Planning the Construction of a Building

by

Martin Aronsson
Planning the Construction of a Building

by

Martin Aronsson

Knowledge Based Systems Laboratory; 1993-06-14

Swedish Institute of Computer Science
Box 1263
S - 164 28 Kista
Sweden
email: martin@sics.se

Abstract

The paper describes a tool for generating plans for the construction of a building. The application is implemented in GCLA, together with a simple constraint solving system. The main idea is that experiences from other plans are stored in methods; which are a systematic way of grouping activities together as higher level activities that can solve more complex tasks. Activities are entities that perform some action on a model of the real world, called the global state. Activities have preconditions, i.e. starting conditions, some representation of time and resource consumption, and postconditions, i.e. how and what to change in the global state. Scheduling activities amounts to allocating resources and placing the activities in time. The goal of the planning process, i.e. what we want the planning process to achieve, is represented by a geometric model of the changed global state, i.e. a design of the specified building that one wants to build.

To create plans, the system is divided into two main phases; the choice-of-method phase and the scheduling phase. In the choice-of-method phase suitable methods are chosen based on experience from the past. Such methods already exists in the building industry, although not in an explicit formal representation. Then the scheduling phase allocates resources and places the activities in time by reasoning about the activities' change of the global state. The goal of the planning process is that the objects of the specified design should be produced and represented in the global state.

The user can change most of the behaviour of the system by indicating what he wants it to do. He can change activities, their preconditions, calculations and postconditions, he can change methods, or add or remove activities to them, he can change resources etc. By this flexibility, the user can form his system to reflect his own preferences about how to plan and what to plan.

Keywords: Project planning, GCLA, scheduling
1. Introduction

For some years the project "planning and management at the construction site" has been going on at SICS. Members in the project group have been people from SICS, two architects of which one is a member of the department of Design Methodology of the Royal Institute of Technology, and an economist specializing in organisational structures. The project started in 1988, and was finished during 1992, with some seminars, and delivered a couple of videos together with a demonstration program. The demonstration program explains the overall idea without going into detail about how all its parts would be implemented. In the course of the project some prototype systems were implemented, one of which is described in this paper: the planning system and its prototype implementation.

The material presented herein is not a complete presentation of the prototype system. Some simplifications have been made to get a more presentable version, although most of the implementation and the ideas behind it are described.

The system is implemented in the prototype system GCLA. The ability to pose hypothetical question and reasoning about hypotheses makes GCLA a good tool for testing various ideas during the development of an application. GCLA has been presented in various papers, and we refer the interested reader to [Kre92, Aro91a] for further details about language specification, and to [Aro92] for basic programming methodology.

The planning system and the GCLA system have been developed in parallel, where the development of each system has gained from being tested on the other. The development of GCLA has benefited from a lot of experience of real world problems, as well as the opportunity to prove its usefulness in a real application. The planning system has gained understanding of the problems at hand, together with an executable specification as a result.

Also, the work presented herein should not be regarded as the best implementation and the final result. Rather, it should be regarded as a specification of a first working prototype of a planning system for the construction site. There are a lot of places where one could go further and develop more efficient solutions. However, the work presented constitutes a good specification of how such a system should behave. From this specification, each phase and each algorithm can be more efficiently implemented, and together with a good interface to the user, it should constitute a good tool for modelling the construction of a building. The application should be seen as an executable specification, where GCLA has served as the formal specification tool for modelling the behaviour and the declarative properties of the system.

But saying that the planning system specified in GCLA cannot be used for real world examples would be wrong. The system described is tested on some toy examples to see that the ideas coincide with reality, but the system has also been tested on a real case, a 40000 square meter office building in 8 floors, described in a database by about 2000 facts (only comprising the supporting parts of the building). It turned out that the plans produced by the system and the plans produced by the planner at the site mapped quite well, and that the planning system was able to adapt to the studied disturbances that appeared at the site. We do not by this mean that the system by itself figured out how to correct the plans based on occurred disturbances, but that the expressiveness and the possibilities for the user to specify how he plans and what kind of plan he wants suffice to mimic the way the user actually plans 'by hand'. It is still the user that is in command of the system, but the system does all the calculations while the user makes the critical decisions.
One could look at this application as giving the user a special 'planning language', in which he has certain constructs that he could utilize to represent his own planning knowledge. Methods hold the knowledge about how the user solves problems, and activities hold the user's knowledge about time consumption, resource consumption etc, together with what must be achieved in order to start the activity and what the result of the activity is.

Much of the representation of the building uses standards introduced by the Swedish building industry. One such standard is the BSAB system, where BSAB stands for "Byggandets Samordning AB", which would be something like "Coordination of Construction Ltd" in English. (BSAB was the first attempt to create an industrial standard for workmanship and functionality for the construction companies in 1972, but the standard has been further developed since then.) Table 2 of the BSAB system deals with the function of the building, which we have used in the implementation. We have also used table 1, which has the technical construction as basis, but that work is not presented in this paper.

The main purpose of this paper is to describe representational ideas, planning algorithms and properties of a project planning system at the construction site together with its implementation in GCLA, and not the cooperation between the user and the planning system. Therefore we will not develop issues related to the cooperation between the user and the system, but refer the interested reader to two different 'reports': a video tape [MDA], which can be ordered from SICS, where the issues of interfaces and user cooperation are discussed, and [Lau91] where a possible interface to the method database and the activity database is presented. The video is in Swedish.

When talking about how GCLA works, we will often make use of the term "some rule to the right" or "some rule to the left". The term "to the right" denotes (inference rules applied to) terms and structures to the right of the turnstile |-_. "To the left" is defined analogously.

The paper is divided into 6 main sections. The first section is a domain description, which briefly describes the application domain. The second section contains a system overview, which describes the basic ideas of methods and activities. The third section describes some additional primitive constructs added to GCLA. These primitives, notably index functions, are not described elsewhere. Then some different programming methodologies in GCLA are briefly presented, followed by the fifth section, where a detailed explanation of the implementation is given. The sixth section gives examples, followed by a discussion containing comments and references to interesting future projects.

2. Application Domain

To plan the construction of a building is a very complex task. In the ideal case an architect has designed a building in accordance with the wishes of his client. An entrepreneur is hired to erect the building, when the design is ready. It is often the case that several entrepreneurs have to compete to determine which one of them is cheapest, by giving an estimation of the costs, which forms the basis of a contract. This stage is called the calculation stage. It is easy to see that for the entrepreneur it is very important to give a correct estimation of the costs, and therefore he has to plan the construction in some detail already at the calculation stage.

At the calculation stage, a quantity calculation is done, which gives the different materials and construction parts in some values and entities, for example in volume, number of pieces etc. To get the overall cost, the calculator (the person who produces a calculation) has to help him statistical measurements of the cost of producing different parts of the
building. These statistical values are based on the experience of the entrepreneur from projects in the past.

As soon as the entrepreneur gets the job, he starts to plan the construction of the building. In this phase, the entrepreneur produces the production plans, i.e. the plans that on an abstract level show how the process of constructing the building will be performed. Those plans are also a legal contract in the sense that the time limits are specified as different milestones. To help him, the planner (the person that plans the construction) has a set of methods, which specify data about different activities together with how those activities are combined. Some of those methods are explicit, but a lot of them are based on the experience that the planner has gained from other projects.

When the construction process starts, the planner is almost finished with his job. It is now the site leader that takes over. His job is to manage the construction site in such a way that the production plans are followed, without violating the cost frames that the calculator has specified. Up until now, the calculator and planner have worked only on the description of the building (i.e. drawings, specifications etc). The site leader has to amalgamate both the costs and the plan, together with solutions to all the different kinds of problems and disruptions, jointly called disturbances, that will occur on the site. Examples of disturbances are bad weather, incomplete design, changed design, the actual efficiency is not the same as the calculated one, some supplier cannot deliver on time, the delivered material is of wrong quality, different groups of workers try to use the same area for different things, machinery breaks down... This list is mostly very large. It is very seldom that the production plans and reality coincide.

Therefore, the site leader and his staff make plans for one to three weeks at a time, aiming at following the production plans. One day each week they gather together to discuss the state of the process, and how to continue the next few weeks. These plans try to reflect the reality as much as possible (i.e. take into account actual efficiency values etc), and in these plans the site leader tries to adapt efficiency and resources to the current situation, with the goal of adhering to the production plan and the calculated costs.

There are many persons and subcontractors that come and go. There could be as many as 40 subcontractors at one site. To keep track of them and manage them is also a very complex task. The subcontractors also have other responsibilities, and cannot adapt freely to a new situation on one specific site.

In this context, the site leader is under heavy stress, and systematic planning for the future is often replaced by intuition since there are so many things happening around the site leader, that there is no room for sitting down to plan. Also, there is hardly any time to record what is happening, even though all parties agree that follow-up and evaluation would save much time (and money) in the future. We think that a computer based tool could

- help the site leader to keep track of activities and events during the construction,
- help the site leader to continuously support planning by having knowledge of the building (both the design and the current state), and by dynamically change properties of methods and activities, reflecting the way the site leader plans,
- be an advanced calculator for determining various areas, volumes etc, which gives the site leader much better calculating possibilities than before,
- give an accurate description of the design, through the possibility of getting corrections from the architect by electronic mail, or other computer based communications, and when corrections arrive, help the site leader find out the consequences,
- in general, give a better possibility of finding out consequences of disturbances, efficiency differences etc,
- give the possibility to reuse other site leaders' experiences,
• give the possibility to book, query and report to other databases, such as machinery database, economical reports, subcontractor's databases etc.

This paper describes a planning system for planning activities on the construction site, and requires the current state, the design, a method database and descriptions of the activities being part of the methods. The system acts as an advanced calculator, i.e. it calculates the efficiency, resources and time limits (durations). The system could be used anywhere in the construction process.

3. System Overview

The system consists in principle of two subsystems; the choice-of-method phase, and the scheduling phase. They are sequentially applied, i.e. methods are applied first to determine how (part of) the building should be produced in terms of activities, and then the scheduling phase determines when different activities should be performed. But that does not rule out that a method phase could be applied after a scheduling phase, to determine in more detail how some still complex activity should be done.

The starting point of the method choosing phase is the task that we want to do, which must be specified in some way. In this domain, the best specification is the design together with the written descriptions of the building. Some method is applied to the task of constructing (part of) the building, which leads to a collection of activities that, when executed, realize the building. Each activity consists of three parts: (i) preconditions, which must be satisfied in order for the activity to be executable, (ii) a duration description, which determines time- and resource consumption, and (iii) postconditions which specify the effects on the global state that the activity has.

The choice-of-method phase is where the building is decomposed into manageable parts and objects, by decomposing the building into smaller parts, and by decomposing the building into functionally different objects, which are produced separately by different resources (personnel, machinery etc, more about that later). It is in the choice-of-method phase that the production strategy is determined.

The decomposition is done through two hierarchies: the functional hierarchy, and the building hierarchy. The functional hierarchy can best be described by the BSAB table 2, while the building hierarchy determines how the building's different parts are put together to form larger pieces, in terms of their three-dimensional coordinates or some description of location.

In [Metod&Data] descriptions of methods for the building industry are given, where a method is a description of what has to be done in order to produce some part of the building, together with needed resources, material, etc. The descriptions also give some information about what the preconditions are, and also includes some dependency graphs of the contained activities, which are further specified on other sheets. Those data sheets were the most influential source of information for us, but we have made the distinction between a method and an activity clear in that methods are merely a way of grouping different activities into useful 'packages'.

There are also other extensive descriptions, for example [BPN], which are two large loose-leaves with information about different building objects and their related unit costs and unit working time. These have also been a source of information for us.

The execution of an activity could take shorter or longer time, depending on the resources allocated to the activity. If a lot of resources are allocated to the activity, the duration of the activity will be shorter, and the other way around, if just a small number of resources are allocated to the activity. By tuning the allocation of resources to activities differently, the resulting plans can vary a lot.
Resources can be of different kinds. We distinguish between materials, which are built into the object, and resources that are means to produce the object. We have looked at personnel as a resource, but resources can also be machinery of different kinds, or a subcontractor. To be able to reason about personnel, we have introduced three variables that can be set for each activity: \textit{Teamsize}, which is the size of a team for an activity, \textit{density}, which is the maximum number of teams working on a specific object or area, and \textit{number-of-teams}, which denotes the number of teams allocated to an activity. The relation between these variables is roughly \(1 \leq \text{number-of-teams} \leq \text{density}\). The personnel allocated to an activity is determined by multiplying \textit{teamsize} by \textit{number-of-teams}. By giving values to these variables, the planner can allocate resources to his activities in a multitude of ways, implementing different kinds of allocation strategies.

There is also a maximum number of personnel at the site, which at any moment during the construction cannot be exceeded, which has to be checked by the scheduler. This value is held in the global variable \textit{max-resources}.

3.1 Representing the Building

The building and its different parts (also referred to as \textit{objects}) are represented by two different structures: one which describes the \textit{location} of the object, and one which describes the \textit{function} and quality of the object.

The most exact representation of an object would be its three-dimensional coordinates describing its location, plus its structure (length, width, solidness, material etc). This is a very complex representation, which is hard to use efficiently. Also, a complete detailed 3D representation would be too detailed for a planning program, unless there is another system that handles neighbouring objects and other such properties, which is the kind of 3D information the planning system needs (we would like to have such a system). Instead, a simplified version based on the way the drawings are numbered is used, together with a grid on the 2D design. The location is described by a composite number, where the first part of the composite number denotes a first division into several, independent houses. The second subnumber denotes the floor of the house, and the third subnumber is simply a unique label for the object. The grid is used to determine where an object is located in the 2D design, by 'gridding' the design and then determining where in the grid each object is located. An object could be in one or several grid areas. The grid should be the same on each floor, i.e. the building is divided into boxes, from the basement to the roof. There is nothing that says that the grid should be constructed of squares, any kind of areas could be used.

