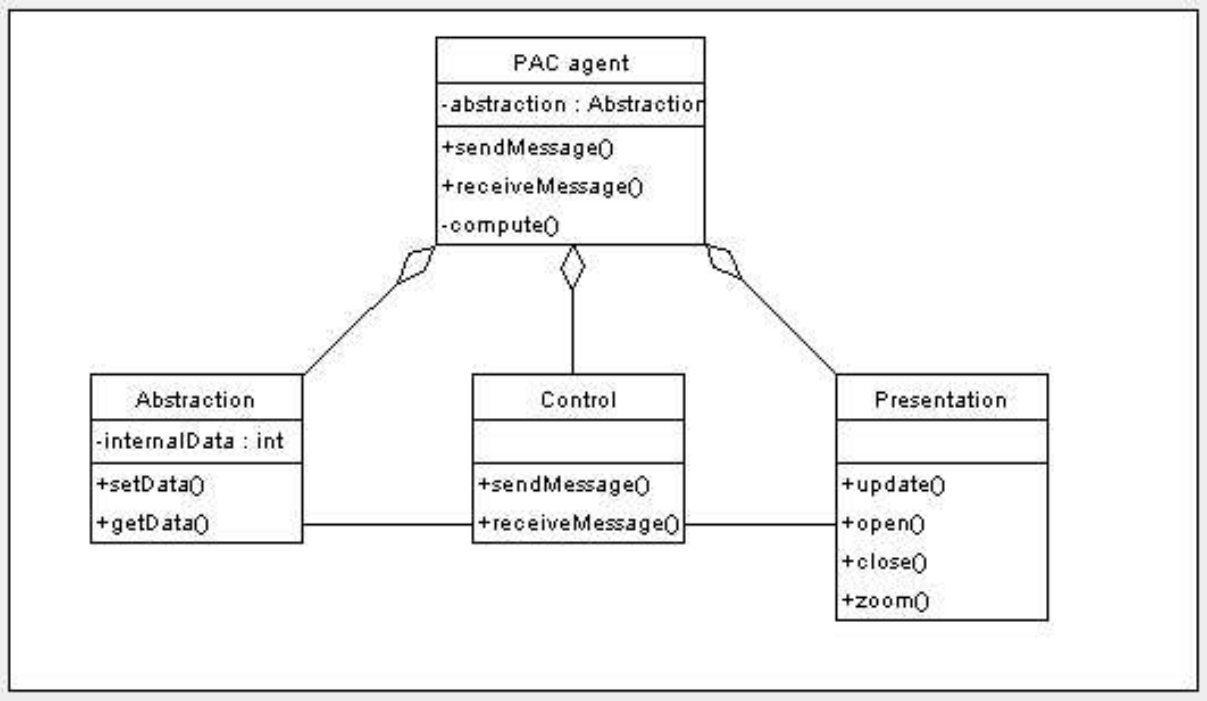


Figure 4.3.: Class diagram of a PAC-Agent.

4.2.2. Architecture Decision

The Presentation-Abstraction-Control pattern fits best for the Plan-Presentation because of its hierarchical decomposition structure. In this situation, this pattern is an improvement of the MVC pattern, because the bottom-level-agents are similar to a MVC pattern.

The Four-Layer-Architecture is not usable for the Plan-Presentation application, because it is missing flexibility. New features cannot be added without changing at least three layers.

4.2.3. Building a System Architecture

In the following sections every subsystem is described abstractly.

Plan-Presentation

As seen in the section before, the Presentation-Abstraction-Control pattern needs a top-level-agent named *Plan-Presentation* which is connected to the data source (see the rectangle on top of figure 4.4). The data source is the Global-Search-Control of the DragonBreath engine.

The top-level-agent also opens the planning window and loads all plugins into the application.

Action-Coordinator

The first plugin is the Action-Coordinator-Plugin. It represents an overview of all actions, but without any details. An example for a detail would be: Which task is connected to an action.

The Action-Coordinator is an intermediate-level-agent, because the objects on the screen are the Action-Semantic-Graphic classes. These are the bottom-level-agents.

Transition-Coordinator

The Transition-Coordinator-Plugin represents the external effects caused by the environment. The internal data structure is the same as a normal action so that many attributes and functions can be inherited from the Action-Coordinator-Plugin, and no bottom-level-agent is required, because it will use the bottom-level-agent of the Action-Coordinator-Plugin.

Action-Resource-Constraint

The Action-Resource-Coordinator is an intermediate-level-agent and is represented as a list of all available Action-Resources. It coordinates the views of the action resources.

The Action-Resource-Controller is the entry in the Action-Resource list and is a bottom-level-agent representing the Action-Tasks on an Action-Resource.

In Figure 4.4, the Action-Resource-Coordinator is connected with the Action-Resource-Controller, but the line is behind another box and therefore not visible.

State-Resource-Constraint

The State-Resource-Coordinator-Plugin is the most complex subsystem, as the State-Resource-Constraint has two subclasses: The Symbolic-State-Resource-Constraint, where the state is chosen from a predefined set of states, or the Numerical-Heap-State-Resource constraint, whereas the state is a numerical value between minus infinity and infinity.

The State-Resource-Coordinator manages both in a list displayed on the left side of the planning window. Both the Symbolic-State-Resource-Controller and the Numerical-HSRC-Control are entries in this list.

The Symbolic-State-Resource-Controller is a bottom-level-agent and connected to the State-Resource-Coordinator. It represents a State-Resource-Constraint with a Symbolic-SRC as subclass. Furthermore, it represents all states, precondition tasks, and state tasks which are contained.

A bottom-level-agent which is also connected to the State-Resource-Coordinator is the Numerical-HSRC-Control. This subsystem represents a Numerical-Heap-SRC.

Plan-Plugin

The Plan-Plugin is a subsystem to manage most parts of the graphical user interface functions, like the speed control and the file menu. As there are no agents to coordinate below, the Plan-Plugin is a bottom-level-agent.

Moreover, it represents the current time which corresponds to the Current-Time-Object-Constraint in the data model. The Plan-Plugin informs the Plan-Presentation top-level-agent about time changes, but the Plan-Plugin itself has no representation of its own for the current time implemented. This functionality is provided by the top-level-agent.

Swimlane-Control

A particular type of a subsystem is the Swimlane-Control. This subsystem provides a service to the plugins and draws a line between two GUI objects.

The Swimlane-Control is defined as a bottom-level-agent, even though it has no abstraction or control class. Furthermore, it could be separated into an intermediate-level-agent and a bottom-level-agent, because the Swimlane-Control has many Swimlanes connected to it. However, an architecture pattern is only an orientation for the software architect, and the nomenclature of top-, intermediate- and bottom-level-agents should be consistent through-out the whole architecture.

4.3. Identify Boundary Conditions

Boundary conditions are situations, when the system state is not in the normal running mode. This is the case during startup or shutdown and in cases of exceptions. The identifying of boundary conditions prevents the overlooking of rare situations by the developer.

4.3.1. Startup

The first boundary condition is the startup of the application. In contrast to normal application, the Plan-Presentation startup takes place while another application, the DragonBreath engine, is running. The planning window can be opened any time during the planning process. Therefore, the Plan-Presentation has to read the existing CSP-graph from the DragonBreath data model.

In addition, when the Plan-Presentation is instantiated, it has first to load all plugins, which add menu items, control elements and view to the GUI.

4.3.2. Shutdown

The shutdown is very simple, as there are no data to save. So, when clicking on the menu entry $File- > Exit$ the application calls the Java command System.exit().

4.3.3. Exceptions

There are two types of exceptions for this application. The first is an error in the user input, for example when editing sensor data. These exceptions can be caught or avoided by the input-dialog, which checks the input value or only allows to put in the correct data by a pulldown input field.

The second type of exceptions is more complex, because it happens in the DragonBreath engine. These exceptions cannot be caught or handled, because the engine is running in a different thread and the Plan-Presentation is in another ”branch”. This means that exceptions are passed on from the lowest level, where the exceptions happen, to the next higher level. On this path there are only DragonBreath classes, and the Plan-Presentation application has no possibility to catch them. Therefore these exceptions will be printed to the console window. This solves the conflict of the design goals ”DragonBreath should run independent of the Plan-Presentation” and ”Plan-Presentation should catch all exceptions” in favor of the first design goal ”independence”.

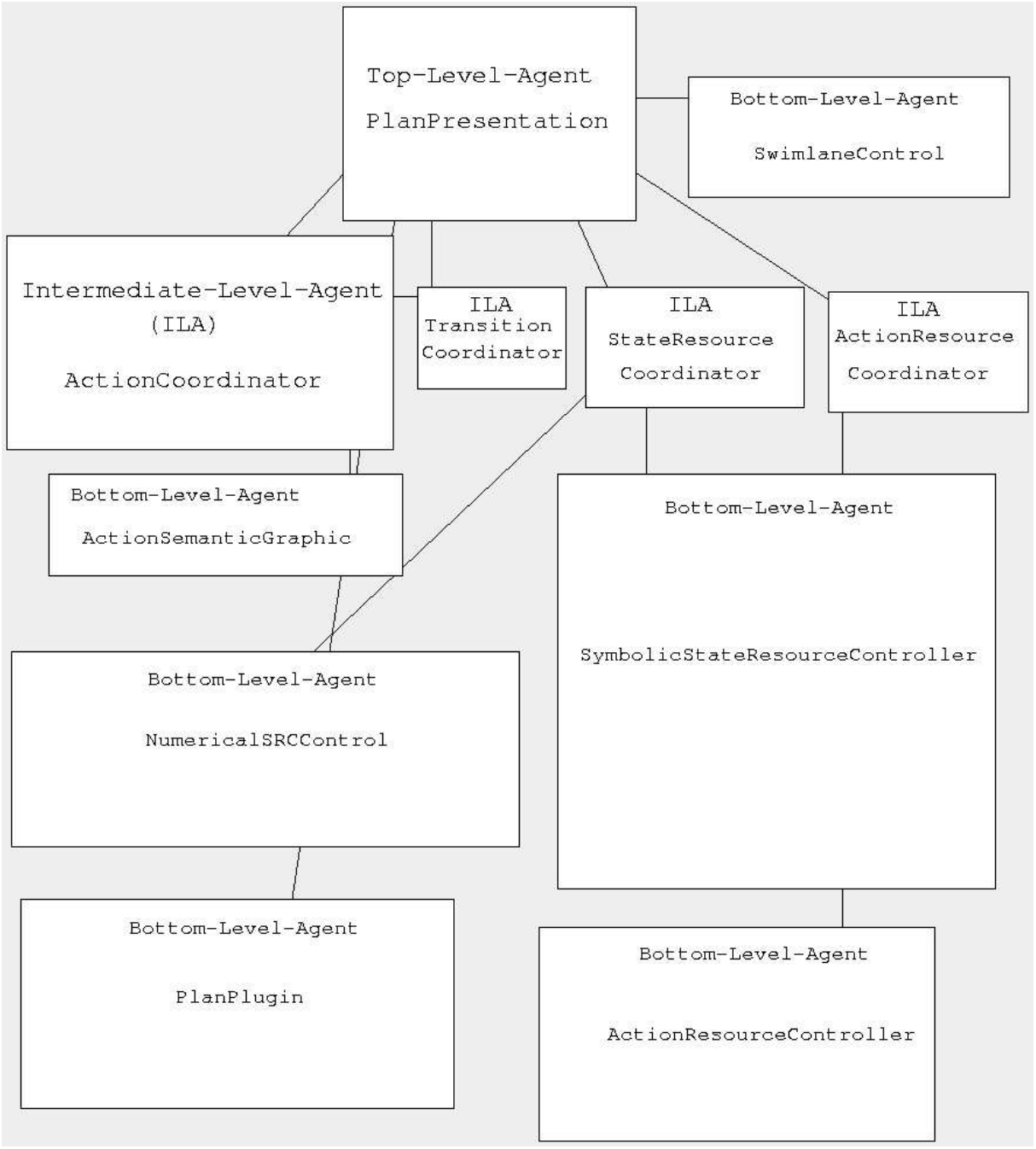


Figure 4.4.: An overview of all subsystems of the PlanPresentation application.