The function of each part of the house is described by the standard in table 2 of the BSAB system. Part of it, the basic table for houses, is presented below.
<table>
<thead>
<tr>
<th>30 Composite house parts</th>
<th>30.0</th>
<th>30.1</th>
<th>30.2</th>
<th>30.3</th>
<th>30.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 (reserved)</td>
<td>31.0</td>
<td>31.1</td>
<td>31.2</td>
<td>31.3</td>
<td>31.4</td>
</tr>
<tr>
<td>32 House substructure</td>
<td>32.0 composite</td>
<td>32.1 vacant</td>
<td>32.2 excavation, ballast</td>
<td>32.3 pilework etc</td>
<td>32.4 basement</td>
</tr>
<tr>
<td>33 House skeleton</td>
<td>33.0 composite</td>
<td>33.1 skeleton walls</td>
<td>33.2 skeleton pillars</td>
<td>33.3 vacant</td>
<td>33.4 skeleton slabs</td>
</tr>
<tr>
<td>34 Roof</td>
<td>34.0 composite</td>
<td>34.1 compl. skeleton</td>
<td>34.2 inner climate shield</td>
<td>34.3 outer climate shield</td>
<td>34.4 roof end</td>
</tr>
<tr>
<td>35 Outer walls</td>
<td>35.0 composite</td>
<td>35.1 inner climate shield</td>
<td>35.2 vacant</td>
<td>35.3 outer climate shield</td>
<td>35.4 vacant</td>
</tr>
<tr>
<td>36 Room formation</td>
<td>36.0 composite</td>
<td>36.1 reserved</td>
<td>36.2 below flooring</td>
<td>36.3 inner walls</td>
<td>36.4 ceiling</td>
</tr>
<tr>
<td>Skeleton compl.</td>
<td>37.0 composite</td>
<td>37.1 vacant</td>
<td>37.2 stair surface etc.</td>
<td>37.3 wall surfaces</td>
<td>37.4 ceiling surface</td>
</tr>
<tr>
<td>37 Inner surface, room</td>
<td>38.0</td>
<td>38.1</td>
<td>38.2</td>
<td>38.3</td>
<td>38.4</td>
</tr>
<tr>
<td>compl.</td>
<td>(reserved)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>39 Other house parts</td>
<td>39.0</td>
<td>39.1</td>
<td>39.2</td>
<td>39.3</td>
<td>39.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>30.5</th>
<th>30.6</th>
<th>30.7</th>
<th>30.8</th>
<th>30.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>31.5</td>
<td>31.6</td>
<td>31.7</td>
<td>31.8</td>
<td>31.9</td>
</tr>
<tr>
<td>32.5 culverts etc</td>
<td>32.6 vacant</td>
<td>32.7 reserved</td>
<td>32.8 house compl.</td>
<td>32.9 other</td>
</tr>
<tr>
<td>33.5 vacant</td>
<td>33.6 stairs; elevator shaft</td>
<td>33.7 roof skeleton</td>
<td>33.8 house compl.</td>
<td>33.9 other</td>
</tr>
<tr>
<td>34.5 gap completions</td>
<td>34.6 vacant</td>
<td>34.7 roof terraces</td>
<td>34.8 house compl.</td>
<td>34.9 other</td>
</tr>
<tr>
<td>35.5 gap completions</td>
<td>35.6 vacant</td>
<td>35.7 vacant</td>
<td>35.8 house compl.</td>
<td>35.9 other</td>
</tr>
<tr>
<td>36.5 gap completions</td>
<td>36.6 inner stairs, compl.</td>
<td>36.7 vacant</td>
<td>36.8 house compl.</td>
<td>36.9 other</td>
</tr>
<tr>
<td>37.5 vacant</td>
<td>37.6 reserved</td>
<td>37.7 reserved</td>
<td>37.8 room compl.</td>
<td>37.9 other</td>
</tr>
<tr>
<td>38.5</td>
<td>38.6</td>
<td>38.7</td>
<td>38.8</td>
<td>38.9 other</td>
</tr>
<tr>
<td>39.5</td>
<td>39.6</td>
<td>39.7</td>
<td>39.8</td>
<td>39.9 other</td>
</tr>
</tbody>
</table>

**BSAB table 2**

For example, supporting walls are numbered 33.1 (the dot has no other function than giving better readability); the first number '3' denotes that it is a part of the house, the second number '3' denotes that it is a supporting part of the house, and the third number '1' denotes that it is a wall. Similarly, supporting slabs take the number 33.4.

The design is static, i.e. the system cannot change the design, and that is because changes to the design are the architect's job. The planning system should plan the construction of the specified building on the basis of available information, and not hypothesize about what could be added or changed, since then we are doing design. In real life, it is common that the architect makes changes during the construction of the building, or that the entrepreneur has to guess what will be in the detailed design, which sometimes leads to conflicts, and that the entrepreneur having to rebuild some part of the building.

### 3.2 The Choice-of-Method Phase

The goal of this phase is to choose how the building (or part of it) should be produced, to a relevant level of detail. To help us there is a collection of methods and/or section division possibilities (see [Lau91] for a description of a prototype presentation system for...
methods). In real life, one could section divide a complex object arbitrarily, but we have limited that possibility to just a few divisions. This limitation is due to the fact that we in this prototype system do not have an interactive graphical interface, since we want to focus on the fundamental problems of planning. Of course, if there were such an interactive interface, the user should be able to divide the object as he wishes to do. There is nothing in theory that makes it impossible to group different objects arbitrarily together. When, how, why, and which objects that an expert groups together, is another interesting question from a KBS view, but we do not present any ideas on that matter in this paper.

### 3.2.1 Methods

A method is a collection of activities that solves a specific problem. But what is a problem? And how do we describe it? One way would be to describe the problem in terms of the desired state that we want to reach. This has the advantage that it is easy to describe what the method achieves, but would not accord with our view that a method is a way to name small plans that in more detail describe what a complex task achieves. Another disadvantage is that it is hard to describe all state changes as an indexing argument of the method.

We have chosen to define a method as a name of a set of activities that together solve a higher level activity (or task). If we put it the other way around, the method indicates how to perform a complex activity (task) by replacing the complex activity by 'smaller' ones. This formulation makes it easy to model a hierarchy of activities with the methods as functions from one layer of activities to another layer, by defining methods to be functions from one activity to a set of activities.

Let $a_1$ be a superactivity, $a_{11}, \ldots, a_{1n}$ be subactivities and $m_1$ a method. The functional relationship is described by

$$m_1(a_1) = \{a_{11}, \ldots, a_{1n}\}$$

The function states that $m_1$ is a function mapping an activity $a_1$ to a set of activities $a_{11}$ to $a_{1n}$. It can be read as: If $a_1$ is a complex task that should be further detailed, $m_1$ can be applied to it and the result is $\{a_{11}, \ldots, a_{1n}\}$. The achieved state can be found by looking at the postconditions of the definition of the indexing activity $a_1$. The achieved state when the activities $a_{11}$ to $a_{1n}$ are executed is described by the postconditions of $a_{11}$ to $a_{1n}$, and these postcondition should be a more detailed description of what is achieved when executing $a_1$. The two descriptions of the achieved state (the postconditions of $a_1$ and the postconditions of $a_{11}$ to $a_{1n}$) could coincide, if the description of $a_1$ is complete with respect to $a_{11}$ to $a_{1n}$. But often the postconditions of $a_1$ are described on a higher level than those of $a_{11}$ to $a_{1n}$.

### 3.2.2 Section Divisions

To divide an object into smaller ones is much like applying a method to an object. When dividing an object (part of, or the whole building) we get a set of smaller objects, where the smaller objects are of the same type. For example, if a whole building is split into its flats, we call that a section division, since each flat is of the same type. But if a building is split into its supporting parts, facade etc, that is a method, since supporting parts and facade etc are produced in different ways.

Theoretically, a section division is a function mapping a larger object into a set of parts of the larger object, where each part is of the same type. The relationship is described by

$$d_1(a_1) = \{a_{11}, \ldots, a_{1n}\}$$
where $d_1$ is the name of the section division function, $a_1$ is the superactivity and $a_{11}$ to $a_{1n}$ are the subactivities. That $d_1$ is said to have a name does not rule out arbitrary divisions, since one can call a section division function anything (and, in fact, it could be nameless).

It could be confusing that we section divide an activity, but the same intuition as for methods applies to section divisions, i.e. a possible way to perform a complex task is to split it into smaller, manageable tasks.

### 3.3 The Scheduling Phase

When the choice-of-method phase is finished, there are a set of activities that, when executed, achieves our desired goal. In order to know when each activity can start to execute, the activities must be scheduled, i.e. each activity should have resources allocated to it which fix its duration, and the activity should be placed in time depending on the global state, which is affected by the other activities.

The activity database defines the properties of the activities used in the methods. The method presentation interface presented in [Lau91] handles activities as well, and could serve as an example of an interface to the editable activities. The idea is, as with the method database, that the user can build his own activity database, to match the method database.

#### 3.3.1 Activities

Activities are seen as processes, which have a start condition (called preconditions), duration description and a finish condition (called postconditions). When the preconditions are satisfied, including enough resources available, the activity starts. By consuming resources and material of different kinds during a period of time the activity accomplishes a change of the global state, which is reflected in the postconditions. The real world activity is modelled through an efficiency formula in the duration description, which reflects how long time and what resource consumption the activity would require when applied in the real world.

One way to represent an activity would be as a function from resources (and material, which we will take for granted that it exists) to a change in the global state:

$$a_1(R) = \text{change}(\text{PostCond}, \text{when}(\text{PreCond}) + \text{timeformula}(R))$$

where \text{when} is a function which returns the time $T$ when its argument conditions are satisfied, \text{timeformula} is a formula to calculate the duration based on the allocated resources $R$, and \text{change} changes the global state according to the state description $S$.

The above representation is a declarative one, based on functions. If we look at the simulation of a set of activities represented as above, the activities would have to suspend until their precondition is satisfied, then execute during some time and after that leave the global state changed in some way. This is a more procedural description, which could be represented as

$$a_1(R) = \text{PreCond} \rightarrow (\text{Execute} \rightarrow \text{PostCond})$$

where it is intuitively clearer that the activity $a_1$'s preconditions $\text{PreCond}$ must be satisfied in order to execute and change the global state. The duration is calculated when $\text{Execute}$ is executed, and the global state is affected by $\text{PostCond}$. We feel that the later representation more accurately coincides with our view of an activity as a process. We have chosen to have the time formula in the $\text{Execute}$ slot, and pass values from the preconditions $\text{PreCond}$ through $\text{Execute}$ to the postconditions $\text{PostCond}$ by logical variables. The use of the later representation can be compared with modus ponens in
logic calculus, i.e. if PreCond can be deduced, we can deduce (Execute → PostCond). Still, the latter representation could be viewed as a function, which returns a state change described by the postconditions.

Since methods and activities are both functions, methods are second order functions, i.e. applying a method to some activity gives a set of activities, which are also functions (see below).

3.3.2 Scheduling

The scheduling algorithm works in steps, where each step schedules one activity at a time, i.e. places it in time and calculates resource consumption. The algorithm takes one of the possible activities (i.e. whose preconditions are satisfied), calculates the duration of the activity based on efficiency coefficients and some measurement of the amount of work to be done given in the time formula of the activity, and stores the state change described in the postconditions of the activity in the global state database. If there is a choice of, for example, the amount of resources, one element of the possible ones is chosen through some defined function. The process is finished when all activities are scheduled.

Another way of looking at the scheduling algorithm is that when the choice-of-methods phase is finished, the set of all activities forms an equation system. The goal of the scheduling phase is to solve that equation system, if that is possible. If not, there is no solution meaning there is no plan, and the scheduling phase fails. The system notifies the user and backtracks into the method phase again. If there are several solutions, which is the common case, the system chooses one. And there are several possibilities to solve the equation system. For example, different resource allocation to the activities gives rise to different plans, and activities could be spread out in time in several different ways. The latter one can be done in infinitely many ways, but often there are just a few cases that are interesting. However, we have not defined what distinguishes the interesting cases from the uninteresting ones. This is expert knowledge, which we certainly would like to know more about. The conclusion section discusses this further.

The scheduler also has to check that the maximum resources available are not exceeded when it has allocated resources to an activity. The check is done by going through the already scheduled activities and their allocated resources, and comparing the total allocated resources with the maximum resources available.

The planning described here differs from 'ordinary' AI planning ([AC87, Geo88, GL87, GI89, Kae86, Kae88, Wil89, BD90, Ham89, GN87] are some papers giving a short survey), where the system tries to find a sequence of actions that leads to a desired state based on descriptions of different actions, which are performed in more (classical planning) or less systematic ways (reactive planning). The more systematic approaches often have something that guides the planning algorithm to its desired state in the form of a leads to operator [Kae88] or something similar, but seldom do these system have methods or canned plans based on experience. One exception are so called case-based planning systems [Ham89], but in such systems the planning system itself tries to deduce its experiences from failed plans, and not in cooperation with a user. In general, AI planning systems try to solve the problem of planning for an autonomous agent, which is something else than giving a user support for making plans, like our system.