5. Object Design and Implementation

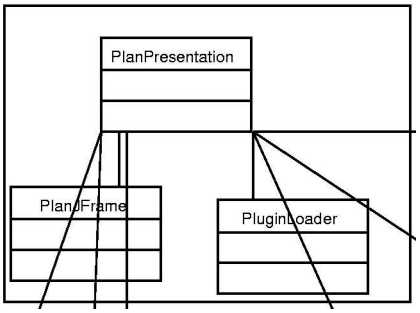
In this chapter, it is explained how the application is implemented. The focus is the concept of the application and the plugins, not the description on function- or attribute-level. A more detailed description is available as javadoc websites at www.ai-center.com.

First of all, a detailed overview about the whole application is given in Figure 5.1. Every subsystem is explained in an extra section of this chapter, so this figure is only to understand the correlation between the subsystems.

5.1. Plan-Presentation Base System

The Plan-Presentation base system, which is the top-level-agent, consists of three classes:

- *Plan-Presentation*
- *Plan-JFrame*
- *Plugin-Loader*



The Plan-Presentation is instantiated in the main program with

```
PlanPresentationAPI p = new PlanPresentation((GlobalSearchControl) ↩
    ↪ gsc);
```

First of all, it opens a planning window: the Plan-JFrame. The constructor of this class initializes most parts of the GUI-elements. After this, the Plan-Presentation constructor creates a Plugin-Loader class[1]. This Plugin-Loader instantiates every plugin and calls the *initPlugin()* function of the plugin.

Every plugin can request control elements, menu entries or views to be added by the Plan-Presentation class. They can also register for one or more constraint types, about which they will get informed if one is created or deleted by the Global-Search-Control. Additionally, the plugins can register for support information, like the changing in the Main-View-Model. This Main-View-Model contains the data what timespan is shown on screen, and therefore this model is necessary to allow scrolling for the views.

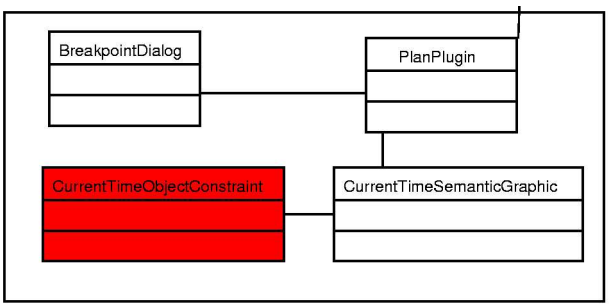
[1] There are different Plugin-Loaders available: A loader which walks through the directories and a loader which loads the plugins defined in a config file.

When all plugins are loaded completely, the Plan-Presentation reads the CSP-graph. When it finds an object in this graph with a constraint type, which is registered by a plugin (registered means that the plugin has registered for getting information about the creation or deletion of such a constraint type), the Plan-Presentation informs the plugin about this.

After the completion of this startup process, the Plan-Presentation is mainly a mediator for the plugins and the Global-Search-Control. The plugins are not connected directly, so they send messages through the Plan-Presentation to other subsystems. Furthermore, the Plan-Presentation holds all references to subsystems where a direct connection is necessary, like the Swimlane-Control.

5.2. Plan-Plugin

The Plan-Pugin has mainly a support task for the application, but it does also represent a constraint type: the Current-Time-Object-Constraint.



At first, it creates the buttons:

- *Sortmode update* sorts the views in the planning window in the order of the last access.
- *Sortmode grouped* groups the views into views belonging to action resources and views belonging to state resources.
- *Play*, *Pause*, *Fastforward*, *Stepping* control the planning speed of the application. The play-mode waits for a given time between two breakpoints. This waiting time can be adjusted by a slider called *Pause-Slider*.

Additionally, it creates the following menu entries:

- $File- > Exit$ to exit the whole application.
- $View- > Show\ Breakpoints$ which open an extra window, the *BreakpointDialog*.

This *Breakpoint-Dialog* allows the user to activate or deactivate breakpoints by a single mouse click. If the user presses the ”ok-button”, the state of every breakpoint is transmitted to the Real-Time-Control of the DragonBreath engine.

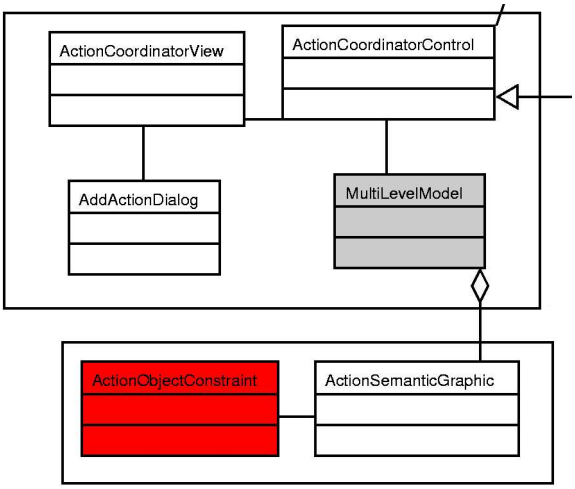
As promised, the Plan-Plugin also represents a constraint of the data model. Therefore the *Current-Time-Semantic-Graphic* is the representation of the *Current-Time-Object-Constraint*[2]. So the Plan-Plugin registers the constraint type of the Current-Time-Object-Constraint during the *initPlugin()*-call. When this Object-Constraint is created, the reference to this object is sent to the Plan-Plugin *addConstraint()* function. The *addConstraint()* function instantiates the

[2]The Current-Time-Object-Constraint is of red color in the UML model, because this is an existing class in the DragonBreath data model.

Current-Time-Semantic-Graphic, which is now directly connected to the Current-Time-Object-Constraint[3]. If something changes at the Current-Time-Object-Constraint or its connected variable, the Current-Time-Semantic-Graphic gets informed about this. Then the Current-Time-Semantic-Graphic changes the current time of the graphical user interface by calling the *setCurrentTime()* function of the Plan-Presentation class.

5.3. Action-Plugin

The Action-Plugin is the most complex plugin, because for a single action displayed on screen, a lot of precondition, state and action tasks have to be analyzed to compute the beginning and the ending time of the action.



The class Action-Coordinator-Control-Plugin is loaded by the Plan-Presentation, and it registers for the Action-Object-Constraint. Then it creates the Action-Coordinator-View, which is the view on top of the planning window. Additionally, it instantiates the Multi-Level-Model[4], which is a data model to arrange objects with a fixed duration on different high-levels so that no overlap occurs.

When a new Action-Object-Constraint is created in the Global-Search-Control, the Action-Coordinator-Control-Plugin gets informed about this. It then creates a new Action-Semantic-Graphic, which can be seen as bottom-level-agent. After that, it adds the Action-Semantic-Graphic to the Multi-Level-Model, which computes the high-level of this action.

This Action-Semantic-Graphic connects to the Action-Object-Constraint and to the Action-Object-Constraint.connections set. Then it reads all connected tasks and selects the lowest beginning time of all tasks and sets it as the beginning time of the action. The same happens to the biggest ending time of all tasks, which results in the ending time of the action. When the boundary times are computed, the Action-Task-Semantic-Graphic checks every precondition task whether it is fulfilled. If yes, the traffic light color of the action is green, otherwise it is red.

If the user wants to move an action to another beginning time, he can drag it, move it to the desired time and drop it. The Action-Semantic-Graphic translates the screen pixel to a time value. Then it computes the difference from the old beginning time to the new beginning

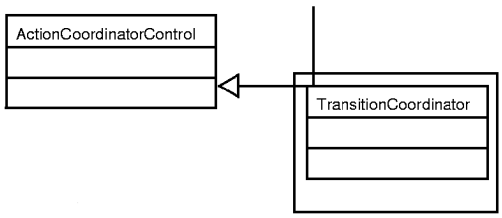
[3]The Current-Time-Semantic-Graphic is actually connected to the variable containing the current time, and this variable is connected to the Current-Time-Object-Constraint.

[4]The Multi-Level-Model is grayed, because it is not in the same package as the other classes. The reason is that this class is often used in different plugins and modeling the inheritance in the UML diagram would make it look very complex.

time, which was set by drag&drop. After this, it adds the (possibly negative) difference to the beginning time of every connected task.

For debugging purposes, it is possible to give the planning engine a hint by inserting an action. For this, the user must click on the menu entry *Add− > Action* so that the corresponding GUI action activates the mouse tracking mechanism in the Action-Coordinator-View. After that, the mouse cursor changes to a crosshair, and if the user clicks inside of the Action-Coordinator-View, the Action-Coordinator-View opens the Add-Action-Dialog. The beginning time is already filled in and the user has to choose one amongst all possible action types from a pulldown. After clicking the "ok button" the Add-Action-Dialog inserts the action by first inserting the Action-Object-Constraint and then all tasks connected. After that the Add-Action-Dialog closes.

5.4. Transition-Plugin



The transitions are very similar to normal actions. They also consist of an Action-Object-Constraint and a lot of connected tasks. The difference is that there are no action tasks connected to them (only precondition and state tasks are connected) and that the action type of the Action-Object-Constraint is registered at the Transition-Constraint.

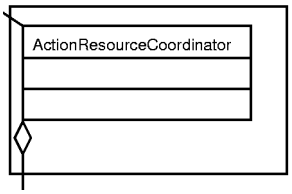
To remind: The transitions are external effects. At a normal action, the agent decides to execute it, whereas a transition is always executed if its preconditions are fulfilled.

As a result of the similarity with the actions, it is not necessary to develop a new plugin. The Transition-Coordinator-Plugin only extends the Action-Coordinator-Plugin and overwrites the *addConstraint()* function: While the Action-Coordinator-Plugin adds only Action-Object-Constraints if the action type is NOT registered at the Transition-Constraint, the Transition-Coordinator-Plugin adds only Action-Object-Constraints if the action type IS registered at the Transition-Constraint.