The planning system PRS [GL87, GI89] has a form of canned plans, which has similarities with our approach using methods, but their activities are quite different, since the concept of resource consumption is lacking.

Our planning application also differs from commercially available project planning applications in that our planner does not group activities together by declaring them as coupled together (activity \(a_1\) demands \(a_0\) to be executed before \(a_1\)), but orders them
implicitly by use of preconditions and postconditions, which change the global state. Our
approach tries to model the state changes of the real world, influenced by the several
scheduling choices that exist, while commercial softwares we have seen (see e.g.
[ET89]) has simplified planning to a deterministic calculation of one possible plan, with
more or less possibilities of affecting the scheduling algorithm. Most of the commercial
software that is available is based on the Critical Path Method (CPM), which calculates
what activities are on the critical path and thus cannot be disturbed without disturbing
the total finish time. The other activities are often scheduled to start as soon as they can.

With the possibility to model the world, we are free to have any scheduling algorithm to
allocate resources and place activities in time. This paper describes a version which gives
CPM like results, but we have tried some other simple scheduling algorithms, like
allocating as few resources as possible, starting activities as late as possible, giving
priority to some activities (like 'try to get the house rain-proof as soon as possible') etc.,
all by further specifying control of what activities to choose and when, and what
resources to allocate. With more knowledge about what to prioritize in which situation,
the user can configure his own scheduling algorithm, without having a critical path if he
does not wish to.

3.4 Adaptation to the User's Knowledge and Experiences

The main idea is that the user specifies the planning knowledge himself through the
databases methods, section divisions and activities. The design databases (that comes
from the architect) and the global state are used by the planning system to simulate what
happens, and for the planning system to know what the actual global state is. A resource
database should also exist, where the site leader can keep track of the site's resources.
This database is in this application a part of the design database. Through a user interface
the user can change the content of the databases, and so change the behaviour of the
system into the behaviour that reflects his own planning preferences and knowledge.
Thereby the system does not reflect one single expert in the field, but each expert can
program the system to reflect his own planning knowledge.

An example of an interface to this planning system is given in [Lau91], and also in
[MDA].

4. Extensions of GCLA

As the development of this application proceeded, the need for some new constructs in
GCLA was detected (see [Kre92, Aro92] for an introduction of GCLA). There was a
need for having a construct that could collect every instance of a term that holds given a
definition, which is implemented as an index function. Index functions are connected
with quantifiers [Eri92], although they are not quite the same thing. While a universally
quantified variable stands for every possible instance, without demanding that the
instances could be determined, an index function is an arbitrarily function that generates
the possible instances of the specified term. While there are a lot of computational
problems with universally quantified variables, index functions are assumed to terminate
and produce a finite number of instances. If not, the system will go into a loop.

Also, we have added some new constraints to the system by implementing them in
Prolog and using them at the control level, and also introducing an if-then-else statement
at the control level.

4.1 Numbers and Arithmetic

Numbers are represented as the structure \( n(Num) \), which is a kind of typing, declaring
\( Num \) to be a number. \( n/1 \) is declared as a condition constructor, i.e. to be on the same
level as for example the arrow \( \rightarrow/2 \).
Arithmetic is implemented by new rules, which are loaded and incorporated into the GCLA system. These rules are part of a rule library, which the user could load to perform arithmetic. A typical example of an arithmetic rule is the rule for addition of two numbers,

\[
\text{add_left}(+(A,B), I, PT) \leq
\begin{align*}
(PT &\rightarrow (I\oplus[A|R] \setminus n(A1))), \\
(PT &\rightarrow (I\oplus[B|R] \setminus n(B1))), \\
X &\text{ is } A1 + B1, \\
(PT &\rightarrow (I\oplus[n(X)|R] \setminus C)) \\
&\rightarrow (I\oplus[(A + B)|R] \setminus C).
\end{align*}
\]

For a complete listing of the arithmetic rules, see appendix E.

4.2 Index Functions

An index function is a function mapping a GCLA query to a list or comma-separated vector of terms. Index functions are a control condition, i.e., they occur as constructs of the control language, and work in much the same way as Prolog's `findall` and `bagof`. An index function is represented as

\[(\text{ListOrVector} \ i \ \text{Query}) \rightarrow \text{Result}\]

where ListOrVector has the form \(\{T\}\) or \([T]\), specifying if the result should be a list or a vector and \(T\) is a term, called \textit{template term}. Result is the list or vector of all instances of \(T\) for which Query holds. Query could be any GCLA II query (i.e. \(\text{ProofTerm} \rightarrow \text{Sequent}\)), or a proviso atom. Variables in Query that do not occur in ListOrVector are the same as outside the \(\times / 2\) function, if they are not explicitly existentially quantified, in which case they are existentially bound in Query. Variables can be existentially quantified by the construct \(\forall v, C\), where \(v\) is a variable and \(C\) is a condition, denoting that \(v\) is existentially bound in \(C\) (\(v\) may be a term, in which case all variables in \(v\) are existentially quantified in \(C\)). If \(v\) is existentially bound outside the actual query (i.e. Query equals \(\forall v', \text{Query'}\)) then the control level handles all substitutions of variables in \(v\) with new variables, but if \(v\) is existentially bound inside an object level sequent, then the inference rules defined by the user must take care of the structure \(\forall v, C\), since it is then an ordinary condition. An inference rule for handling \(\times / 2\) to the right can be found in appendix F, with the name \text{sigma_right}.

If the function \(\times / 2\) is mapped into PID [Hal91], it would be written as

\[
(\text{List})\text{Query} \quad \text{or} \quad (\text{Vector})\text{Query}
\]
depending on whether the collection is in the form of a set or a vector.

An example of an index function query is

\[
\text{\{W\} i} (X^Y^Z^\text{axiom}(X,Y,Z) \rightarrow \{(f(1),f(2)) \setminus f(W))\}) \rightarrow \text{List}
\]

which binds \text{List} to \([1,2]\). If the variables \(X, Y\) and \(Z\) are not existentially bound,

\[
\text{\{W\} i} (\text{axiom}(X,Y,Z) \rightarrow \{(f(1),f(2)) \setminus f(W)]) \rightarrow \text{List}
\]

we get the answers \text{List} = [1], \(X = f(1)\), \(Y = f(1)\), \(Z = [1]\) and \text{List} = [2], \(X = f(2)\), \(Y = f(2)\), \(Z = [f(1)]\).

Index functions are used to implement 'domain terms', which span over collections of terms. In our case domain terms span over elements in the design database and elements
of the global state. Variables that should range over all possible values with respect to the
current definition and global state are denoted as a \( \text{Var} \). For example, it is necessary to
reason about whether every building part for a given story is built and finished. a \( \text{Var} \)-
constructs are related to parameterized variables [Eri92], although they are not the same.
More about rules and strategies handling domain terms in section 7.3.2.

4.3 Additional Constraints

It is possible to implement a constraint solving system for object level variables in the
control code by passing the constraints around in a context list argument, and checking
the constraint's satisfiability at some given points. This solution is not as effective as a
low-level implementation of constraints in Prolog itself, and since there is always the
possibility in GCLA to use the underlying Prolog system, we have used Prolog's
coroutining facilities for implementing a couple of constraints.

Seven new constraints are introduced:

\[
\text{max}(X, \text{List})
\]

where \( X \) should be the maximum value of the elements of \( \text{List} \). It is assumed that the
elements of \( \text{List} \) are numbers.

\[
\text{min}(X, \text{List})
\]

where \( X \) should be the minimum value of the elements of \( \text{List} \). It is assumed that the
elements of \( \text{List} \) are numbers.

\[X > Y, X < Y, X = Y, X := Y, X = Y\]

These are numeric constraints on the variables \( X \) and \( Y \), which have their obvious
meaning.

4.4 If-Statements

There is also a need for an if-then-else construct to be used in the control code. An if-
statement can be expressed in GCLA itself, as shown in [Aro91b], but since we need to
express something like \( \text{ProofTerm} \vdash \text{ObjectSequent} \), i.e. the nonprovability of a
sequent by using a specific proof term, and also a more efficient treatment of if-then-else
statements, a special construct with its own semantics and its own syntax has been
introduced:

\[
((\text{If} \leftarrow \text{Then}), \text{Else})
\]

It should occur as the result of a rule or strategy, i.e. it is a function, which should be
used to give a sequent as answer:

\[
((\text{If} \leftarrow \text{Then}), \text{Else}) \leftarrow (\text{Antecedent} \leftarrow \text{Consequent})
\]

The arrow '\( \leftarrow \)' is in principle the same arrow as '\( \rightarrow \)', but turned around. Typically, the
if-statement is used as

\[
((\text{rule1}(\ldots, (A \rightarrow C)) \leftarrow (\text{ProofTerm1} \rightarrow (A \rightarrow C)), \text{ProofTerm2})
\]

with the reading "if rule1 is applicable to the current sequent with \( (A \rightarrow C) \) as the result,
then continue with \( \text{ProofTerm1} \) proving \( (A \rightarrow C) \), else apply \( \text{ProofTerm2} \) to the current
sequent".
The if-then-else construct has the operational semantics

\[
\begin{align*}
\text{If } & \vdash \text{Seq} \quad \forall \text{Then} \quad \text{Else} \vdash \text{Seq} \\
& = \text{Seq} \\
= \text{Seq} \\
\text{If } & \not\vdash \text{Seq} \\
& = \text{Seq}
\end{align*}
\]

which means that any condition to the left can occur as the \text{If} condition, any condition to the right can occur as the \text{Then} condition, and any condition to the left can occur as a the \text{Else} condition.

Note that the proviso \(\not\vdash\) has no current interpretation in terms of the basic GCLA II primitives, and occurs only as a proviso of the operational semantics for our if-then-else construct. We are currently investigating its properties in a larger context [Kre93], and are quite optimistic about the possibility of giving it a clear and clean semantics. It is implemented in terms of the Sicstus Prolog primitive \(\text{if/3}\).

5. Different Programming Methodologies Used

In this application, some different programming methodologies are used, most of them described in [Aro92]. For example, functional programming, objects (processes) and relational programming, together with different kinds of control, are used. We give a short description of some different methodologies used, which are all explained in [Aro92] except for closures, which have not been described elsewhere.

There is a special library control file for evaluating common arithmetic, which is described in the appendix E. The rules and strategies in this file implements the common arithmetic functions together with rules and strategies for handling user defined functions and locally defined functions (closures).

5.1 Functions and Relations

Functions are implemented in GCLA as a kind of hypothetical reasoning (see [Aro91b, Aro92]). A typical function definition is the following, which implements the addition of integers in successor arithmetic:

\[
\begin{align*}
\text{add}(0,X) & \leftarrow X. \\
\text{add}(s(X),Y) & \leftarrow \text{succ}(\text{add}(X,Y)). \\
\text{add}(X,Y) & \leftarrow \text{pi} \backslash \{X \leftarrow s(_), \ X = 0\} \leftarrow \\
& \ \pi 2\backslash \{X \rightarrow Z \rightarrow \text{add}(Z,Y)). \\
\text{succ}(X) & \leftarrow \text{pi} \backslash \{X \rightarrow Y \rightarrow s(Y)).
\end{align*}
\]

Functions can be evaluated both lazily and eagerly, and both strategies are used in the application. The basic strategy for lazy evaluation is

\[
\text{arl} \leftarrow \text{axiom}_1, \ \text{right}(\text{arl}), \ \text{left}(\text{arl}).
\]

which first tries to return a value by the axiom rule. If that is not possible, some other rule is tried, to further evaluate the functional term. The basic eager evaluation strategy is

\[
\text{lra} \leftarrow \text{left}(\text{lra}), \ \text{right}(\text{lra}), \ \text{axiom}_1.
\]

which first tries to evaluate the function term, and then on backtracking uses the axiom rule to return the term as a value.

Three queries to the \text{add/2} function are:
For a more in-depth description of functional and relational execution the reader should consult [Aro91b].

5.2 Closures
Closures are used to define local functions (or relations). By assuming the local defined function, and then use the inference rules in a certain order, which is accomplished by some strategies in the control language, the locally defined function can be used in the execution.

An example where a locally defined function $f/I$ is used is the query below. $f/I$ implements the factorial function.

```
math \- defun((pi X \ pi Y \ pi Z \ 
  ( (f(n(0)) \ -> \ n(1)),
    (f(n(X)) \ -> \ (n(X) > n(0))
        \ -> \ n(X) * f(n(X - n(1))))),
    (f(X) \ -> \ (not(X=n(1)) \ -> \ f(n(X)))),
    X -> n(Y) \ -> \ f(n(Y))))
),
    f(n(3)); \- n(6).
```

$p$ is bound to 6. The term `defun/1` contains the locally defined function, which is handled by the ordinary rules. The reason for having the `defun/1`-construct is to guide contraction. When its contents is to be used, the `defun(X)`-condition is duplicated first, as seen in the control code below:

```
closure(I,PT) <= (I\[defun(_)\] | R \- _). % If there is a defun,
  closure(I,PT) <=
    contr\_l(_,I,
      defun\_left(defun(_,I,
        closure(I,PT))). % get copy
    % eliminate defun
    % apply closure strat

closure(I,PT) <=
  pi\_left(_,_,closure(I,PT)),
  a\_left(_,I,axiom(_\_C,I)),
  weak\_l(_,_,handle(PT)),
  and\_l(_,_,closure(I,PT)).
% local variable
% instantiate head
% handle body
% choose a condition

defun\_left(defun(X),I,PT) <=
  (PT \- (I\[X\] \- C))
  \- (I\[defun(X)\] \- C).
% get defuns arg free
% dual rule to comma
% to the right
```

- 15 -
contr_1/3 performs contraction on its first argument, and thus when defun_left/3 is performed in closure/2, there exists a copy of the locally defined function. By passing the index argument i around, we are sure that the term we are operating on is the same in the execution. The complete definition of the control code can be found in appendix E.