The second difference is that the Action-Coordinator-View, as the most important view in this application, is always visible, while the Transition-Coordinator-View[5] has a menu entry *View− > Show Transitions* to expand this view. In exchange, it has no Add-Transition feature, because transitions are always executed if possible. If not possible, they also cannot be inserted in the planning model.

[5]In fact, the Transition-Coordinator-View does not exist, because it is identical to the Action-Coordinator-View. This nomenclature is only for better understanding.

5.5. Action-Resource-Plugin

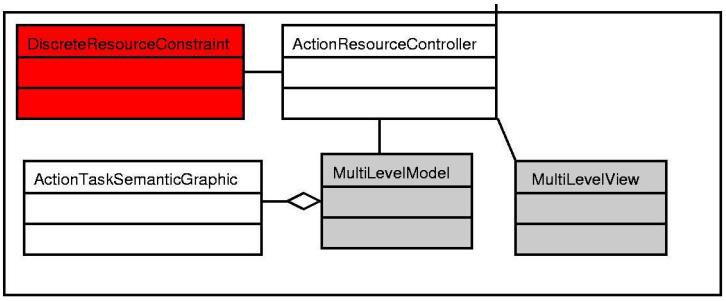


The Action-Resource-Plugin shows the allocation of the action resources. For that purpose, it lists all available action resources in a JList on the left side of the Plan-Presentation window, and if the user clicks on one list entry, it expands or retracts the corresponding detail view of the action resource.

At first, the Action-Resource-Coordinator-Plugin registers for the DISCRETE-RESOURCE-CONSTRAINT[6] type and gets informed, when such a constraint is created. For every constraint created, the Action-Resource-Coordinator-Plugin instantiates an Action-Resource-Controller which is represented as a list entry. To accomplish the properties for a list entry, the Action-Resource-Controller provides a comparison function to other Action-Resource-Controllers, and a change-subscription mechanism to inform the JList about changes in the latest access time.

If such a list entry is clicked on by the user, the *onClick()* function in the Action-Resource-Controller is called. If the view of this controller was retracted, the view is now expanded. For this purpose, the Action-Resource-Controller creates a Multi-Level-Model[7] and a Multi-Level-View. This Multi-Level-View is in fact the parent class of the Action-Coordinator-View, but the Action-Coordinator-View has the extension that it can catch mouse movements. This Multi-Level-View allows the arrangement of objects with a beginning time and a duration. The high level of the object is requested by the Action-Resource-Controller, which requests it from the Multi-Level-Model.

Back to the view expansion. After the Multi-Level-Model and the Multi-Level-View is created, the Action-Resource-Controller adds the view to the planning window and registers it to retrieve scrolling/zooming information. Then it observes the Discrete-Resource-Constraint by adding itself as a Change-Listener to the constraint and its Interval-List. After this, it simulates a data change in the model, by calling *DiscreteResourceConstraint.fireIntervalListUpdate()*. This data change causes an *updateDatamodel()*-call in the Action-Resource-Controller, which reads the tasks from the model and puts all in the Multi-Level-Model.

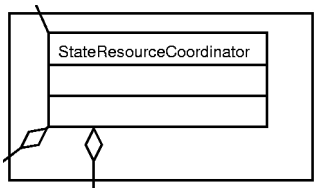


Like the Action-Semantic-Graphic, the single Action-Task can also be moved by the user. Therefore, the user can drag an Action-Task-Semantic-Graphic and drop it at the desired place. This moves only one Action-Task and not all the tasks in an action like the drag&drop with the Action-Semantic-Graphic.

[6]DISCRETE-RESOURCE-CONSTRAINT is the old notation for the ACTION-RESOURCE-CONSTRAINT.
[7]See Section *5.3 Action-Plugin* for more details about the Multi-Level-Model.

Moving an Action-Task-Semantic-Graphic to another action resource is not supported.

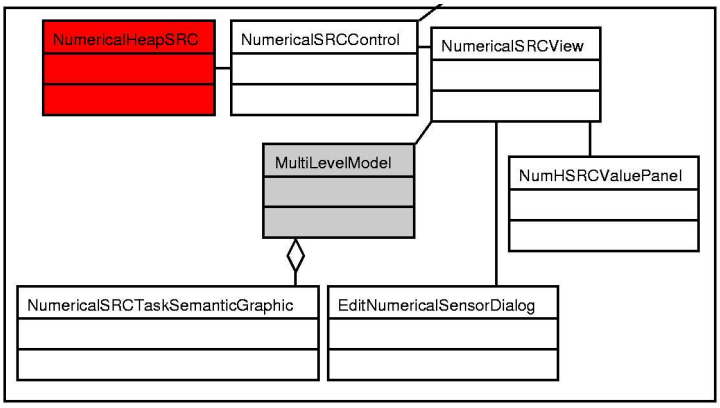
5.6. State-Resource-Plugin



The State-Resource-Coordinator is very similar to the Action-Resource-Coordinator, with one exception: the State-Resource-Coordinator has to handle two types of state resources with different subclasses: Numerical-Heap-SRC or Symbolic-SRC. However, both types are shown in the same state resource list.

When the State-Resource-Coordinator gets informed about the creation of a new State-Resource-Constraint, it checks if the subclass is equal to the Numerical-Heap-SRC or to the Symbolic-SRC. It then creates either a Numerical-SRC-Control or a Symbolic-State-Resource-Controller. Both implement the Resource-Controller interface, which provides support functions for using it as list entry in the state resource list.

5.6.1. Numerical-State-Resource



The bottom-level-agent Numerical-Heap-SRC consists of five parts:

- The *Numerical-SRC-Control*, which is represented as a list entry in the state resource list and is the interface to State-Resource-Coordinator.
- The *Numerical-SRC-View* represents a detailed view on the State-Resource-Constraint. It extends a Multi-Level-View.
- Inside of the Numerical-SRC-View the *Num-HSRC-Value-Panel* paints a horizontal line in dependency of the value and, of course, the value itself as a real number. The level of the line is adjusted as follows: If the present value is the minimum value zero, then the line is on bottom of the Num-HSRC-Value-Panel. The line is on top of the panel, if the present value is the highest value up to now in the planning process.
- To align the state tasks[8] in the Numerical-SRC-View a *Multi-Level-Model* is created by the view (This Multi-Level-Model is only for GUI purposes and has no connection to the

[8]Precondition tasks are so far not supported by the planning engine for Numerical-State-Resources.

real data model.).

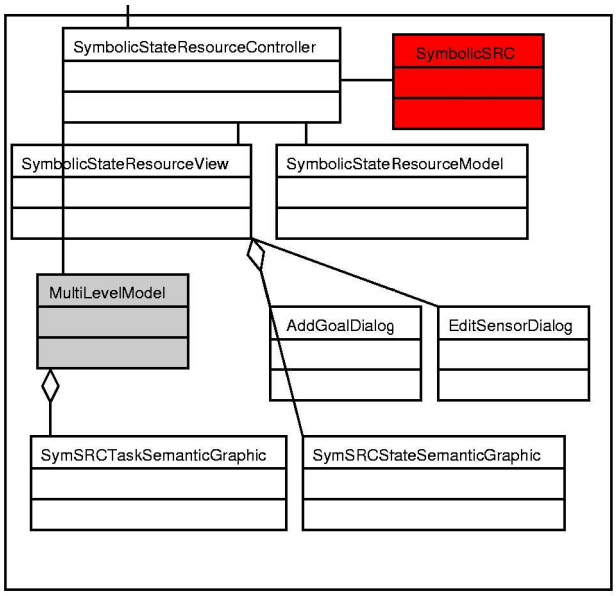
- The objects inside the Multi-Level-Model are *Numerical-SRC-Task-Semantic-Graphics*, which represent the state tasks connected to the State-Resource-Constraint.

The Numerical-SRC-Control is instantiated whenever a State-Resource-Constraint with a subclass Numerical-HSRC is created. The Numerical-SRC-View and Multi-Level-Model are only instantiated if the user clicks on the representation of the Numerical-SRC-Control in the state resource list.

Additionally, the user can enter sensor data. For that purpose, an "edit sensor data button" is displayed in the Numerical-SRC-View beneath the view's name. When the user clicks on it, the Edit-Numerical-Sensor-Dialog opens, and he can enter a real number. An input verifier checks every key stroke, if the entered value is a correct real number. By clicking the "ok button", the Edit-Numerical-Sensor-Dialog sets the state variable with the value entered.

All tasks are displayed with a beginning time of zero, because the Numerical-HSRC of the EXCALIBUR model does not support the projection over the time. Furthermore, it does not support goals at the moment.

5.6.2. Symoblic-State-Resource



The second subclass of the State-Resource-Constraint is the Symbolic-SRC. Whereas the Numerical-HSRC contains a real number as value, the Symbolic-SRC sets its value as one of several predefined states like "Door open" or "Door closed".

The subsystem of the Symbolic-State-Resource consists of eight parts:

- The *Symbolic-State-Resource-Controller* is the interface to the intermediate-level-agent State-Resource-Coordinator. It is represented as a list entry in the state resource list. Like the Numerical-HSRC-Control it also implements the interface Resource-Controller so that it can handle the *onClick()* requests by the list.

- When clicking on the representation of the Symbolic-State-Resource-Controller in the state resource list, the controller instantiates the *Symbolic-State-Resource-Model*.
  This model is different from the GUI models of the other plugins: The normal GUI model, like the Multi-Level-Model, is updated whenever anything changes in the data model

of the engine. Then it reads either the changed element or, in complex situations, it reads the complete part of the data model. During the implementation of the Symbolic-State-Resource it becomes clear that this approach causes deadlocks. The problem is the Interval-List attribute in the Symbolic-SRC. To get a certain interval of the time, the whole Interval-List must be accessed. A direct access is not possible because of the list structure. In addition, the EXCALIBUR engine also very often accesses this Interval-List. From this it follows that two threads (the engine thread and the repaint thread of the GUI) access the Interval-List in an uncontrolled manner. Locking the Interval-List by the Java synchronized statement causes a dramatic slowing down of the application: The GUI freezes and the planning process looses approximately 90 percent of its speed. Besides the waiting time of the GUI caused by locking the Interval-List by the planning process, the Java memory model probably is responsible for the slowing down of the planning process. While using the synchronized statement, the JVM (Java virtual machine) has to copy the data of the shared memory area to a local memory area for every thread.
However, due to the nondeterminism and the lack of time for developing this single subsystem for one of many plugins, it was not possible to find the definite reason for the deadlock. For that purpose, a special tool, like the Nondeterminator-2[Sta98] would be needed, which was not available for the programming language used. Normal debugger do not support the data race detection (*detection of data races is computationally to difficult to be done in a practical debugging tool* [Sta98]), particularly if one thread is fully nondeterministic like the repaint thread which requests random repaints (from the application point of view).
So the Symbolic-State-Resource-Model follows another approach: If something changes in the data model, the engine makes a callback to the Symbolic-State-Resource-Controller (see Change-Listener). Instead of recognizing the change and returning immediately, the Symbolic-State-Resource-Controller (which is now running in the thread of the planning engine, not in the repaint thread) makes a clone[9] of the complete Interval-List. After the cloning is completed, it changes the reference to the new data model. The repaint thread can now read from his own copy of the data model.