5.3 Objects and Processes
As described in [Aro92, Aro91c], terms in the antecedent can be regarded as objects. The object level definition gives the methods for objects (classes), and terms in the antecedent are seen as the objects (class instances). This interpretation of the antecedent’s terms is used in the scheduler, where each activity has its own methods (note that the term 'method' here is used in the terminology of an object oriented language).

We will use a very simple kind of objects; a kind of processes, which are suspended until some state is satisfied. When the state is satisfied, the process starts executing, and finishes with some change of the global state. A simplified version of the activity processes described in the scheduler is described by the small definition below:

```
process(Pre,Post) <= quote(Pre) -> quote(Post).
```

The control code to be used can be the arithmetic library described in appendix E.

A query where all the processes can run at once is

```
process(1,2), process(1,3), process(1,4), quote(1) \- quote(X)
```

where x is bound to 1, 2, 3, or 4, depending on which answer to choose.

A sequential process execution is

```
process(1,2), process(2,3), process(3,4), quote(1) \- quote(X).
```

where all the processes must run in order for the query to succeed. To see that they are all executable at the same time in the first query, we pose the query

```
[X] i ([process(1,2), process(1,3), process(1,4), quote(1)] \- quote(X)) -> z.
```

which binds z to \[1,2,1,2,3,1,2,3,4,1,2,4,1,2,3,4,1,3,1,2,3,1,2,3,4,1,3,4,1,2,3,4,1]\, z contains all the possible ways to unify quote(X) with some term quote(N) in the antecedent.

The above query is to be contrasted with the query

```
[X] i ([process(1,2), process(2,3), process(3,4), quote(1)] \- quote(X)) -> z
```

which binds z to \[1,2,1,2,3,1,2,3,4,1]\.

If we let a parentheses pair represent the antecedent’s quote-terms when the choice of a term of the antecedent in the axiom rule occurs, we get the following lists:

```
Z_1 = [(1), (2,1), (2,3,1), (2,3,4,1), (2,4,1), (2,3,4,1), (3,1), (2,3,1), (2,3,4,1), (3,4,1), (2,3,4,1), (4,1), (2,4,1), (2,3,4,1), (3,4,1), (2,3,4,1)]

Z_2 = [(1), (2,1), (2,3,1), (2,3,4,1)] ⊂ Z_1
```

- 16 -
To add to the process definition that some action *some_action* should be performed is easy. The definition

\[
\ldots \\
\text{process}(\text{name}_1) \leftarrow \\
\text{precondition} \rightarrow (\text{some_activity} \rightarrow \text{postcondition}). \\
\ldots
\]

makes *some_action* be performed when *precondition* is satisfied. *postcondition* is returned when *some_action* has been successfully executed.

6. Representing Plans, Designs and Global States

To plan is to know what goal, or result, to achieve, without knowing how to achieve that goal; we have to specify the process that should 'do it'. This means that we have an *intention*, regarding what we want to achieve, which is something else than actually having reached the goal. In our case the intention is the design that represents the actual building. The represented global state should reflect and coincide with the simulated actual state of the real world, i.e. if the plan were executed in the real world, with the actions taking place in real life, the represented global state would reflect the real state. But since we are *reasoning* about the plan and its activities, the represented global state holds a model of the real state, i.e. what we expect to be the real state.

Since the design is static and given by the architect, it can be reasoned about but not changed. Therefore the design is placed in the definition. The planned state is instead placed in the antecedent of sequents, where it can be dynamically changed and reasoned about.

6.1 Plans

The set of plans that come from the choice-of-method phase are implicitly hierarchically ordered, since every plan originates from some higher level activity being solved by the plan. Plans are represented as

\[
\text{plan}(\text{Name}, \text{SuperAct}, [\text{SubAct}_1, \ldots, \text{SubAct}_n]).
\]

where *SuperAct* is the activity that is solved by *SubAct* to *SubAct_. Name is the name of the method that was chosen. Each subactivity could be either an activity, or a new plan. Activities are represented as

\[
\text{activity}(\text{IdNumber}, \text{BSAB}_2, \text{Position}, \text{BSAB}_1, \\
\text{StartTime}, \text{EndTime}, \text{Means})
\]

where *IdNumber* is a unique ID number based on an informal standard used by SIAB, one of the main entrepreneurs in Sweden, who have in turn based that standard on the BSAB standard. BSAB1 and BSAB2 are the two BSAB standards, Position is a description of the location of the object involved in the activity, and StartTime and EndTime give the starting time and finish time of the activity. The two time arguments are used to pass the time values to other activities. The last argument, Means, holds either the resources allocated to the activity, if the activity is a leaf in the activity tree, or a list of the subactivities' start- and end time values, if the activity is a superactivity.

When an activity has been executed, a trace is left in the antecedent, which holds data about when and how the activity was performed. These terms have the representation

\[
\text{performed}(\text{Act}, \text{StartTime}, \text{Duration}, \text{Means}).
\]
where $\text{Act}$ is the activity represented as above, $\text{StartTime}$ as above, $\text{Duration}$ is the duration of the activity and $\text{Means}$ is as above for activities.

6.2 Design

The design database is a relational database that simulates a product model of the building. In order to be a product model the database should be a hierarchy of the different parts that on each level form compound elements. The current relational database is not able to do that. The planned NICK standard (National CAD exchange format) should be such a product model, but at the time of the implementation of this system, the standard was not yet provided. However, the current database is expressive enough to show what the planning system could do, and we hope to be able to use NICK at a later stage.

The relation $\text{design/4}$ holds all parts and their location in the building. The current design database is flat, and each clause holds something on the level of a wall or a slab. Inside each such clause, there is a property list, holding the properties of the part, together with data about subparts.

\[
\text{design(\text{BSAB2, Position, BSAB1, PropertyList}).}
\]

where $\text{BSAB2}$ is the BSAB table 2, described as a three elements list implements the functional hierarchy of the BSAB table 2 by letting the first number of the list denote the first number of the BSAB table 2, the second element denote the second number of the BSAB table 2, and so on. If the element is the structure $a(\text{Var})$ it denotes all the objects described by all possible substitutions for $a(\text{Var})$. $\text{Position}$ is a simple location specification together with a unique number, also as a three element list. This is not a hierarchy but a cross product, where the first element describes which sub-building the object is in, the second gives the story and the third is the object's unique identifier. $\text{Position}$ has the same structure as the $\text{Position}$ argument of an activity. $\text{BSAB1}$ is the BSAB table 1 and $\text{PropertyList}$ holds the properties of the part, typically length, height, thickness, volume, area, iron-percentage etc. An example of a concrete wall is

\[
\text{design([3,3,1], [1,1,0001], [e,3,2,1],}
\]

\[
\text{[grid([a11]), weight_per_sqrmeter(15), single_layer,}
\]

\[
\text{length(24), height(3), thickness(0.20), volume(14.4)])}.\]

where the wall is in the grid $\text{a11}$, is located in the first building on the first floor with the unique identifier $\text{0001}$, with a single layer of reinforcement which weighs 15 kg per square meter.

The $\text{BSAB2}$ list is a quite useful hierarchy when we introduce methods, while the $\text{Position}$ argument could be some more accurate 3D description of where the object is located. The presented three element list gives the possibility to implement simple section divisions, while we would have a more fine-grained description in a real implementation of this planning system, with a graphic interface consisting of grouping possibilities as well as splitting possibilities of designed objects. This could for example be implemented as a separate layer in a CAD drawing, together with the same grouping possibilities present in most drawing programs, e.g. MacDraw.

6.3 The Global State

The global state database is held in the antecedent, as 'silent' terms, i.e. no rule is applicable to them, except the axiom rule (and the structure rules contraction and weakening). By this encoding, the global state database is passive, but can be dynamically changed by state changing functions, and the global state can be queried through the axiom rule.
The global state is given by the user when the scheduler is entered, by giving the global state from which the scheduler should start to schedule the activities.

We have implemented the global state database as terms \texttt{state/3} holding the global state information as:

\begin{verbatim}
state(Object, StartTime, EndTime)
\end{verbatim}

where \texttt{Object} is the object together with its location representation (if the three dimensional information is relevant, otherwise it is left out), \texttt{StartTime} is the time when the object started to exist and \texttt{EndTime} is the time when the object ceased to exist (at least at the position described in \texttt{Object}). \texttt{Object} can also be other kind of information (abstract information), for which there is a duration.

In some sense the global state database gives the object-time function for every object the activities affect, in intervals, which can be arbitrarily large or small. For example, if a crane is used by some activity from time stamp \( t_1 \) until \( t_2 \), then moved to another location and used by another activity from \( t_3 \) to \( t_4 \), the following global state holds that information:

\begin{verbatim}
state((crane, location(a)), t1, t2),
state((crane, location(b)), t3, t4).
\end{verbatim}

A third activity can be used to move it from the first location to the second during \( t_2 \) to \( t_3 \), in which case we have to represent in the global state that the crane is moving:

\begin{verbatim}
state((crane, location(a)), t1, t2),
state((crane, moving(location(a),location(b)), t2, t3),
state((crane, location(b)), t3, t4).
\end{verbatim}

To be really accurate, we should also give the path that the crane is moving along, but that is too detailed for the current application, since we have not reached the point where we have crane schemas and we are not planning the motion of cranes. We have used the planning system to plan the construction of concrete walls, where the same mould is reused several times, and moved between the different locations, which is similar to the above example with the crane, but we will not develop these ideas concerning moving objects further in this paper, but concentrate on building objects that are location stable and not moving. In a later project we will elaborate the ideas about moving objects.

Note that the global state is the planned global state that the system has created. If the actual state is represented in the system, by typing in information or by other input facilities (cameras, external instruments of various kinds etc), there are much better possibilities to check the planned and the actual state on the site, and much better possibilities to incrementally plan using the latest available information.

7. The Choice-of-Method Phase

The choice-of-method phase determines the way the building is going to be produced. Methods are the entities in which activities are grouped together, as described in section 3. With expert knowledge about which methods are relevant to apply in which situation, a good planner can in the choice-of-method phase produce a set of activities, which can be scheduled rather freely, but at the same time give the plan a good structure. This expert knowledge should be implemented at the control level, and is at the moment not represented in the system. The system freely chooses one of the possible methods, without any knowledge about what method to prefer if there are several possible choices. (There is also the possibility of presenting the choice to the user, who has to make the choice.) This is further discussed in section 10.
7.1 What is a Method

The method database contains various ways activities could be grouped together to solve complex tasks. Each method has a name (and number) and an activity, the superactivity, as arguments, and the body is a representation of a plan, with the name of the plan, its superactivity and the subactivities. The reason why the representation of the plan contains the superactivity is that the superactivity must know which subactivities it is dependent on.

The generic method has the following definition:

\[
\text{m(\text{Number, SuperAct, Name})} \leftarrow \\
\text{plan(\text{Name, SuperAct, [SubAct}_1, \ldots, \text{SubAct}_n]).}
\]

and an example of a method is the plan for erecting a whole building, by its main functional objects:

\[
\text{m(1,{},activity(3, [3, a(V2), a(V3)], [V4, V5, V6], _, Tst, Tsl, Res))} \leftarrow \\
\text{plan(main_groups_of_building,} \\
\text{ activity(3, [3, a(V2), a(V3)], [V4, V5, V6], _, Tst, Tsl, Res),} \\
\text{ activity(2, [3, 2, a(V3)], [V4, 0, V6], _, _, _),} \quad \% \text{basis} \\
\text{ activity(33, [3, 3, a(V3)], [V4, V5, V6], [e1]_, _, _),} \quad \% \text{supporting parts} \\
\text{ activity('4_8', [3, 4, a(V3)], [V4, V5, V6], _, _),} \quad \% \text{outer roof} \\
\text{ activity(6, [3, 5, a(V3)], [V4, V5, V6], _, _),} \quad \% \text{outer walls} \\
\text{ activity(7, [3, 6, a(V3)], [V4, V5, V6], _, _),} \quad \% \text{rooming in} \\
\text{ activity(66, [3, 7, a(V3)], [V4, V5, V6], _, _),} \quad \% \text{inner surfaces}
\]

Note the use of the \( a(\text{Var}) \)-construct in the head of the clause. It imposes the restriction that the activity should not have been expanded before, i.e. the second and third argument of the \text{BSAB2} argument should not be an integer. The expansion of the second argument of the \text{BSAB2} argument is done by the presented method.

7.2 What is a Section Division

Section divisions differ from methods in that all the subactivities are of the same type, but refer to different objects, as explained in section 3. The basis for section divisions is the design, and therefore the function for dividing a building object into smaller parts of the same kind is in general dependent on the building. Typically, a section division function could be to divide into stories, divide all walls on a story into other smaller sets of walls etc, but also arbitrary groupings of building objects, which can be different from building to building. We have studied some general divisions, such as dividing into sub-buildings, floors and homogeneous objects (walls, slabs, etc on basis of how they are drawn).