- The *Symbolic-State-Resource-View* represents a Symbolic State-Resource-Constraint in the planning window. It expands when the user clicks on the list entry of the State-Resource-Constraint. The main difference is that intervals with names inside are painted, instead of the horizontal line at the Numerical-SRC-View. These intervals show the states of the Symbolic-SRC over the time and are represented as *Sym-SRC-State-Semantic-Graphic*. The difficulty at the implementation is that there can be two or more intervals with the same state in a row. This has to be filtered by the Symbolic-State-Resource-View so that the intervals with the same state are displayed contiguously as one state.
  In addition to the intervals, the tasks are painted below the intervals. There are two types of tasks: the precondition tasks and the state tasks. Both are represented by the Sym-SRC-Task-Semantic-Graphic, but in different colors at a time. These tasks are connected by a vertical line to the state intervals. To align the tasks so that they do not overlap, a Multi-Level-Model is used. Every task is inserted with a fixed duration in this model, because up to now all the tasks have a duration of zero. Later, when tasks support variable durations, this can be easily adapted in the GUI by removing the code line which overwrites the present duration.

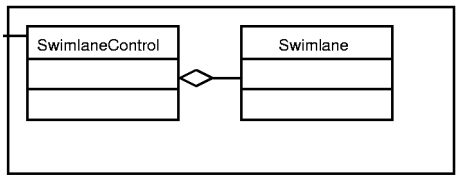
The requirement that the tasks can be moved between different resources results in a more complex drag&drop technique than the easy local drag&drop technique at the Multi-Level-View drag&drop. While local drag&drop works finely at the Multi-Level-View, because

[9] The clone is an intelligent deep clone: Only the necessary parts are cloned by the Symbolic-State-Resource-Model.

the mouse cursor can never leave the Multi-Level-View screen area (moving outside is blocked) and, therefore, it is easy to catch all mouse events, it is impossible to catch all mouse events at a global drag&drop, where the user can move the mouse cursor around the whole screen.
This is the reason why the Java native drag&drop technique is used. This is more complex but the only solution. For that purpose, each Sym-SRC-Task-Semantic-Graphic creates a Drag-Gesture-Listener[10] which recognizes if this object is dragged, starts the drag operation, and removes the task from the data model. This operation registers the dragged object at the JVM and changes the mouse cursor to a drag&drop icon.
On the opposite side, the Symbolic-State-Resource-View registers a Drop-Target-Listener which checks incoming drop requests. If a request is allowed, the Drop-Target-Listener inserts the task in the data model and sends a *dropComplete()* to the JVM. This drag&drop works inside the whole application, but not between different applications.

- Furthermore, the user can enter sensor data. For that purpose, an "edit sensor data button" is displayed in the Symbolic-State-Resource-View. When the user clicks it, the *Edit-Sensor-Dialog* opens, and the user can select a state from a pulldown menu where all possible states are listed. While clicking the "ok button", the Edit-Sensor-Dialog sets the state variable to the state type entered.
- A new type of dialog is the *Add-Goal-Dialog*. Goals can be set so that the planning engine takes it as a target for the planning process. If the user clicks on the "add-goal" button inside the Symbolic-State-Resource-View, the Add-Goal-Dialog opens, and the user can select a state which should be the target for this State-Resource and for the whole planning process. Additionally, the user can enter a time, when the goal must be reached. This field is already filled in by the current time multiplied by 1.10 to have a target in the near future.

5.7. Swimlane-Control



A swimlane connects two or more objects, like precondition and state tasks, with the corresponding action. It is also responsible for all other line painting operations, like the dashed line of the current time, or the connection with a vertical line between the states and the state tasks at the Symbolic-State-Resource-View.

The Swimlane-Control is not a plugin, because it is part of the Plan-Presentation package. However, it shows a lot of information and has its own special representation.

The lines are painted on a glass pane named *GlassPaneArea*. This transparent area lies on top of the content pane, which is the whole JFrame window without the menu bar. Other approaches like painting the lines directly into to the JPanels failed, because the faulty repaint handling caused screen flickering.

Every object, on which a swimlane shall be docked, must implement the *Swimlane-Connector* interface. This provides functions like *getConnectorX()* and *getConnectorY()* to retrieve the

[10] Drag&drop is different from operation system to operation system. The Drag-Gesture-Listener hides this to the developer.

absolute coordinates where the swimlane shall dock.
First, an object must register itself by calling *registerConnector(Object object, SwimlaneConnector sc)*, where *object* is the object from the data model to represent and *sc* is the representation for this object in the GUI.

After this, a swimlane can be requested by *requestSwimlane(int connectionType, SwimlaneConnector ownerConnector, List connectorList)* where the *connectionType* is a number to identify the connection, if one object requests more than one swimlane. *OwnerConnector* is the owner of this swimlane and *connectorList* is a list of all objects which shall be connected to this swimlane.

The Swimlane-Control tries to find the representation for every object and requests their coordinates. The coordinates are in the planning-panel point of view so that they have to be translated to the absolute coordinates in the planning window. Additional clipping is required, because the planning-panel is inside a scroll pane and the swimlanes without clipping would overpaint the scrollbars or other GUI elements in the window.

The Plan-Plugin reports, over the Plan-Presentation to the Swimlane-Control, a percentage value where the dashed line shall be painted on the screen. For that purpose, the Plan-Plugin computes the needed percentage value from the left and right boundary of the visible planning area. The Swimlane-Control itself does not know anything about the timing in the planning process or the visible area in dependency to this. It checks only if the percentage value is greater than zero, then it translates it into an x-coordinate and paints the dashed line there.

The last application domain for the Swimlane-Control is the vertical line, which is painted to support the alignment of tasks while dragging and dropping it. To avoid screen flickering the painted line stays for ten repaint cycles, if no new vertical line is requested by another mouse movement.

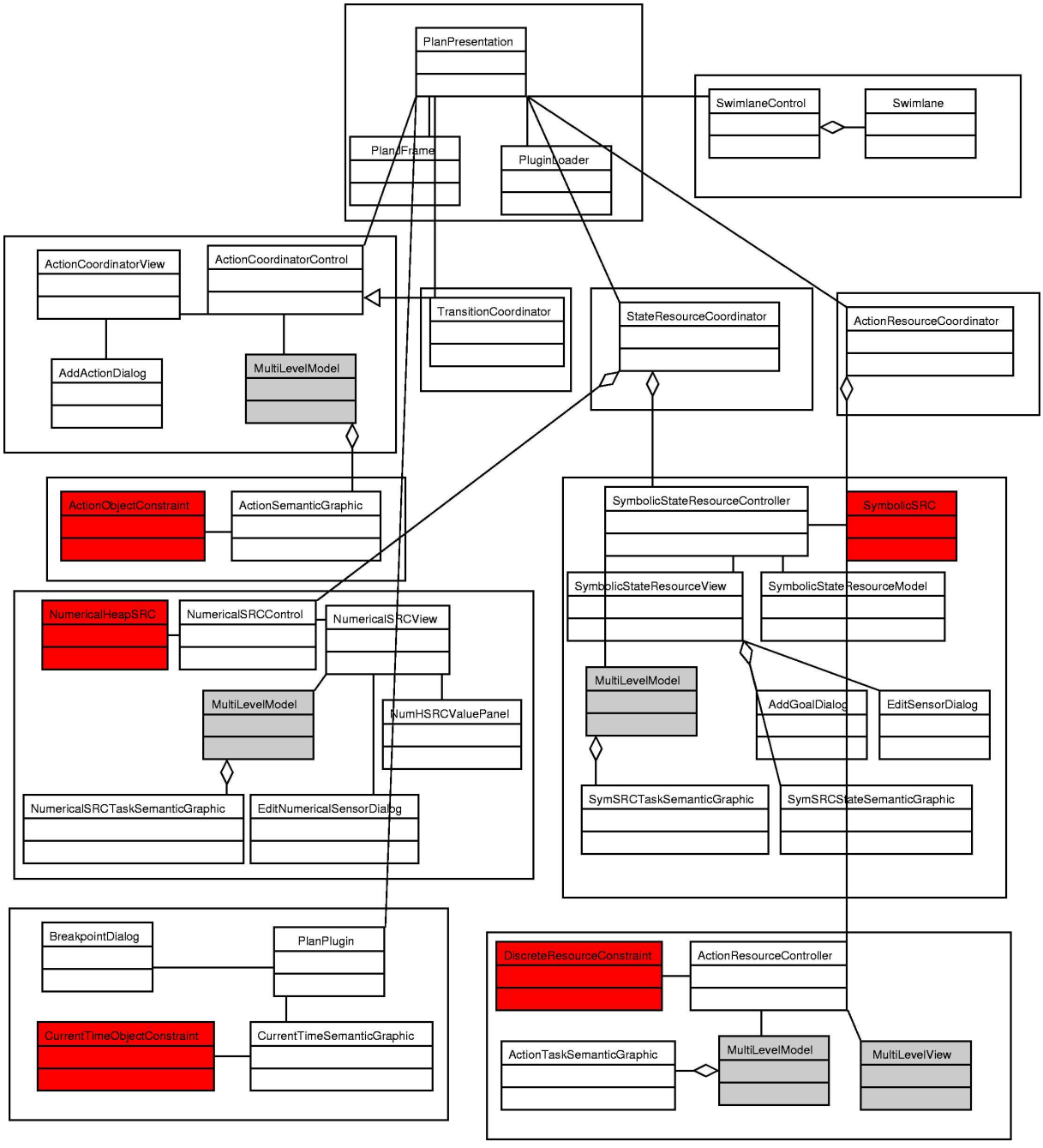


Figure 5.1.: Detailed overview about the Plan-Presentation application.

6. Evaluation

This chapter evaluates the developed Plan-Presentation application. First, a few complications, which appeared during the implementation process, are described with their corresponding workarounds. After this, the divergence from the specification is explained and justified. Then the targets reached by the graphical user interface are listed. At last, an assessment of the application is made, and future work for developers who extend the application is described.

6.1. Implementation Complications

The main problems at the implementation were caused by the multi-thread architecture and undocumented Java virtual machine details:

- As described in *5.6.2 Symbolic-State-Resource*, there occurred deadlock situations or the planning speed slowed down dramatically when the repaint-thread and the engine thread both accessed the SymbolicSRC.IntervalList at the same time. The outcome of this was a detailed analysis of the Java memory model specification and the reimplementation of the Symbolic-State-Resource-Model with another concept than the planned one, which avoided the deadlocks.

- Another problem was caused by unpredictable behavior of the Java swing implementation: While implementing the plugins there were no graphic errors and no screen flickering, but when adding the swimlane subsystem the whole graphic system was broken-down: The menu was no longer usable and the panels (which represent an action) were painted with random memory content.
  The reason was the glass pane, which lies transparent over the planning window, where the lines were painted. Replacing the glass pane by a normal content pane was not possible, because this caused screen flickering. After three days of workarounds and failure search, the solution was very simple: Instead of overwriting the *paint()*-function in the GUI-elements, the *paintComponents()*-functions were overwritten and the graphic errors disappeared. The technical reason was that the *paint()*-function is not allowed to use a lot of processing time. This technical detail was not documented by the Java swing developers, but the Java newsgroups were very helpful.

The second type of implementation problems were caused by the underestimation of the complexity of the EXCALIBUR concept. While the abstract concept looks difficult, but is clearly to understand, the complexity grows during the implementation. An example for this is an action: From the EXCALIBUR documentation it was clear that an action consists of one Action-Object-Constraint and a set of connected tasks, but the check which tasks are connected is more complex at the implementation level. The Plan-Presentation application has to observe the Action-Object-Constraint.connections attribute, where the State-Task-Constraints are connected. These State-Task-Constraints have a connection to a Resource-Type variable, which is connected or not to a State-Resource-Constraint. The whole path over four nodes has to be checked and tracked for updates and this for every connected task.

6.2. Specification Divergence

The main divergence from the specification is, that the lines between the GUI elements are not below the elements but above. The reason for this is that during the complex architecture design and taking into consideration the extensibility, a plugin-concept was chosen, which allows the views to latch in the application. It was, thereby, very important that every view consists of one closed panel. The wrong painting order was overlooked by the author and first recognized in a team meeting. Then it was too late to reorganize the whole architecture and the plugins which were implemented up to now.

Other divergences are that there is no goal-adding feature at the Numerical-State-Resource, because the engine does not support this until now. Additionally, the drag&drop of tasks between different state resources is not working correctly. The Plan-Presentation application removes and inserts the task at the respective state resources, but it is not shown immediately. The error search is in progress.

The last divergences are improvements which were visible after the first iteration of the implementation. For example, there are fewer buttons at the left side of the planning engine, but instead the buttons like ”add goal” or ”edit sensor data” are directly in the views of the resources. Additionally, the zoom slider was changed from a fixed value to a relative value: Moving the slider to the right border allows the view over the whole planning process, regardless of the length of the plan. Moving it to the left border results in a fixed value of x-pixel per time unit. Every adjustment between is scaled.

6.3. Targets Fulfilled

The properties requested in Section *1.3. Important Properties for a Planning GUI* for a graphical user interface are all fulfilled.

- The abstract, high-level view of the whole planning process is available with the Action-View.
- More detailed views are available with the Symbolic-State-Resource-View, the Numerical-Heap-State-Resource-View, and the Action-Resource-View.
- Zooming is also working fine, moreover there is even an auto-zoom function available.
- The planning speed is also influenceable on multiple possibilities.
- The breakpoint support is also given.
- The changes in the planning internals are realized in real time on the screen.
- There can also be sensor data changed at the Symbolic-State-Resource just as at the Numerical-Heap-State-Resource.
- Additionally, the goals can be set and actions can be moved.
- The user input is immediately processed and transmitted to the data model.

Furthermore the goals of this thesis, defined in Section *1.4 Goals of this Thesis* are all reached.

- The new graphical user interface for Excalibur-agents is working and the graphical user interface does not freeze, while the engine is computing the plan.
- As mentioned before, changes in the planning internals are immediately displayed on screen and vice versa, the user input is immediately transmitted to the data model.

- Due to the Presentation-Abstraction-Control pattern, the GUI is very extensible and enables the loading of future plugins. This was also the main requirement in the nonfunctional part. Every AI functionality, which can be displayed as a view, a menu entry or a control element can be inserted by a new plugin. Thereby, the control element can any type of a Java swing lightweight component[1], which can be a zoomslider or a container with more than one button, like the JPanel for the Real-Time-Control. Complications arise at the extensibility if a plugin requests interaction with other plugins like new AI objects which should be inside an existing plugin view. The interaction between plugins is solved at the Swimlane-Control, but an other functionality than lines between components would cause major implementation effort in the Plan-Presentation top-level-agent.
- Furthermore, due to the subscriber-observer pattern there are no dependencies from the GUI to the EXCALIBUR-engine. This was also part of the nonfunctional requirements. At central points[2] of the DragonBreath engine, there are three functions inserted: *addChangeListener(), removeChangeListener()*, and *fireChange()*. With this approach, any other graphical user interface can dock the engine without changing anything. If no subscriber is registered, the additional memory usage is zero and the additional CPU usage is minimized to calls of the *fireChange()* function, which returns immediately.

Requirements disposed in Section *2.3. Interface Design Requirements* are also fulfilled. The application development was orientated towards the given usability heuristics. A direct manipulation of the data on screen is possible by the drag&drop functionality. The user is familiarized with the application by using the MS Windows Look and Feel at MS Windows platforms, because Java swing only supports this Look and Feel for this platform. At other platforms, the native Look and Feel is used. Furthermore, the other features described in detail in the Section *2.3* are also fulfilled.

The use cases defined in Section *2.2. Identifying Use Cases*, which are the detailed functionality specification, are of course also fulfilled, because the software engineering process was straightened on them.

6.4. Assessment

The architecture concept of the Plan-Presentation allows an extreme extensibility. Every plugin can insert menu entries, control elements or views. Additionally, the Plan-Presentation top-level-agent provides easy services for the plugins to retrieve information from the data model. Furthermore the subscriber-observer-pattern works very well at the Plan-Presentation and makes the DragonBreath engine independent from the GUI. The developing of the system architecture and the GUI concept took nearly 50 percent of the working time of the diploma thesis. This is justified, because the whole concept is designed for future extensibility.

Beyond this, the architecture is also well to understand because of the tree-structure (top-, intermediate-, bottom-level-agents) of the presentation-abstraction-control pattern.

During the implementation, it showed that the code reusability between plugins was better than expected. This results in a GUI code library, which is contained in the com.ai-center. planpresentation.utilities package. For example, the Multi-Level-Model and the Multi-Level-View can be used by any plugin which needs the representation of parallel tasks with a timespan.

[1] Java swing differs in heavyweight components, which have a peer in the native programming language of the platform and lightweight components, which where fully implemented in Java. All subclasses of JComponent are lightweight components.

[2] The central points are the top-level classes of the inheritance hierarchy.

There are also disadvantages of the Plan-Presentation:
Due to the lack of time it was not possible to implement the best solution for every plugin. With much more time available there would be another concept for the Symbolic-State-Resource subsystem. The cloning approach is working well, but a more filigree approach with catching only the updates would be more consistent and would comply more with the art of software engineering.

Another point is the interaction between plugins. At present, the communication between plugins is made only for line painting purposes between objects. A more generalized approach could be helpful in the future, but this would have caused a much higher implementation effort. Only some workarounds allow the misusage of the Swimlane-Control as generalized line painting subsystem, which also paints the dashed line of the current time.

Additionally, the naming conventions could be improved. During the implementation the difference between Symbolic-State-Resource-Controller and Numerical-HSRC-Control was overlooked. A renaming at the end of the implementation was not possible until now because of time famine.

At last, there is the hope that the future plugin developers do not have problems with unpredictable Java swing features, like painting random memory content when adding another subsystem. But it is to be expected that no developer has to change such global implementation details in the Plan-Presentation.

6.5. Future work

The main work in the future will be new plugins if new features are added to the EXCALIBUR model. For that purpose, a Plugin-Programming HOWTO is contained in the appendix.

Furthermore, the *Action-Semantic-Graphic* must be adjusted, if features are added to the AI model, because this GUI element aggregates a lot of information from all data structures in the data model.

It would be also interesting to extend this AI ”debugger” to an AI ”runtime environment”, where new actions can be defined or new resources can be created. As the Plan-Presentation gets all information through the data model, it is possible that a new plugin enables the creation of new resources and the Plan-Presentation shows them without any further programming.

A. User and Developer Manual

This is the user manual for the Plan-Presentation application. Users who work with it only for demonstration or debugging purposes can skip Section *A.1 Starting Plan-Presentation*, because this section is only interesting for EXCALIBUR developers with Java knowledge.

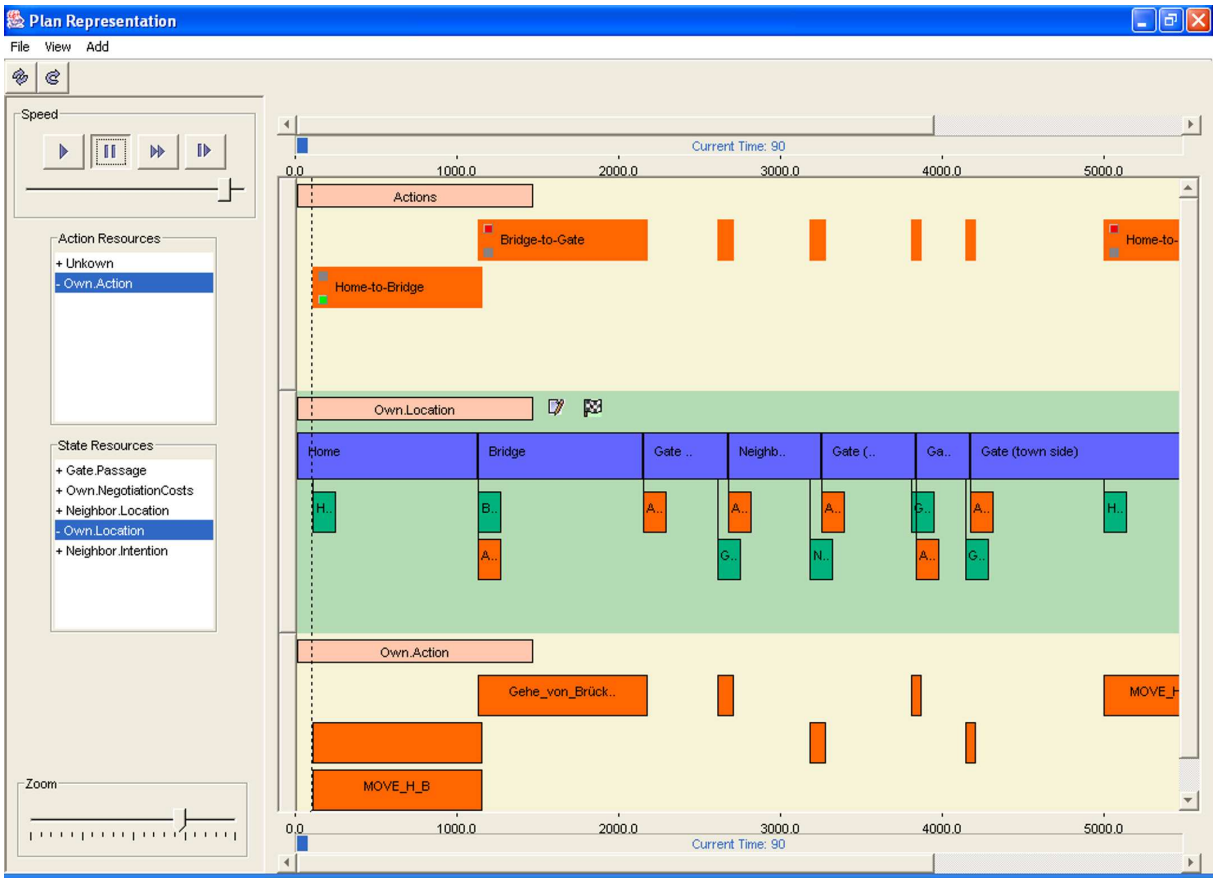


Figure A.1.: The Plan-Presentation application with expanded resources.