An example of a section division is
where (i) generates all floors, (ii) sorts them, (iii) generates one activity for each floor, and (iv) returns the new plan. Also note the use of the \( a(V5) \)-constructor in the third argument of the head. We demand that the floor division should not have been made before by the \( a(V5) \)-term.

7.3 The Choice-of-Method Algorithm

The choice-of-method algorithm is implemented as a lazy functional program, which expands an activity one step, and then returns the new plan as the result. Since laziness is implemented through backtracking, and the choice of a method is also implemented through backtracking, it means that backtracking is used in two ways: to consider new methods for the same activity, and to further specify the plan by expanding a new activity.

7.3.1 Definition

The object level definition is a functional program, which in principle has three atomic definitions: \texttt{plan/3}, \texttt{expand/2} and \texttt{expand_one/2}. \texttt{member/2} is an ordinary member predicate, with the difference that it is deterministic.

\[
\text{plan}(N,A,X) \leftarrow \text{expand}(X,Y) \rightarrow \text{pl}(N,A,Y).
\]

\[
\text{expand}([],[]).
\]

\[
\text{expand}([X|Y],[X|X1|Y1]) \leftarrow \text{expand}(X,X1),\text{expand}(Y,Y1).
\]

\[
\text{expand}\left((\text{plan}(N,A,X),Z)\right) \leftarrow \text{plan}(N,A,X) \rightarrow Z.
\]

\[
\text{expand}(X,X1) \leftarrow [N \leftarrow [],X \leftarrow [__|__],X \leftarrow \text{plan}(_-,_-)) \leftarrow \text{expand_one}(_-X) \rightarrow X1.
\]

\[
\text{expand_one}(1,X) \leftarrow X.
\]

\[
\text{expand_one}(2,X) \leftarrow m(_-,-X).
\]

\[
\text{expand_one}(N,X) \leftarrow [N \leftarrow 1;N \leftarrow 2] \leftarrow e(_-,-X).
\]

\[
\text{member}(X,[X|X]).
\]

\[
\text{member}(X,[X1|X2]) \leftarrow \text{f}(X,X1) \leftarrow \text{f}(W,X2) \leftarrow \text{member}(X,R).
\]

\[
\text{plan}(Name,HigherActivity,Plan) \text{ is a function, which calls expand/2 to elaborate on the plan Plan, and returns the new plan through the data structure pl(Name,HigherActivity,NewPlan). expand(Old,New) traverses Old in order to further specify some activity. The fourth clause of expand is applicable when such an activity is found, and expand_one/2 is called.}
\]

\[
\text{expand_one} \text{ first returns the same activity again as the answer. When backtracking occurs, a method is tried first, and then a section division is tried. This order could be changed by changing the control code to behave differently.}
\]
7.3.2 Control

The control is separated into three different parts: one general part, one part dealing with methods, and one part dealing with section divisions. The section division part is a bit more complex since it determines how many parts the object should be split into by looking at the design.

A generalized form of the derivation tree for a method-choosing query is presented below. Note that the generation of the different plans is done in the leftmost subproof by ordering the choice of the terms in the set \( \{ \text{Act}_1, \ m(... \text{Act}_1), \ e(... \text{Act}_1) \} \). In our case we have chosen the order 1) \( \text{Act}_1 \), 2) \( m(... \text{Act}_1) \), 3) \( e(... \text{Act}_1) \).

- \( \text{One of } \{ \text{Act}_1, m(... \text{Act}_1), e(... \text{Act}_1) \}^{**} \)
- \( \text{expand_one}(_, \text{Act}_1) \leftarrow \text{PLAN}_1^{*} \)
- \( \leftarrow \text{expand}(... \text{Act}_1... \text{PLAN}_1...) \leftarrow \text{pl}(..., \text{PLAN}_1...) \leftarrow \text{pl}(..., \text{Plan}) \)

- \( \text{plan}(... \text{Act}_1...) \leftarrow \text{pl}(..., \text{Plan}) \)

*The double line represents a number of applications of the \text{expand/2} definition, to get to the case where \( \text{Act}_1 \) is handled. The '...' represent other pairs of activities-plans, which are handled analogously as \( \text{Act}_1 \) and \( \text{PLAN} \) are, after the double lines. In the implementation they are realized as lists.

**'If \( \text{Act}_1 \) is chosen, no expansion is made, i.e. \( \text{PLAN} \) is bound to \( \text{Act}_1 \). If \( m(... \text{Act}_1) \) is chosen, \( \text{expand_one}(...) \) is replaced by a new plan \( \text{plan}(...) \), and another derivation of the same kind is performed. If \( e(..., \text{Act}_1) \) is chosen, \( \text{expand_one}(...) \) is replaced by a new plan \( \text{plan}(...) \), and another derivation of the same kind is performed.

\text{plan/3} is a function, which returns a plan, which is represented by the term \( \text{pl}/3 \).

\text{plan/0} is the top most strategy of the choice-of-method phase. There are no special rules to handle the choice-of-method algorithm, unless one wants another behaviour of the \text{expand_one/2} definition, in which case the \text{d_left1/3} rule can be extended with a generation proviso, guiding which of the three \text{expand_one}-clauses that should be applicable first. Otherwise there is nothing special about the control code. It is the same kind of control code used at several places in [Aro92].

\begin{verbatim}
plan <= axiom1(_, _, _), right1(plan), left1(plan).
right1(PT) <=
  sort_right,
  findall_right(PT),
  bagof_right(PT),
  number_right,
  v_right(_, PT, PT),
  a_right(_, PT),
  0_right(_, PT),
  true_right,
  d_right1(_, PT).
\end{verbatim}
left1(PT) <=
  false_left(0),
  v_left(_,_,PT),
  a_left2(_,_,PT,PT,_,),
  o_left(_,_,PT,PT),
  d_left1(_,_,PT),
  pi_left(_,_,PT).

d_right1(C,PT) <=
  (not(functor(C,plan,_)),
   not(functor(C,pl,_))
   -> (_ \ C)).

_ d_right1(C,PT) <= d_right(C,PT).

_ d_left1(C,I,PT) <=
  (not(functor(C,pl,_)),
   not(functor(C,activity,_)),
   not(functor(C,state,_))
   -> (I[\ C] \ _)).

_ d_left1(T,X,PT) <=
  atom(T),
  definiens(T,Dp),
  DP \= false,
  (PT -> (X\[Dp|Y] \ C))
  -> (X\[T|Y] \ C).

axiom1(T,C,I) <=
  data(T)
  -> (I[\ T] \ C).

axiom1(T,C,I) <=
  axiom(T,C,I).

a_left2(A -> Cl1),I,PT,PT1,left) <=
  data(Cl1) -> (I[\ (A -> Cl1)] \ _).

a_left2(A -> Cl1),I,PT,PT1, left) <=
  a_left1(X,I,PT,PT1).

a_left2(A -> Cl1),I,PT,PT1, right) <=
  not(data(Cl1)) -> (I[\ (A -> Cl1)] \ _).

a_left2(A -> Cl1),I,PT,PT1, right) <=
  a_left(X,I,PT,PT1).

a_left1(A -> Cl1),I,PT,PT1) <=
  (PT1 -> (I\[Cl1|Y] \ C)),
  (PT -> (I\[Y \ A])),
  -> (I\[A -> Cl1]|Y) \ C).

%% Proviso
data(X) :- functor(X,pl,_).
data(X) :- functor(X,activity,_).

Note how the strategy a_left2/5 distinguishes between two versions of the arrow-left rule, a_left/4 and a_left1/4. a_left/4 first proves the antecedent of the arrow condition, while a_left1/4 first proves the sequent where the conclusion of the arrow condition is added to the antecedent. a_left2/5 is used to unify the result of a function with the partially instantiated answer given by the user, and is used when the user has given some information about which kind of plans he has demanded.

Section divisions need some new rules to be written, to handle sorting, bags etc. Bags are implemented by index functions, and are used to find all instances of a template using the design, for example all floors of a building or all walls on a floor. The new conditions findall/3 and bagof/4, which in many ways resemble the corresponding ones in Prolog, are used at the object level to implement bags.
To be sure that domain terms (i.e., terms containing the special construct \( a(Var) \)) are handled correctly, all cases of bagof/4 and findall/3 replace every occurrence of \( a(Var) \) with \( Var \), and existentially quantify \( Var \) inside the index function call, before the bag is generated.

\[
\text{sort_right} <= \\
\text{sort}(X,Y) -> \\
(- \_ \text{\textless sort}(X,Y)). \\
\text{number_right} <= \\
\text{number}(C) -> \\
(- \_ \text{\textless number}(n(C))). \\
\text{findall_right}(PT) <= \\
\text{lift_from_a}(B,B1,[],_), \\
([A] \text{i PT As}^*B1^*(PT->(Ass \_ B1)) -> C) \\
-> (Ass \_ \text{\textless findall}(A,B,C)). \\
\text{bagof_right}(PT) <= \\
\text{lift_from_a}(B,B1,[],Vars), \\
\text{append}(EVars,Vars,Vars1), \\
([A] \text{i Vars1^*(PT->(Ass \_ - B1)) -> C}) \\
-> (Ass \_ \text{\textless bagof}(EVars,A,B,C)).
\]

\[
\text{lift_from_a}(V,V,L,L) \leftarrow \text{var}(V). \\
\text{lift_from_a}(A,V,L,[V|L]) \leftarrow \text{functor}(A,a,1), A = a(V). \\
\text{lift_from_a}(A,Atom,L,L) \leftarrow \text{atomic}(Atom). \\
\text{lift_from_a}(X,[F|R],L,L2) \leftarrow \\
\text{nonvar}(X), X = [F|R], \\
\text{lift_from_a}(F,F,L,L1), \\
\text{lift_from_a}(R,R1,L1,L2). \\
\text{lift_from_a}(Str,Str1,L,L1) \leftarrow \text{nonvar}(Str), \\
Str1 =..\text{[S|A]}, S \text{\textless= .'}, S \text{\textless= a,A \textless= []}, \\
\text{lift_from_a}(A,A1,L,L1), \\
\text{Str1} =..\text{[S|A]}.
\]

The \text{bagof_right}/1 rule, the \text{findall_right}/1 rule and the \text{lift_from_a}/4 proviso are also used in the scheduling phase.

### 7.4 Complexity in the Choice-of-method Phase

A word on the complexity of the choice-of-method phase is justified. The search space is enormous. The choice-of-method phase builds a tree consisting of activities as leaves, and two types of nodes: method expansion nodes and section division nodes. If we compare that tree with an ideal case, where a tree is built from leaves and from two different kinds of nodes, and where the branching factor for each node is 3 (i.e. there are three subtrees to every expanded node), then for a tree of depth 2 there are 37 different possible trees, ranging from one single leaf to a maximum expanded tree. If the depth is set to 4, we get 140797364928697 different possible trees! Below is a table for some other results, where the branching factor is in the left column, and the depth is in the first row:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>7</td>
<td>15</td>
<td>31</td>
<td>63</td>
<td>127</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>19</td>
<td>723</td>
<td>1045459</td>
<td>2.19e+12</td>
<td>9.56e+24</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>55</td>
<td>332751</td>
<td>7.37e+16</td>
<td>8.00e+50</td>
<td>1.6e+153</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>163</td>
<td>1.412e+9</td>
<td>7.95e+36</td>
<td>8.0e+147</td>
<td>inf</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>487</td>
<td>5.48e+13</td>
<td>9.8e+68</td>
<td>inf</td>
<td>inf</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>1459</td>
<td>1.93e+19</td>
<td>1.0e+116</td>
<td>inf</td>
<td>inf</td>
</tr>
</tbody>
</table>
inf is a number greater than 1.797693e+308. The value for branching factor 6 and depth 6 is about 3.823e+4179. So, there is a need for heuristics about when and where to apply which method and/or section division! The results presented above are a bit larger than in the choice-of-method case, since at some nodes the expansion is restricted, e.g. there cannot be a section division into stories for garden work, but the figures give some feeling for the search space and the complexity of generating plans.

To guide the use of methods and section divisions we have to extend the control code. One possible extension is to add a new clause to the d_left1/3 strategy,

\[
d_{\text{left1}}(C,I,PT) \leftarrow \text{allowed}(C,I,IR) \rightarrow (I \in [C \setminus IR] \leftarrow _{}).
\]

which restricts the d_{left1} strategy to be applicable to C whenever allowed/3 succeeds. It is now possible to cut away for example section divisions of a whole building into its stories, or vice versa, always section division a building into its stories before going into detail about how each story is going to be produced, etc.

Another solution to the problem of guiding the search is to always ask the user whenever there is a choice. This can be done in several ways, and we will not present any particular way here, but refer to section 9.6.4 for an example of how user interaction can be accomplished.

7.5 Preparing for the Scheduling Phase

When the choice-of-method phase is finished, there is an intermediate step where the hierarchical tree of activities and plans is flattened. This step puts all the activities on the same level, and they will all show up in the antecedent when the scheduling phase is to begin.