A.1. Starting the Plan-Presentation

At first, the Plan-Presentation application must dock the Global-Search-Control. This is done by inserting the following code in the EXCALIBUR planning scenario file and running it.

```
PlanPresentationAPI p = new PlanPresentation((GlobalSearchControl) ↩
    ↪ gsc);
```

It can be inserted at any place where the Global-Search-Control is already instantiated but the improvement-loop has not yet been started. Existing nodes from the CSP-graph will be read by the Plan-Presentation application. This line will also open the planning window.

A.2. Controlling the Planning Speed

When starting the Plan-Presentation application, the planning process is in pause mode. Normally, it has not started until then.

The usage of the planning speed control is similar to a CD-Player. The planning process can be started by clicking on the ”play-button” or on the ”single-step-button” on the right side. To stop the engine again the pause-button is to be pressed.

To influence the planning speed the slider can be moved to the right for faster or to the left for slower planning. Alternatively, the ”fastforward-button” can be pressed, then the planning process runs with maximum speed. A disadvantage is that this takes a lot of CPU resources and blocks the GUI sometimes.

Figure A.2.: The Real-Time-Control interface.

While the planning process is in progress, a dashed vertical line runs from the left to the right side of the screen. This indicates the current time of the agent. At the beginning, the line is not visible, because the current time is zero. When the line disappears on the right border of the window, the zoom of the plan can be adjusted.

For that purpose, the zoom slider is available. Moving the slider to the left side gives a detailed view on the plan to see where exactly an action begins or ends, whereas moving it to the right gives a full view on the whole plan. This calibration also activates an auto zoom which adjusts the zoom level so that the whole plan is also visible if the maximum planning time is extended by inserting an action later than the existing maximum planning time.

Figure A.3.: Moving the zoom slider to the right activates auto zooming so that the full plan is always visible.

A.3. An Abstract Overview on the Plan

During the planning process, rectangle boxes appear on top of the planning area. These boxes are the actions which are inserted by the agent into the plan. Actions on the left side of the dashed line will be no longer changed, because they have already been executed. A traffic light at every action indicates if the action can be executed (a green light is glowing) or if there are any inconsistencies like failed preconditions (a red light is glowing).

Figure A.4.: The present plan of the agent.

A.4. Viewing Resources

On the left side of the planning window, all resources are listed. The Action-Resources are exclusively used by the agent itself, like its arm, while the State-Resources can also be used by other agents and receive sensor data as the level of a fuel tank.

Figure A.5.: The resources lists.

When clicking on such a list entry a detailed view opens in the planning area of the window. There are three types of views available:

- The *Action-Resource-View* (see Figure A.6), which represents exclusively usable resources by the agent, like its arm. Only one task can contribute to this resource at a time. An overlap indicates a planning inconsistency. The name of the resource is in the upper left

corner of the written view, and the rectangles are the tasks. The text inside a rectangle is the name of a task.

- The *Symbolic-State-Resource-View* (see Figure A.7) represents a resource with a state as value. This can be, for example, ”door open” or ”door closed”. The states are below the name of the resource and are represented as a set of rectangle boxes which are joined together. Below the states again, the tasks are visible. A turquoise box is a precondition task, while a orange box represents a state task. When moving the mouse over the task, additional information pops up, like the exact precondition description or the contribution of the state task.
- The last view is the *Numerical-State-Resource-View* (see Figure A.8), which represents a resource with a real number as value. This can be a fuel level, for example. The state tasks are similarly represented as in the Symbolic-State-Resource-View. Instead of the states, a horizontal line indicates the value of the resource. A line on bottom displays a value near zero, while a line on top describes the highest value of this resource until now. If a new higher value occurs, the line stays, but if the value sinks, the height of the line also decreases. Additionally, the real number is also displayed.

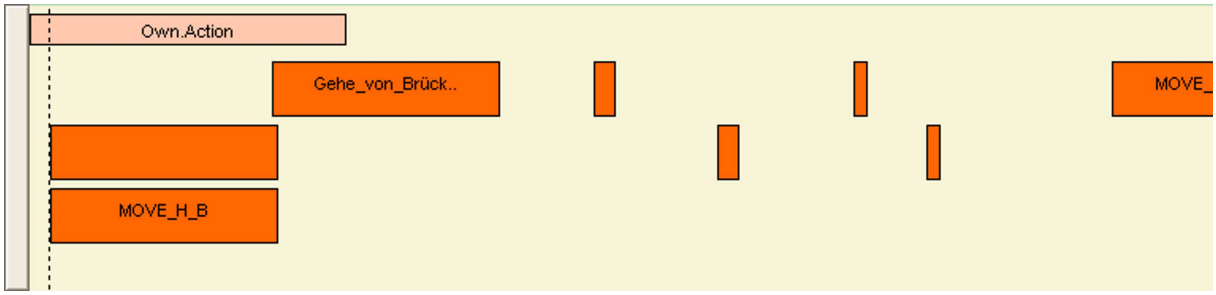


Figure A.6.: An Action-Resource, which is always exclusively used by the agent.

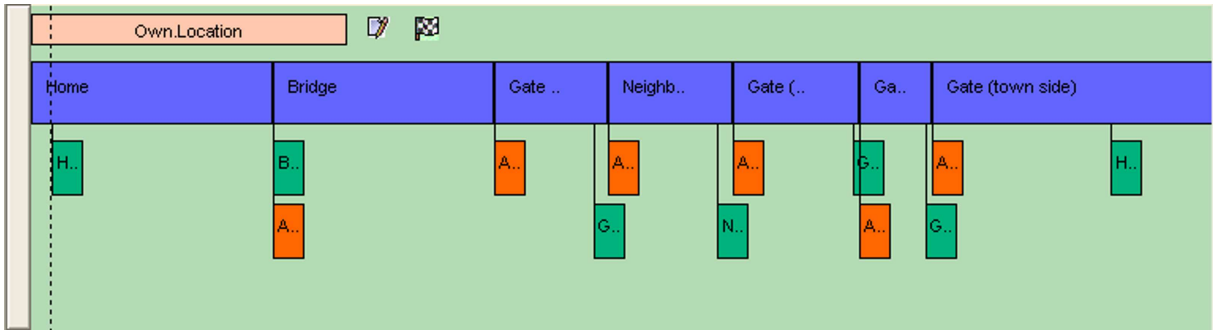


Figure A.7.: A Symbolic-State-Resource which consists of states, like being at *home* or the *bridge*.

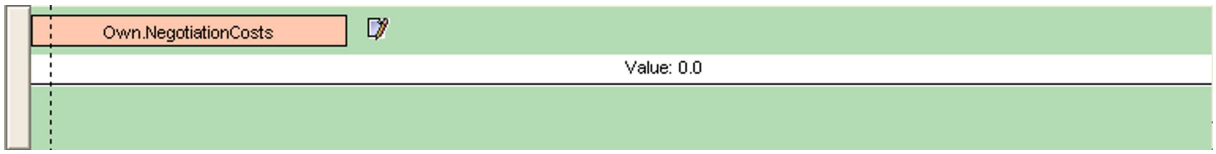


Figure A.8.: A Numerical-Heap-State-Resource which displays a real number as value, like a fuel level.

To see which task belongs to which action the user can move the mouse cursor over the action. A line connects the action and all tasks which belong to this action. A click on the action sorts all those resources which have a task connected to this action to the top.

The sorting can also be started by the buttons in the menu bar. For that purpose, a button "sortmode update" exists where all resources, which are recently accessed, are sorted to the top, whereas the button "sortmode grouped" groups Action-Resources and State-Resources. This means that all Action-Resources are sorted on top, while all State-Resources are below.

For a detailed information which tasks belong to the actions, the showing of all lines can be activated. For that purpose, the menu entry $View \rightarrow Show\ Swimlanes$ must be clicked, and to remove it again $View \rightarrow Hide\ Swimlanes$ must be clicked.

A.5. Showing Transitions

Transitions are external effects, like the overflowing of a bath tube, if too much water is inside. The transitions are very similar to actions, with one exception: They do not have an action task, because they do not allocate a resource, which is owned by the agent, like his arm.

The transition view can be activated by clicking on the menu entry $View \rightarrow Show\ Transitions$ and removed again by clicking on $View \rightarrow Hide\ Transitions$.

A.6. Adding a Goal

The planning process can be influenced by adding a goal to the agent. A goal can, for example, be that, at time tick 124, the door must be opened. There can be even more goals defined for a single planning process. The agent tries to find a plan to fulfill these goals. However goals are only possible for Symbolic-State-Resources and not for Numerical-Heap-State-Resources or Action-Resources.

To add a goal first the Symbolic-State-Resource must be visible in the planning window. This is done by clicking on the list entry in the state resource list. After that a click on the checkered flag the Add-Goal-Dialog (see Figure A.9) opens.

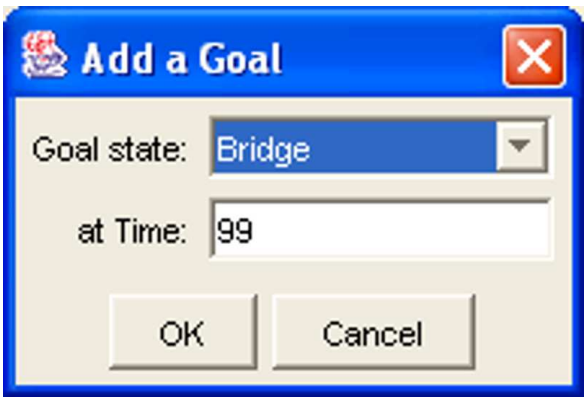


Figure A.9.: The dialog to define a goal for a Symbolic-State-Resource.

The time when the goal must be reached is already filled in, but it can be changed into the desired time. Then the state can be chosen in which the state resource must be at the given time. After pressing the "ok button", the new goal is defined.