The definition consists of a function, flatten_act/2, which, apart from flattening the tree-structure and removing the pl/3-structure, also adds a delayed functional call akt(Number, Activity, Name), around the activity, and through unification builds up the list of the start times and finish times for the superactivities. The latter is performed through the second argument of the function flatten_act/2. One could say that a plan L is replaced by the starting times and finishing times of each activity of L, forming the list List. This list is used in the scheduling algorithm to determine the superactivities' start- and finish times.

\[
\text{flatten_act}([pl(N, \text{activity}(A,B,C,D,E,F,Res),L)], [(E,F)]) \leftarrow \\
\text{cons}(\text{akt}(-1, \text{activity}(A,B,C,D,E,F,Res), \text{times}(E,F,List))), \\
\text{flatten_act}(L,List)).
\]

\[
\text{flatten_act}([pl(N, \text{activity}(A,B,C,D,E,G,Re),L)], [(E,G) | R1]) \leftarrow \\
\text{cons}(\text{akt}(-1, \text{activity}(A,B,C,D,E,G,Re), \text{times}(E,G,List))), \\
\text{append}((\text{flatten_act}(L,List), \text{flatten_act}([F|R],R1))).
\]

\[
\text{flatten_act}([\text{activity}(A,B,C,D,E,G,Res),F|R], [(E,G) | R1]) \leftarrow \\
\text{cons}(\text{akt}(A, \text{activity}(A,B,C,D,E,G,Res),N), \text{flatten_act}([F|R],R1)).
\]

\[
\text{flatten_act}([\text{activity}(A,B,C,D,E,G,Res)], [(E,G)]) \leftarrow \\
\text{cons}(\text{akt}(A, \text{activity}(A,B,C,D,E,G,Res),N), []).
\]

append([], L) \leftarrow L.

append([F|R], L) \leftarrow \text{cons}(F, \text{append}(R,L)).

append(X,Y) \#(X \notin [1], X \notin [[]]) \leftarrow \\
((X \rightarrow Z), (Z = [ ] ; Z = [[]]) \rightarrow \text{append}(Z,Y)).

All superactivities get the number -1, to distinguish them from activities occurring as leafs in the tree.
The control definition for executing this phase is a changed version of the traditional arl strategy. There are also some simple restrictions on the general rules d_right/2, d_left/3 and axiom/3. The top level strategy is flatten/0.

flatten <=
    axiom_flatten(_,_,_),
    right_flatten(flatten),
    left_flatten(flatten).

axiom_flatten(T,C,I) <=
    (functor(T,akt,3) ;
     functor(T,[1],0) ;
     functor(T,'.',2))
    -> (I@[T] \- C).
axiom_flatten(T,C,I) <= axiom(T,C,I).

d_right_flatten(C,PT) <= functor(C,\-,2) -> (_,\- C).
d_right_flatten(C,PT) <= d_right(C,PT).

right_flatten(PT) <=
    true_right,
    d_right_flatten(_,PT),
    o_right(_,_,PT),
    a_right(_,PT),
    v_right(_,PT,PT).

left_flatten(PT) <=
    d_left_flatten(C,I,PT),
    a_left(_,_,PT,PT).

d_left_flatten(T,I,PT) <=
    (functor(T,flatten_act,2) ;
     functor(T,append,2) ;
     functor(T,cons,2))
    -> (I@[T] \- C).
d_left_flatten(A,B,PT) <= d_left(A,B,PT).

8. The Scheduling Phase

When the activities have been generated by the choice-of-method phase, and the tree-structure has been flattened out, the activities should be scheduled. This is done through a simulation of the planned activities, by treating all the activities as delayed processes, which start when their preconditions are satisfied. But the simulation is not actually something that goes forward one timeunit at each step. One could better regard it as an equation solver, which finds time values and resource values for the variables, by incrementally taking one activity at a time and placing it in time by determining its resources and starting time.

Most of the computation time is spent in the scheduling phase, since most of the calculations are performed in this phase, although there is not more knowledge used here than in the choice-of-method phase. As mentioned before, a good planner makes good choices of methods, which simplifies the scheduling phase.

What makes the scheduling phase complex is a lot of calculations together with the ability to change just one parameter, which formally gives a new solution. This new solution could be hard for the human eye to find, since just one parameter is changed. Human planners tend to make more drastic changes if they have to backtrack and reschedule the activities. We would like to have a better understanding of when two solutions should be regarded as different, and we hope to investigate this issue later, as discussed in section 10.
8.1 What is an Activity

An activity is regarded as a process, which consumes resources and has some duration, from the time it starts to the time it stops. To model the duration, a time formula is used, which calculates the duration. Typically, this formula is based on an efficiency coefficient together with a formula to calculate the amount of what should be done. To this time formula, a coefficient can be added to compensate for specific calculated better or poorer performance (typically season, which personnel, complicated building or complicated surroundings etc). Also, the team size is given, and how many teams can work in parallel on the same part or area.

Each activity in the activity database has the form

\[
\text{akt(}\text{Number}, \text{Activity}, \text{Name}) \leftarrow \\
(\text{PreConditions} \rightarrow (\text{Calculations} \rightarrow \text{PostConditions}))
\]

where \text{Number} is the same unique number that can be found in \text{Activity}, \text{Activity} has the form explained in section 6.1, and \text{Name} is the name of the activity. \text{PreConditions} holds the preconditions of the activity, and \text{PostConditions} holds the postconditions. \text{Calculations} has often the following form (but it is not limited to it):

\[
(\text{Initial_funcalls}, \\
(\text{Local_hypothesis} \rightarrow (\text{Resource_allocation}, \text{Time_calculation})))
\]

where \text{Local_hypothesis} are used to give names to different formulas. This gives the code better readability, since each formula has a mnemonic name that tells us what it stands for. One could substitute away all the named locally defined hypothesis into \text{Resource_allocation} and \text{Time_calculation} (which the author also did once), but the readability is then much worse then.

An example of an activity is the following, which defines the activity for erecting the whole building including work on the surroundings.

\[
\text{akt(0,A,project)} \leftarrow \\
(A = \text{activity}(0, [a(V1), a(V2), a(V3)], [V4, V5, V6], \\
B, Time, End, NT), \\
\text{when}(\text{started}(\text{official_start}, __, T)) \rightarrow \\
((\text{area}(\text{design}([3, 3, 4], [a(_), a(_), a(_)])) \rightarrow n(TotArea)), \\
(\text{area}(\text{design}([3, 3, 4], [a(_), 0, a(_)])) \rightarrow n(BasArea))), \\
(\text{defun}((\text{time_formula} \rightarrow n(9) * n(TotArea))), \\
\text{defun}(\text{building_area} \rightarrow n(Area))), \\
\text{defun}(\text{team_size(general_workers} \rightarrow n(2))), \\
\text{defun}(\text{place_coeff} \rightarrow n(1.0))), \\
\text{defun}(\text{density(general_workers} \rightarrow (n(BasArea) / n(25)))) \\
\rightarrow (\text{get_manpower(team_size(general_workers), \\
\text{density(general_workers), NT}) \\
\rightarrow n(Res)), \\
(\text{consume}(T, n(Res), \text{time_formula}, End, A) \rightarrow \text{quote(Act)})) \\
\rightarrow (\text{change}(\text{started}([a(V1), a(V2), a(V3)], [V4, V5, V6], B, End, \\
\text{Act}))).
\]

where \text{when}/2 is the precondition, \text{area}/1 is the initial function call and \text{get_manpower}/2 is the resource allocation function call. \text{consume}/5 calculates the duration and sets the start time based on \text{T}, and returns the representation \text{performed}/4 of the performed activity, which will be held in the antecedent as a trace of the executed activity. \text{change}/2 states what should be changed in the global state. \text{when}/2, \text{consume}/5, \text{area}/2, \text{get_manpower}/2 and \text{change}/2 are defined by the planning system.
The example description above of the activity database shows how all activities that are not superactivities can be coded. To handle superactivities, the following definition is used:

\[
\text{akt}(-1, \text{Act}, [F|R]) \Leftarrow \\
\quad \text{true} \\
\quad \rightarrow (\text{find max min}([F|R], \text{Min}, \text{Max}) \\
\quad \rightarrow \text{performed}(\text{Act}, \text{Min}, \text{Max}, [F|R])). \\
\]

\[
\text{find max min}(L, \text{Tst}, \text{Tsl}) \Leftarrow \\
\quad \text{find max min1}(L, [], [], \text{Tst}, \text{Tsl}). \\
\]

\[
\text{find max min1}([], \text{Min}, \text{Max}, n(\text{Tst}), n(\text{Tsl})) \Leftarrow \\
\quad \text{constr}((\text{Tst} = \text{min}(\text{Min}))), \\
\quad \text{constr}((\text{Tsl} = \text{max}(\text{Max}))). \\
\]

\[
\text{find max min1}([n(\text{Tst}), n(\text{Tsl})] [R], \text{Min}, \text{Max}, \text{ST}, \text{SL}) \Leftarrow \\
\quad \text{find max min1}(R, [\text{Tst} | \text{Min}], [\text{Tsl} | \text{Max}], \text{ST}, \text{SL}). \\
\]

\text{akt}(-1, \text{Act}, [F|R]) \text{ is applicable when the third argument is a list, which is the sign of a superactivity. The starting time for the superactivity is the value of the subactivity that starts first of all activities in the plan of the superactivity, i.e. the minimum of the start times of the list [F|R]. The finish time is calculated analogously, but the maximum value is chosen instead. true is the precondition for this activity, since every activity must have a precondition.}

Since we are using the constraints \text{max} and \text{min} to implement \text{find max min1}, and the precondition of superactivities is always true, these activities can be scheduled first, and the values of \text{Max} and \text{Min} will be correct when all the subactivities have been scheduled.

8.2 Resources

Resources can be of different kinds, e.g. personnel, different kinds of machinery, etc. We have looked at personnel, and just in a simple way. There is a maximum number of personnel stored in the case database, i.e. together with the design. For each activity which demands personnel there are some different functions for grouping and scheduling, as showed in the former section.

By giving values to the two locally defined functions \text{teamsize} and \text{density}, and by defining control code for the allocating function \text{get_manpower/2}, the user can tune his allocation of personnel to generate possible allocations in a certain order. By use of the locally defined functions, better and more fine-grained allocation algorithms can be defined than those described here, it is all up to the user. Also, when talking about other resources, machinery for example, the possibility to have completely different sets of locally defined functions for different categories of machinery exists.

8.2.1 Team Size

The team size is determined by a local function in the activity, or could be just a number. In our case the team size will always be a local function in the activity description. In the example of the previous section, it was the local function

\[
\text{defun}((\text{teamsize}(\text{general_workers}) \rightarrow n(2))).
\]

8.2.2 Density

The maximum number of teams working on the same part or area, which we will call the \text{density}, is determined by a local function described in the activity, or could be a fixed number. All our activities have area functions to determine the maximum density. In the example of the previous section the local function was
defun ((density(general_workers) -> (building_area / n(100)))).

8.2.3 Number of Teams
The number of teams could vary between 1 and the maximum density. The scheduler
should make an intelligent guess which is the best choice, from a single team up to the
maximum number of teams. It is here that the experienced human planner could tune a
plan to be good or bad. We have only a very simple heuristic, namely try to allocate as
much resources as possible. Again, the option of having expert knowledge of this would
be good, and is discussed further in section 10.

The used function is defined by the planning system, but the user could either redefine it,
or put it as a whole as a local function. The system defined function is called
get_manpower/2, and has the following definition:

```
get_manpower(TeamSize, TeamFormula) <=
  max_resources(man, _, MaxSize),
  (int(MaxSize / TeamSize) -> n(MaxSize1)),
  (int(TeamFormula) -> n(TeamNum)),
  (min([TeamNum, MaxSize1]) -> n(Top)) ->
  TeamSize * gen(n(1), n(Top)).
```

```
gen(A, B) <= ((int(B) -> n(B1)) -> genl(A, n(B1), _)).
genl(n(_), n(X), 1) <= n(X).
genl(n(X), n(X), Z)#{Z \= 1} <=
  n(Y) < n(X) -> genl(n(Y), n(X) - n(1), _).
genl(X, Y, _)#{Y \= n(_)} <=
  (Y -> Z) -> genl(X, Z, _).
```

where max_resource/3 refers to the maximum resources available, given by the case
database (i.e. the same database as the design). gen/2 generates one number between A
and B, and is used to allocate the number of teams to the current activity.

8.2.4 Overload
After having determined the number of teams, the duration of the activity can be
calculated, and the activity could be placed in time, which is done in consume/5 when
called from the activity function akt/3 Before the scheduler can schedule the next
activity, the scheduler must check that the manpower is not overloaded by the newly
scheduled activity, i.e. that the scheduled resources are not greater than the maximum
resources available. One way of doing this is to check that for each day the number of
allocated men does not violate the site's resource maximum, which is done by the
function overload_check/4 below.

```
consume (n(Time), Resources, TimeFormula, n(End), Act)
  #{Resources \= n(0)) <=
  ((int((TimeFormula / Resources) / n(B)) -> n(Duration)),
   call(constr(End = Time + Duration)) ->
   overload_check(Act, n(Time), n(Duration), Resources)).
```

- 29 -
overload_check(Act,Xst,Dur,Res) <=
(Xst + Dur -> Xst1),
 findall(N, (gen(Xst,Xst1) -> N), L), % Find all days Xst - Xst1
 forall1(N, (member(N,L)) % Check no overload during
 -> findall(R, % those days
 (performed(_,Xst1,Xst11,R),
 Xst1 <= N, N <= Xst11),
 L1),
 (sum(L1) + Res -> ResTmp), % sum all resources
 max_resources(man,N,ResMax),
 ResTmp =< ResMax)) % check maximum
 -> quote(performed(Act,Xst,Xst1,Res)).