A.7. Entering Sensor Data

Another possibility to influence the planning process is entering sensor data. With this option a environment can be simulated.

Sensor data can only be entered for Symbolic- and Numerical-Heap-State-Resources. Of course Action-Resources do not support sensor data, because only the agent defines its contribution.

The sensor data for a Symbolic-State-Resource is the state at the current time. For entering sensor data the state resource must be visible in the planning window. This is done by clicking on the list entry in the state resource list. After clicking on the sensor data button a dialog appears, where the state can be chosen.

Figure A.10.: The dialog to enter sensor data for a Symbolic-State-Resource.

In contrast to the Symbolic-State-Resource a real number can be entered at the Numerical-Heap-State-Resource. Everything else is similar.

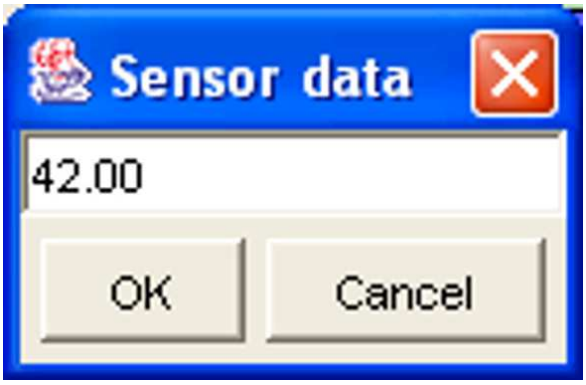


Figure A.11.: The dialog to enter sensor data for a Numerical-Heap-State-Resource.

A.8. Adding an Action

Until now, all described techniques are to simulate a real world environment for the agent. The following two sections are about giving the agent a hint to solve the planning problem.

At first, an action can be inserted to help the agent finding the plan. For that purpose, the menu entry *Add* → *Action* must be clicked. The mouse cursor then changes to a crosshair, and it can be selected, when the action should be start. This is done by clicking in the action view (where all actions are displayed), and the Add-Action dialog opens.

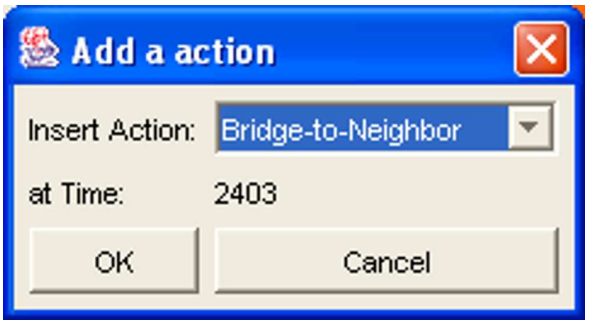


Figure A.12.: The dialog to add an action to the plan and to give the agent a hint.

From a list of all possible action types, one can be chosen, and after hitting the ”ok button”, the action is inserted in the plan. However, it is possible that the agent removes this action soon if it has another plan.

A.9. Moving Tasks

The second possibility to help the agent is to move tasks back and forward or from one resource to another.

Besides the task, actions can also be moved, thereto the mouse cursor must be over the action, and the left mouse button must be pressed. Than the mouse can be moved, and the action follows the mouse cursor. This can be done to tell the agent that it should execute the action earlier or later. Moving a task is similar to moving an action, but then only a task of the action is moved, for example, only the precondition is checked earlier.

The Symbolic-State-Resource-View supports also the movement of a task from one resource to another. Unlike the action or action resource moving, the task does not follow the mouse cursor, but the mouse cursor changes to a drag&drop symbol. The mouse cursor can now be moved with a pressed left mouse button to a resource, which must be visible in the planning window. When releasing the mouse button the task is dropped at the resource and inserted at the time which is indicated by the x-coordinate. However, the task is dropped without any further information whether or not it can be inserted here. This can only be asserted, if the task is visible soon after or not at this place.

A.10. Breakpoints

During the single step mode the engine stops at every breakpoint and waits for the user to jump to the next breakpoint. These breakpoints can be activated or deactivated with the menu entry *View* → *Show Breakpoints*. A dialog opens where the active breakpoints are marked with a checked box. The breakpoints can be enabled or disabled with a click on the checkbox.

The selection is committed by clicking on the ”ok button”.

A.11. Exiting the Planning Process

The exiting of the Plan-Presentation also exits the EXCALIBUR planning process. The application can be exited by clicking on the menu entry *File* → *Exit*.

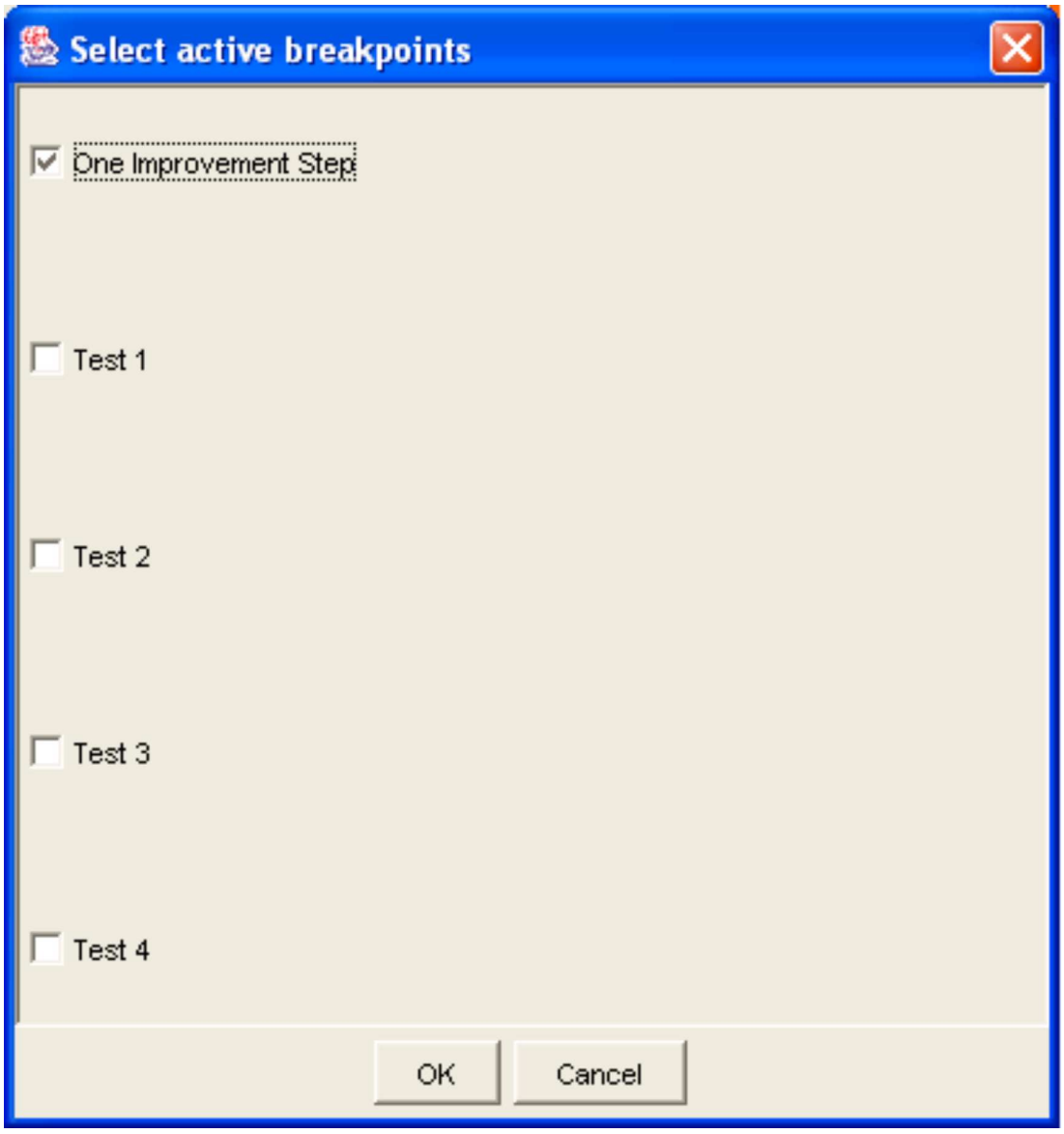


Figure A.13.: The dialog to enable or disable breakpoints.

B. HOWTO Plugin-Programming

This chapter is for developers with Java know how, who program a new plugin for the Plan-Presentation application. A plugin is necessary if new AI concepts are added to the EXCALIBUR engine.

As an example it is explained how the Action-Resource-Constraint-Plugin is developed.

B.1. Plugin-Programming Basics

At first, a new directory must be added at

```
com/ai_center/planpresentation/plugins
```

The file must be named with the suffix *Plugin.java* so that the PluginLoader can recognize it as a plugin. After this, the full qualified name of the class must be inserted into the array *classnames* of the file *com/ai_center/planpresentation/utiltities/PluginInitConstant.java* .

Additionally, the plugin must implement the *Plugin-Interface*.

```
import com.ai_center.planpresentation.plugins.Plugin;

public class ActionResourceCoordinatorPlugin
        implements Plugin
```

The plugin-interface requires the following functions:

- *initPlugin()* is necessary, because of the plugin loading concept the constructor of the plugin is not called. Therefore the plugin initialization is realized in this function.
- *addConstraint()* is called, when a new constraint is added to the DragonBreath data model.
- *removeConstraint()* is called, when a existing constraint is removed from the DragonBreath data model.

The first function to implement is the *initPlugin()*.

```
    public void initPlugin(PlanPresentationAPI _pc)
    {
        planPresentationAPI = _pc; // save the reference for later ↩
            ↪ use

        // register constraint types
        planPresentationAPI.registerConstraintType(
                Registry.DISCRETERESOURCE_CONSTRAINT, (Plugin) ↩
                    ↪ this);

        planPresentationAPI.addControl(
                initDefaultResourceList("Action Resources"), 15);
```

```
}
```

The function *planPresentationAPI.registerConstraintType(NodeType, Plugin)* requests a callback from the Plan-Presentation to the *addConstraint()* function, if a constraint with this NodeType is added to the data model. The second parameter is the plugin itself.

```
// register constraint types
planPresentationAPI.registerConstraintType(
        Registry.DISCRETERESOURCE_CONSTRAINT, (Plugin) ↩
            ↪ this);
```

After that, GUI elements can be added. In the left row of the window the *control* elements are displayed, while the *views* are shown in the planning window. Additional menus or menu items can be added at this place.

```
planPresentationAPI.addControl(
        initDefaultResourceList("Action Resources"), 15);
```

This line adds a JList (returned by the *initDefaultResourceList()*) to the GUI with the sorting number 15.