By noting that the only places where the number of men allocated to work changes are
where activities start or end, a more efficient version of the algorithm can be defined. It is
this more efficient version that is used in the implementation presented in appendix D.
The number n(8) in consume/5 comes from the transformation of hours into days.

8.3 The Scheduling Algorithm

In principle, the scheduling algorithm is GCLA itself. The activities are added as
hypotheses in the antecedent, and the GCLA system chooses one that can be executed.
The choice of which activity should be executed is done through the control code, so
there are actually no definitions for the scheduling algorithm. But there are some help
functions that handle preconditions and postconditions, which are described here.

8.3.1 Definitions for Handling Preconditions

Preconditions can be of two kinds. First, there can be a precondition demanding that
some global state must contain a specific object at some specific position before the
starting time of the current activity, and secondly, there can be a precondition demanding
that some object must have been removed some time before the starting time of the
current activity. The first one is called started, and the second one is called finished.
In the first case, the global state database state/3 is checked for the existence of named
objects at time Time. In the second case, the global state database is checked for the non-
existence of the named objects at time Time, i.e. that the third argument of the state/3
terms is less than the starting time of the current activity. The latter kind is not used
further in this paper, since we are only considering plans that operate on static building
objects (walls, slabs etc), but is interesting as soon as we are talking about moving
objects (c.f. section 6.3 and section 10).

when((A,B),Time) <=
 when(A,Time1),
 when(B,Time2),
 constr({Time = max([Time1,Time2]))}.
when(started(B1,B2,B3),n(Time)) <=
 forall([B1,B2,B3,Prop], (design(B1,B2,B3,Prop) ->
 state(bsab(B1,B2,B3),_,_)),
 findall(Start, state(bsab(B1,B2,B3),n(Start),_,List),
 constr({Time = max(List)})),
 findall(End, state(bsab(B1,B2,B3),_,n(End)),List1),
 constr({Time = min(List1)}).
when(finished(B1,B2,B3),n(Time)) <=
 forall([B1,B2,B3,Prop], (design(B1,B2,B3,Prop) ->
 state(bsab(B1,B2,B3),_,n(End)))),
 findall(E, state(bsab(B1,B2,B3),_,n(E)),List),
 (max(List) -> n(Time)).
8.3.2 Definition for Handling Postconditions

As for preconditions, postconditions could be of the same two kinds: started and finished. started means that the starting time (the second argument) of the global state term state/3 is instantiated to Time, and finished means that the finishing time (the third argument) of the state term state/3 is instantiated to Time.

\[
\text{change} \left( \text{started}(B1, B2, B3), n(Time) \right) \leftarrow \\
\quad \text{forall}([B1, B2, B3, \text{Prop}], \\
\quad \quad \text{design}(B1, B2, B3, \text{Prop}) \rightarrow \\
\quad \quad \text{state}(\text{bsab}(B1, B2, B3), n(Time), n(End))) .
\]

\[
\text{change} \left( \text{finished}(B1, B2, B3), n(Time) \right) \leftarrow \\
\quad \text{forall}([B1, B2, B3, \text{Prop}], \\
\quad \quad \text{design}(B1, B2, B3, \text{Prop}), \\
\quad \quad \text{state}(\text{bsab}(B1, B2, B3), n(Start), n(Time)) \rightarrow \text{true}) .
\]

\[
\text{change} \left( (A, B), \text{Time} \right) \leftarrow \text{change}(A, \text{Time}), \text{change}(B, \text{Time}) .
\]

8.3.3 Control

The control code consists in principle only of strategies, and is split into three main parts (apart from the mathematical rules and strategies that are loaded from the GCLA control library): handling preconditions (i.e. queries about the global state), handling calculations and handling the postcondition (i.e. state changes).

The derivation tree for a scheduling step is presented below. Note how the subproofs to the left are used to evaluate the preconditions and calculations, while the rightmost proof is the continuation of the execution, where the postconditions change the global state.

### Further execution of activities in Acts

<table>
<thead>
<tr>
<th>Proof of PreCond</th>
<th>Proof of Calc</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \text{State}[\text{PostCond}</td>
<td>\text{Acts}] \leftarrow \text{plot} )</td>
</tr>
<tr>
<td></td>
<td>( \text{State}[\text{Calc} \rightarrow \text{PostCond}]</td>
<td>\text{Acts}] \leftarrow \text{plot} )</td>
</tr>
<tr>
<td></td>
<td>( \text{State}[\text{Calc} \rightarrow \text{PostCond})]</td>
<td>\text{Acts} \leftarrow \text{plot} )</td>
</tr>
<tr>
<td></td>
<td>( \text{State}[\text{akte}(</td>
<td>\text{Activity})</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

The top level strategy is schedule, whose definition is

\[
\text{schedule} \leftarrow \\
\quad \text{(ready\_sch} <\text{true),} \quad \text{% All activities scheduled?} \\
\quad \text{((search(akte(_,), I),} \\
\quad \quad \text{d\_left(akte(_,), I),} \\
\quad \quad \text{a\_left(_,),} \\
\quad \quad \quad \text{weak\_all(akte(_,),} \\
\quad \quad \quad \quad \text{weak\_all(perform(_,), schedule\_pre))}, \\
\quad \quad \quad \quad \quad \text{a\_left(_,),} \\
\quad \quad \quad \quad \quad \text{weak\_all(state(_,),} \\
\quad \quad \quad \quad \quad \quad \text{weak\_all(akte(_,), schedule\_calc))}, \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \text{v\_left\_all(schedule\_post((A \rightarrow C))))))} \leftarrow \text{(schedule} \rightarrow \text{(A \rightarrow C))}, \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \text{false}) .
\]
The execution is finished if ready_sch holds, where we have chosen to define ready_sch to hold when the antecedent consists of only the global state database together with the trace from the executed activities (the performed database).

The functional expression beginning with search/3 searches for an activity akt(_,_,_) that is applicable (i.e. schedule_pre succeeds), then executes that activity's different calculations by the strategy schedule_calc, and then changes the state by schedule_post. The calls to weak_all/2 remove the unnecessary terms for that particular part of the execution. Note the use of '=<-' as a sequential operator, where the execution continues with the rest of the activities in A that are not yet scheduled, when schedule_post has successfully finished its execution. By this sequentialization search/3 does not continue to search through the antecedent for other possible activities to choose instead.

8.3.3.1 Checking Preconditions
There are a number of different preconditions that can occur. The strategy for handling preconditions, called schedule_pre/0 is based on the general library strategy right/1, with a few changes. Since schedule_pre/0 is not that specialized, the execution is not fixed to just a few different kinds of preconditions.

```
schedule_pre <=
  (a_right(_,math) <- true),
  true_right,
  d_right(_,schedule_pre),
  v_right(_,schedule_pre,schedule_pre),
  forall r(_,schedule forall),
  findall_right(schedule_findall),
  constr_right(_).
```

The only very special treatment of the ordinary construction symbols is that when an arrow is encountered, which we have restricted to be a function call, and therefore the math strategy is applied.

8.3.3.2 Calculating Durations and Other Activity Data
The calculation strategies are also based on some of the library strategies. schedule_calc/0 is based on the library strategy ral/0, with the extension that the execution continues with the math/0 strategy if the current object level sequent can be treated as a function call. The switch to the math/0 strategy is implemented through the strategy mathing/0.

right_sch/1 and left_sch/1 are both extensions of the library strategies right/1 and left/1.

The d_left/3 rule is restricted not to be applicable to performed- and state-terms, in the d_left_sch/3 strategy.

```
schedule_calc <=
  (mathing <- true),
  (right_sch(schedule_calc),
   axiom(_,_,_),
   left_sch(schedule_calc)).
```
right_sch(PT) <= (not (functor(C,state,\_)) -> (\_ \(\rightarrow\) C)).
right_sch(PT) <=
                  relations,
                  findall_right(schedule_findall),
                  bagof_right(PT),
                  forall_r(C,scheduleforall),
                  constr_right(C),
                  v_right(_,PT,PT),
                  a_right(_,v_left_all(PT)),
                  o_right(_,PT),
                  true_right,
                  d_right_sch(_,PT),
                  keysort_right(_).
left_sch(PT) <=
                  forall_l_generate_all(C,I,PT,r,2),
                  v_left(_,PT),
                  a_left(_,PT,v_left_all(PT)),
                  o_left(_,PT,PT),
                  d_left_sch(_,PT),
                  pi_left(_,PT).

d_left_sch(C,I,PT) <=
                  (not (functor(C,performed,4)) -> (\@[C\_] \(\rightarrow\) \_)).

d_left_sch(C,I,PT) <=
                  (not (functor(C,state,3)) -> (\@[C\_] \(\rightarrow\) \_)).

d_left_sch(C,I,PT) <=
                  d_left(C,I,PT).

d_right_sch(C,PT) <= not (functor(C,plot,\_) -> (\_ \(\rightarrow\) C).

d_right_sch(C,PT) <= d_right(C,PT).

8.3.3.3 Changing the State (Postconditions)
The global state is changed by either adding new state terms, or by instantiating some argument of a state-term (typically the argument specifying the end-time, see section 6.3), or by a combination of both. Several state/3-terms can be changed according to a single postcondition. The control strategy is quite simple here, too:

schedule_post(PT) <=
                  (d_left(change(_,\_),\_,(A \(\rightarrow\) C))
                  <- (schedule_post(PT) -> (A \(\rightarrow\) C)),
                  (forall_l_generate_all(_,\_,(A \(\rightarrow\) C),ar,2)
                  <- (schedule_post(PT) -> (A \(\rightarrow\) C))),
                  PT.

The rule forall_l_generate_all is in principle a restricted form of forall elimination. All possible terms G1 for which A1 holds are generated. It is used to generate the global state database, and typically A1 is a building part in the design database and G1 is a state term describing that the specified part is being built.

forall_l_generate_all(foreall(X,(A \(\rightarrow\) G),I,PT,Petbag,2) <=
                  lift_from_s((A \(\rightarrow\) G),(A1 \(\rightarrow\) G1),[],Vars),
                  (G1) \^Vars \(\rightarrow\) (\@[\_G] \(\rightarrow\) \_)
                  -> Goals),
                  (PT -> (\@Gals\_R \(\rightarrow\) C))
                  -> (\@forall(X,(A \(\rightarrow\) G))\[R] \(\rightarrow\) C).

9. Example
Below is a small example to demonstrate the ideas and how the different parts fit together. The example is kept small for the sake of presentation simplicity.
9.1 The Example Building
The example building is a small two-floor building with concrete walls and slabs, where only the supporting walls and slabs are present. A simplified design of both the floors is presented below.

The walls and slabs are of reinforced concrete, and the facade is made of bricks. We have not taken into account windows, doors etc, but just supporting parts and the facade. This is due to a lack of implemented methods and activities presented in this paper, and not to any known inability to handle such objects. The complete listing of the design database is shown in appendix A.

9.2 Methods
The method database is a small one, just comprising some basic methods for producing supporting parts in concrete. Of course, a larger database should be used in a real application of this kind, based on the experienced site leader's knowledge. We present the methods here in a generalized way, and refer to appendix B for a complete listing of the methods used.

```
method(0, activity(0, whole_building, [V4, V5, V6])) <=
  plan(main_groups,
    activity(0, whole_building, [V4, V5, V6]),
    [activity(3, building, [V4, V5, V6]),
     activity(81, 'HVAC', [V4, V5, V6]),
     activity(83, 'Electricity', [V4, V5, V6]),
     activity(1, ground, [V4, V5, V6])).
method(1, activity(3, building, [V4, V5, V6])) <=
  plan(main_groups_building,
    activity(3, building, [V4, V5, V6]),
    [activity(2, basement_ground, [V4, 0, V6]),
     activity(33, supporting, [V4, V5, V6]),
     activity('4 8', outer_roof, [V4, V5, V6]),
     activity(6, outer_walls, [V4, V5, V6]),
     activity(7, supplements&rooming, [V4, V5, V6]),
     activity(66, inner_surfaces, [V4, V5, V6])).
method(2, activity(33, supporting, [V4, 0, V6])) <=
  plan(supporting_parts,
    activity(33, supporting, [V4, 0, V6]),
    [activity('3_2', slab, [V4, 0, V6])].
```
method(3, [], activity(33, supporting, [V4, V5, a(V6)]))

#(V5\=0, V5\=a(_)) <=
plan(supporting_parts,
  activity(33, supporting, [V4, V5, a(_)]),
  [activity('3_3', supporting_walls, [V4, V5, a(V6)]),
   activity('3_4', supporting_pillars, [V4, V5, a(V6)]),
   activity('3_6', supporting_slabs, [V4, V5, a(V6)]),
   activity('3_7', stairs, [V4, V5, a(V6)]),
   activity('4_8', supporting_parts_roof, [V4, V5, a(V6)]))

As mentioned before, the section division possibilities is very limited, and we use only one very rough division in the example:

section_division(divide_into_floors,
  activity(X, building, [V4, a(V5), a(V6)]))

#(V2\=2, V2\=4, V2\=6)
<=
bagof(Floor, design(building, [V4, Floor, a(V6)]), Tmp),
bagof(activity(X, building, [V4, V5, a(V6)]),
member(V5, Tmp),
List)
-> plan(divide_into_floors,
  activity(X, building, [V4, a(V5), a(V6)]), List).

The basic search among possible plans are: first no expansion at all (the activity is returned unchanged), then try all possible methods, which in the examples are only three which are mutually exclusive, and last try a section division.

9.3 Activities

The activities in the activity database are the ones used in the method database. Since there are 13 activities in our example database, we only give an example of an activity, which is a generalized version of the most general activity; to build the whole building in one activity. It is used to get an overall estimation of needed resources and time limits. The complete activity database is listed in appendix D.

act(0, activity(0, whole_building, [V4, V5, V6], B, Time, End),
project) <=
when((started(OK_byAuthorities, _), T) ->
((area(designed([3, 3, 4], [a(_), a(_), a(_)])) -> TotArea),
(area(designed([3, 3, 4], [a(_), 0, a(_)])) -> BaseArea),
(defun((timeformula -> 9 * TotArea)),
defun((buildingarea -> Area)),
defun((teamsize(general_workers) -> 2)),
defun((number_of_teams(general_workers)
    -> (BaseArea / 25)) ->
  (get_manpower(teamsize(general_workers),
    number_of_teams(general_workers))
  -> Res),
(consume(T, Res, timeformula, End, A) -> Act))
-> (change((started(whole_building, [V4, V5, V6], B), End), Act)).

Our example resource allocation algorithm is simple; try to allocate as much resources as possible to an activity, based on the team size and the maximum number of teams possible. The starting time of the activity is also simplified; instead of starting some time after the preconditions are satisfied, the activity starts at the same time as the preconditions are satisfied. This is due to the complexity introduced when the relation '≤' is used instead of '=' in the constraint solving algorithms.
9.4 Calculation Examples

It is possible to use the system for calculating volumes and areas of the building. Just specify the parts that should be in the calculation, and the system collects and sums them together. It works in much the same way as a query language to a relational database, which is not surprising since GCLA is a logic language.

An example of such a query is

```
schedule_calc
  \<- area(design([3,3,4],[a(_),a(_),a(_)])) \<- n(X)

X = 864
```

which gives the area of all supporting slabs of the building. Another query is

```
schedule_calc \<- length(design([3,3,1],[a(_),1,a(_)])) \<- n(X)

X = 96
```

which gives the total length of all walls on the first floor.

We can also state indeterministic queries, such as the area for each slab,

```
schedule_calc \<- area(design([3,3,4],[1,Y,a(_)])) \<- n(X)

X = 288
Y = 0 ;
X = 288
Y = 1 ;
X = 288
Y = 2 ;

no
```

This is useful for every day calculations, and the combinations of different calculations are very large. And, as soon as there is a change in the design, that change is reflected at once in the calculations.

9.5 Queries About Teams, Times and Durations.

There are possibilities to query the activity database about durations, facts, settings etc. This is useful when finding out relationships between different activities, and to get to know single activities properties.

To ask the activity project about its teamsize, and also to get the type of the workers in the team, we pose the query

```
(\<- clause(akt(_,_,project),
          (A, pre -> (Init,(Defuns -> _calc)) -> _post))),
(schedule_calc \<- \<- A, Init,
              (Defuns -> (teamsize(X) -> n(N)))).

... N = 2,
X = general_workers
```
To get the total amount of work in manhours to build the supporting parts of the building, we pose the question

\[
\text{(\texttt{\textbackslash \textbackslash \textbackslash \textbackslash - clause(akt(33,activity(_,[3,3,a(_)],a(_),a(_)),
\textbackslash \textbackslash \textbackslash \textbackslash \textbackslash ,a(_),\_\_,\_\_),
\textbackslash \textbackslash \textbackslash \textbackslash \textbackslash (A, \texttt{pre} -> (\texttt{Init},(\texttt{Defuns} -> \_\_\_calc)) -> \_\_\_\_post))
\textbackslash \textbackslash \textbackslash \textbackslash \textbackslash ,
\textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \texttt{schedule\_calc} \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \_\_\_ \_\_ A, \texttt{Init}
\textbackslash \textbackslash \textbackslash \textbackslash \textbackslash ,
\textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \texttt{Defuns} -> (\texttt{timeformula} -> n(N))))].}
\]

\[\ldots\]

\[N = 720.0\]

To get the duration for constructing the supporting parts of floor 1 of house 1 when 4 workers are allocated to that activity, we pose the query

\[
\text{(\texttt{\textbackslash \textbackslash \textbackslash \textbackslash - clause(akt(33,activity(_,[3,3,a(_)],[1,1,a(_)],
\textbackslash \textbackslash \textbackslash \textbackslash ,a(_),\_\_,\_\_),
\textbackslash \textbackslash \textbackslash \textbackslash (A, \texttt{pre} -> (\texttt{Init},(\texttt{Defuns} -> \_\_\_calc)) -> \_\_\_\_post))
\textbackslash \textbackslash \textbackslash \textbackslash ,
\textbackslash \textbackslash \textbackslash \textbackslash \texttt{schedule\_calc} \textbackslash \textbackslash \textbackslash \textbackslash \_\_\_ \_\_ A, \texttt{Init}
\textbackslash \textbackslash \textbackslash \textbackslash ,
\textbackslash \textbackslash \textbackslash \textbackslash \texttt{Defuns} -> (\texttt{timeformula} / n(4) -> n(N))))].}
\]

\[\ldots\]

\[N = 180.0\]

9.6 Planning Examples

We will present the planning examples through some screen dumps of the graphical output produced by some routines implemented in the Sicstus GM package. The plans are Gantt-schemas, with the activities listed to the right and the duration and place in time plotted in the graph. Each vertical bar on the x-axis is one week, i.e. 5 working days. We also give the corresponding resource graphs, which plot the number of men working on the site as a function of the time. The basis for the resource graph calculations is the performed/4 database.

9.6.1 The Whole Building by One Activity

The first query is to build the building in one step. This query gives the user an approximate resource value to use later. The query is

\[
\text{\texttt{(plan \textbackslash \textbackslash \textbackslash \textbackslash \_\_\_ (plan\(\texttt{start},
\texttt{activity}(0,[a(_),a(_),a(_)],a(_),a(_),a(_)],\_\_,\_\_,T,T1),
\texttt{activity}(0,[a(_),a(_),a(_)],a(_),a(_),a(_)],\_\_,\_\_,T,T1))
\textbackslash \textbackslash \textbackslash \textbackslash \_\_\_ \texttt{pl(N,A,X))},
(\texttt{flatten \_\_\_ flatten\_\_act(X,_) \_\_ X1),
(\texttt{incorporate(_,[],schedule) \_\_\_ X1,stock(stock\_\_act,_,n(0),_), \_\_ plot).}}
\]

where each call \texttt{(ProofTerm \_\_\_ Sequent)} to the GCLA interpreter corresponds to one phase.

The first answer delivered by the system is where as much resources as possible are allocated to construct the building, in this case 22 workers. We have set the maximum number of workers at the site to 30, which means that the frame 22 is given by the number-of-team formula multiplied by the teamsize, given by the activity.
The second answer is where the number of teams has been decreased by one, by the \texttt{gen/2} function, i.e. 2 workers are withdrawn. As can be seen, the duration increases slightly.

The system decreases the number of workers by 2 down to just one team working on the site, which is the case below. When backtracking now occurs, the system backtracks into the choice-of-method phase again, which is presented in the next section.
9.6.2 The Whole Building Expanded by Two Methods

The first method to apply on a whole building is when the building is split into its main functional parts: The house itself (i.e. without installations), different kinds of installations, and work outside the house. Another possibility would have been to section divide the building into smaller parts.

The first answer when the method main groups is used is the following, where we have as many workers as possible on each activity. Note that the time scale is changed in the Gantt schema.

The next answer is where the number of men is decreased by one team on the last scheduled activity (i.e. the "work on ground" activity). This is a new plan for the system, although a human planner would perhaps not regard it as such.
There are about 1000 different plans, or rather, a plan can be constructed in about 1000
different ways by the scheduler. All these plans are different plans in the way the system
treats them, since they differ at some point, even if very little. Here there is room for a
better specification of when a plan is to be regarded as a new solution. We give just one
more example of a plan.

9.6.3 Section Divide the Building

The next thing to do is perhaps to section divide the actual building into its floors, which
is presented below. Note that it is only the building that is section divided. The query that
accomplishes the plan is

\[
X = \{pl(main\_groups,\_,[pl(floor\_partitioning,\_,\_)]\}],
\]

\[
(plan \ \|\|- \ (-
  (plan(start,
    activity(0,[a(\_),a(\_),a(\_)],[a(\_),a(\_),a(\_)],_T,T1),
    [activity(0,[a(\_),a(\_),a(\_)],[a(\_),a(\_),a(\_)],_T,T1)])
  )
  )
  (flatten \|\|- flatten\_act(X,\_) \|- X1),
  (incorporate(_,[]),schedule)\|-\-
  X1,state(bsab(official\_start,\_,\_),n(0),\_) \|- plot).
\]

where the only difference compared with the previous query is that we have specified
which plans we are interested in. The first plan and resource graph produced are
Note that the activity building is expanded by the three activities Building 0, Building 1, and Building 2. Building 0 is the foundation of the house, including the bottommost slab. The same generation as in section 9.6.2 starts if we backtrack, i.e. the activity Work on ground's resources is changed first, then the resources allocated to Electrical installation are changed etc.

9.6.4 Manually Setting Values
The number of plans generated by the system can be reduced by letting the user fill in values at different places in the execution, leading to a decreased search space. By adding user input, we also get a close cooperation between the user and the system, where the user can determine himself how much he wants to be in control of the system.

User input can be realized in several ways, by different kinds of input facilities (extended definitions, input windows etc), of which one of the possible extensions is presented below. The example shows how the user is able to set different parameters before the scheduling phase starts. By imposing values on some parameters, the search space radically decreases.

It is easy to add different input devices to the control code. Often we need only add a clause to some strategy. A clause containing the proviso present_partial_act/1 is added to d_left_flatten/3, which is used in the intermediate phase between the choice-of-method phase and the scheduling phase.

The original code
d_left_flatten(T,I,PT) <=
    (functor(T,flatten_act,2) ;
     functor(T,append,2) ;
     functor(T,cons,2))
    -> (I@[T|_] \- C).
d_left_flatten(A,B,PT) <= d_left(A,B,PT).

is changed into

d_left_flatten(cons(A,B),I,PT) <=
    present_partial_act(A) -> (I@[cons(A,B)|_] \- _).

d_left_flatten(T,I,PT) <=
    (functor(T,flatten_act,2) ;
     functor(T,append,2) ;
     functor(T,cons,2))
    -> (I@[T|_] \- C).
d_left_flatten(A,B,PT) <= d_left(A,B,PT).

where present_partial_act/1 generates the window below, to be filled in by the user. Note that the change of code is local to the intermediate step, and local to the control definition. The object level function cons/2 is not changed.

The input fields must contain a number, or the symbol auto, in which case the system keeps the uninstantiated variable, to be filled in later during the scheduling phase (as is the case without the user input added).

By taking the same query as in the previous section, but setting all the number-of-teams fields to 2, the following plan is generated:

Since all number-of-teams fields were filled in, the duration of the activities are fixed, and since we demand that an activity starts as soon as possible (by the definition of the
10. Discussion

We have described a system which spans the set of all possible plans given a design. The system has no expert knowledge about what plans are good and what plans are poor, and which method to apply in a given situation. But the system has expert knowledge about what different systematical ways there are to solve a complex task. The next step would be to add expert knowledge about what plans should be preferred over other plans, for example, if there are several methods to choose among, the system should be able to pick one which seems better based on the overall information. As for the described system, this additional information should be editable by the user, to reflect his personal knowledge and preferences. There are different ways to do that, but one promising way is to use decision theory, which is used to calculate a priority when there exist choices, for example resource allocation to competing activities. By this an ordering of the possible plans can be imposed, which in turn guides both the method choosing algorithm and the scheduling algorithm. Whether it is possible to find priority functions for preference of methods and resource allocation, and if the expressiveness of them is enough for experts in the field, is not clear, but to us it seems the most promising approach.

The basis of the planning theory described here is very basic. It should be possible to generalize it to other domains as well. The basic entities of our planning theory are activities which are applied to and change a global state when resources are allocated to it. A scheduler reasons about when and with what resources different activities are allocated, as soon as the activities are known. To generate the activities, methods are used to systematically group activities, that solve a more complex task. The overall goal is expressed as the design of a building, representing a state change. We hope to be able to apply these ideas to some other domain as well, to see what parts are general and what are parts specific for this domain.

There is one known logical deficiency that ought to be mentioned. The application described here lacks the possibility to add constraints on the global state. For example, assume that there is one activity $a'$ changing the state into a state $s'$, and then an activity $a''$ that changes the state before state $s'$ into a state $s''$, from which $s'$ cannot be reached. This means that $s'$ must contain the information that all states before $s'$ must satisfy some constraints that should not violate it.

\[ \begin{array}{c}
  a'' \\
  a' \\
\end{array} \]

\[ \begin{array}{cccc}
  s & s' & s'' & \text{time} \\
\end{array} \]

It is possible to order the activities for the scheduling phase by imposing conditions on the preconditions, that cause such critical activities as $a'$ to be performed after $a''$, when the state $s''$ is known. This is quite natural, since one often delays an activity $a'$ that is known to be dependent on another activity $a''$ until its duration and place in time are calculated.

To be able to manage the search space, the user must have ways to guide the generation of plans to obtain plans he is interested in. Some such guiding possibilities have been sketched in the paper, e.g. partially specifying what plans the choice-of-method phase should produce, interaction with the user to fill in values in the preparation phase between the choice-of-method phase and the scheduling phase, and one could also add user interaction in the scheduling phase when there are several ways to solve the produced