The adding of menu entries works as follows:

```
planPresentationAPI.addMenu("File", 0); // menu on the ↩
    ↪ left side
planPresentationAPI.addMenuItem(new JMenuItem(exitAction) ↩
    ↪ , "File");
```

First a menu is created with *planPresentationAPI.addMenu("File", 0)*. The name is "*File*" and the sorting number is *zero*. The menus are sorted from left to right ascending. After this a menu item is added with *planPresentationAPI.addMenuItem(new JMenuItem(exitAction), "File")*. The *exitAction* is an action which contains the name "Exit" and calls the *System.exit()* command if executed. The name where the menu item should be added is "File". However, the order of the commands is irrelevant, because the PluginLoader does not create the GUI elements until all plugins are loaded.

The adding of a view is explained in section B.4.

The plugin can also be connected to the internal messaging system of the Plan-Presentation. There are messages distributed, like changing the sort mode of the views. However, because it is not necessary to use this feature, here is only a short explanation given:

```
planPresentationAPI.addChangeListener((ChangeListener) ↩
    ↪ this);
```

This line connects to the internal messaging system. For that purpose, the class must also implement the *ChangeListener-Interface* and the appropriate *stateChanged()* function. Note: The internal messaging system does not send information about zooming or scrolling. This is explained in Section B.4.

Next, the *addConstraint()* functions must be implemented. This function is called, whenever a constraint is added for which the plugin has requested a callback.

All callbacks for every registered constraint type are collected here, so it is necessary to check which constraint type is to be added now. This is done with the *instanceof* operator. Note: The Discrete-Resource-Constraint is soon renamed to Action-Resource-Constraint.

```
public void addConstraint(Constraint constraint)
{
                                // at first, check which ↩
                                   ↪ constraint type this is
    if (constraint instanceof DiscreteResourceConstraint)
    {

        addActionResourceConstraint(( ↩
           ↪ DiscreteResourceConstraint) constraint);

    }
}

private void addActionResourceConstraint(
        DiscreteResourceConstraint actionResourceConstraint)
{

                                // create a corresponding GUI ↩
                                   ↪ element for this constraint
    ResourceController resourceController = new ↩
       ↪ ActionResourceController(
            (Coordinator) this, actionResourceConstraint);

                                // give it a name
    resourceController.setName(actionResourceConstraint. ↩
       ↪ getName());

                                //add it to the JList on the left ↩
                                   ↪ side
    defaultResourceModel.addElement(resourceController);

}
```

The *removeConstraint()* function needs to find the corresponding GUI element and removes all references (probably in a list of all known GUI elements) to it so that the garbage collector can free the memory.

Now the basic specification is complete. The reader should now be able to program plugins supporting lists of any constraint type.

B.2. The Coordinator and the Controller

Normally, a plugin handles more than one instance of a constraint. The *Action-Resource-Constraint-Plugin* for example manages a list of all available action resources in the data model.

For that purpose, it installs a list of resources on the left side of the planning window. Every list entry is an *ActionResourceController*, a minimized form of a constraint representation.

The coordinator creates the corresponding GUI element and tells it which constraint is his corresponding part in the data model.

```
    ResourceController resourceController = new ↩
       ↪ ActionResourceController(
```

```
                (Coordinator) this, actionResourceConstraint);
```

Only when the user clicks on the list entry, the controller expands a more detailed view in the planning window. Then the whole docking and observing mechanism starts to get all updates in the data model which are in context with the constraint.

```
protected void expandView()

                                    // create view
        multiLevelModel = new MultiLevelModel((Controller) this);
        multiLevelView = new MultiLevelView(this, getName());

                                    // add view to the planning window
        coordinator.addView((RetractableView) multiLevelView);
        coordinator.addViewChangeListener((ChangeListener) ↩
            ↪ multiLevelView);

                                    // observe data model
        discreteResourceConstraint.addChangeListener(this);
        discreteResourceConstraint.getIntervalList(). ↩
            ↪ addChangeListener(this);
    }
```

In the *expandView()* function of the *ActionResourceController*, first a MultiLevelModel is created. This is a supporting data structure aligning the tasks by shifting them on the y-axis so that no overlap appears on the screen. Secondly, the MultiLevelView is instantiated. This is the real view, which is visible on screen.

Next, the coordinator delegates the calls *addView()* and *addViewChangeListener()* to the PlanPresentationAPI. *addView()* simply adds the view to the planning window, while *addViewChangeListener()* connects the view to the internal scrolling and zooming event queue (for details about this, see Section *B.4 The Model and the View*).

At last, the controller must observe the data model. For that purpose, it connects to the DiscreteResourceConstraint and requests to be informed about changes in it or in the IntervalList (a list of all contained tasks at this action resource).

If something has changed, the DiscreteResourceConstraint calls the *stateChanged()*-function in the ActionResourceController which updates the view elements by rereading the data model.

```
    public synchronized void stateChanged(ChangeEvent e)
    {
            updateDatamodel();
    }
```

This concept is orientated at the Presentation-Abstraction-Control architecture. The coordinator is an intermediate-level-agent or is part of it. The controller is part of a bottom-level-agent. See chapter *4.2.1 PAC* for more details about it.

B.3. The Model

The observing of the data model is realized by the subscriber-observer pattern. For that purpose, each object to be an observer must contain the following functions:

- *addChangeListener()*
- *removeChangeListener()*
- *fireChange()*

Additionally, the attribute *listenerList* is required. These functions and this attribute can be copied from the Global-Search-Control, if they do not exist in the inheritance hierarchy.

The function *fireChange()* must be called whenever something important changes. But it creates, however, problems, if it is too often called, because in some modules, the update-mechanism requires at lot of computational time.

B.4. The View

Should some parallel tasks be aligned on a time-axis, the best solution is to use the com.ai_center. planpresentation.utilties.MultiLevelView class or to extend it.

However, an individual view can also be developed. Therefore two things are important:

- The view must extend the class com.ai_center.planpresentation. plugins.RetractableView, which provides the retract and expands functionality.
- The view must connect to the View-Change-Listening system of the Plan-Presentation, which informs all views about scrolling and zooming.

The following examples are taken from the SymbolicStateResourceView.

```
import com.ai_center.planpresentation.plugins.RetractableView;
import com.ai_center.planpresentation.plugins.View;
import javax.swing.event.ChangeListener;


public class SymbolicStateResourceView extends RetractableView
        implements
            ChangeListener,
            View
```

At first, the class must extend the RetractableView, implement the View-Interface and the ChangeListener-Interface.

Of course, the view must be registered to get the scrolling information. This is done by calling *coordinator.addViewChangeListener(myView)*.

The ChangeListener-Interface requires the *stateChanged()* function:

```
    public synchronized void stateChanged(ChangeEvent e)
    {
        if (e.getSource() instanceof BoundedRangeModel) // a  ↩
           ↪ message from the MainViewModel
        {
            BoundedRangeModel mainViewModelEvent = ( ↩
               ↪ BoundedRangeModel) e
                    .getSource();
            if (!mainViewModelEvent.getValueIsAdjusting())
            { // model makes no further adjustments
```

```
            // get the new values from the model
                mainViewModelValue = mainViewModelEvent. ↩
                    ↪ getValue();
                mainViewModelExtent = mainViewModelEvent. ↩
                    ↪ getExtent();
        }
    }
```

After a *stateChanged()* call, the new *mainViewModelValue* is the time tick at the left border of the screen, while *mainViewModelValue + mainViewModelExtent* is the time tick at the right border of the screen. The *stateChanged()* function is called whenever the planning window is scrolled or zoomed.

The second interface to implement is the *View-Interface*, the required functions are:

- *int getWidth()* - The width of the view.
- *int getHeight()* - The height of the view.
- *void scroll(int scrollstep)* - This function is called, from a component inside of the view, to communicate a scrolling request. The view should forward this request to the controller or coordinator.
- *void engineStop()* - This is also a delegate function and should be forwarded to the controller or coordinator.
- *int absoluteY(int relativeY)* - The implementation is normally very simple: *return controller.absoluteY(this.getY()) + relativeY.*
- *int getMainViewModelExtent()* - The width in time ticks of the visible screen area.
- *int getMainViewModelValue()* - The time tick on the left border of the visible screen area.
- *boolean isRetracted()* - True if the view is retracted and no painting is required.

Bibliography

[Bar03] Dr. Roman Barták. *Foundations of Constraint Programming - Tutorial 1 at ETAPS'03.* 2003.

[BD00] Bernd Bruegge and Allen H. Dutoit. *Object-Oriented Software Engineering.* Prentice Hall, 2000.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture - A System Of Patterns.* Wiley, 1996.

[KM99] Robert Kosara and Silvia Miksch. *Visualization Techniques for Time-Oriented, Skeletal Plans in Medical Therapy Planning.* In Proceedings of the Joint European Conference on Artificial Intelligence in Medicine and Medical Decision Making (AIMDM'99). Springer Verlag, 1999.

[KSdR02] Kishalay Kundu, Chad Sessions, Marie desJardins, and Penny Rheingans. *Three-Dimensional Visualization of Hierarchical Task Network Plans.* Proceedings of the Third International NASA Workshop on Planning and Scheduling for Space. Technical report, 2002.

[Lam99] Geoff Lambert. *Envisioning Timetables - The books of Edward Tufte.* The Times - Journal of the Australian Association of Time Table Collectors, December, 1999 Issue No. 189 (Vol. 16 No.12), 1999.

[MYWA02] Steven A. Morris, G. Yen, Zheng Wu, and Benyam Asnake. *Timeline Visualization of Research Fronts.* Journal of the American Society for Information Science and Technology (JASIST), 2002.

[Nar00] Alexander Nareyek. *Open World Planning as SCSP.* In Papers from the AAAI-2000 Workshop on Constraints and AI Planning, Technical Report, WS-00-02, 35–46. AAAI Press, Menlo Park, California. (available via the Excalibur project's webpage), 2000.

[Nar01] Alexander Nareyek. *Constraint-Based Agents* - An Architecture for Constraint-Based Modeling and Local-Search-Based Reasoning for Planning and Scheduling in Open and Dynamic Worlds. Reading, Springer LNAI 2062, 2001.

[Nar02] Alexander Nareyek. *Planning to Plan - Integrating Control Flow.* In Proceedings of the International Workshop on Heuristics (IWH'02), 79-84. Also appeared in Tsinghua Science and Technology 8(1): 1-7, 2003, 2002.

[Nie94] Jakob Nielsen. *Usability Inspection Methods.* John Wiley and Sons, New York, NY, 1994.

[NS03] Peter Norvig and Russel J. Stuart. *Artificial Intelligence : A Modern Approach.* Saddle River, N.J. : Prentice-Hall, 2003.

[Shn02] Ben Shneiderman. *User Interface Design - Deutsche Ausgabe.* Addison Wesley Lngman, Inc., 2002.

[Sta98] Andrew F. Stark. *Debugging Multithreaded Programs that Incorporate User-Level Locking.* Technical Report MIT/LCS/TR-750, Massachusetts Institute of Technology, 1998.

[WA96] David E. Wilkins and John Mark Agosta. *Using SIPE-2 to plan emergency response to marine oil spills.* IEEE Expert 11(6), December 1996, pp. 6-8., 1996.

Index