COPERNICUS PROJECT 0149
ENCODE

Task 1.2 Porting VITAL debugging and visualisation tools to PC

E N C O D E

Environment for Configuration Design

Document Reference:	TR-Encode-TUKE-4-95	Date: 12/95
Activity:	T1.2	Status: Internal Release
Distribution:	All ENCODE Partners	
Title:	**Configuration design in CLIPS - room allocation problem**	
Abstract:	Experimental implementation of the room allocation problem by the use of the Complete-Revise-Optimize strategy in order to test the applicability of the CLIPS system to solve configuration design problems.	
Author(s):	Marian Mach (TUKE)	

Collaborators (ENCODE Partners):
OU - THE OPEN UNIVERSITY (Coordinator), UK
CVUT - CZECH TECHNICAL UNIVERSITY, CZECH REPUBLIC
*TUKE - TECHNICAL UNIVERSITY OF KOSICE, SLOVAK REPUBLIC
DCCS - DCCS ENGINEERING, CZECH REPUBLIC
ASEPO - ASEPO, CZECH REPUBLIC
*marked partners are involved in this task

Configuration design in CLIPS – room allocation problem

TR-Encode-TUKE-4-95

Marian Mach
Dept. of Cybernetics and Artificial Intelligence
Technical University of Košice
Letna 9/B, 041 20 Košice
machm@ccsun.tuke.sk

December 29, 1995

1 Introduction

This report describes results of an attempt to prove the applicability of the *CLIPS* system to *configuration design* problems. The aim of this attempt was to verify the possibility of implementing a system dedicated to configuration problems in CLIPS. This was tested by the implementation of a particular problem (assignment of people to rooms). For the purpose of the CLIPS applicability test one particular method for the solution of configuration problems was implemented.

Since the aim was to prove only the applicability of CLIPS, this implementation does not claim either completeness or the optimality of the solution of the test problem.

2 Definition of configuration design

Configuration design can be viewed as the selection of *components* and the composition of these components into a final *artefact* in order to provide a required functionality of the constructed artefact. Various predefined sets of components with given characteristics are available. Only (some of) these components can participate in the final artefact design. The components can be interconnected in different ways, predefined in advance.

From the designer's point of view not all solutions providing the required functionality of the artefact are equal. The designer gives priority to the design which is the best from some viewpoint, i.e. it optimizes some function representing this viewpoint.

3 Configuration design problem structure

3.1 Parameters

The solution of a typical configuration problem is represented by a set of *parameters*, related to:

- components selected in order to participate in the artefact design;
- the number of selected components;
- component attributes;
- interconnections of selected components;
- properties of the final artefact.

For every type of the configuration problem a set of parameters is defined. But not each parameter must participate in the final solution — these parameters are only available and they can be used. For instance, one particular value of one parameter can cause another parameter to be useless for the solution (but it can be included in the solution) or its presence in the solution does not have sense. For this reason the current stage of the solution in general uses only a subset of all parameters. These used parameters are called *active* and the other parameters are *inactive*.

The activity attribute of a parameter can change in a dynamic way. During the search for the final solution some parameters can be activated and others can be transferred into the set of inactive parameters. A special case is when the set of active parameters does not change during the design process (and all parameters which are active at the beginning of the design process are included in each intermediate and final solution).

The question is which parameters are active at the start of the search for the solution when no parameter has a value. There are several strategies:

- at least one parameter has to be active in order to start the design process;
- all parameters are active and after the initialization some of them are inactivated;
- only those parameters are active which are always (or very often) used in the early stages of the design process.

The last case is typical for the situation when designer for the sake of simplicity first deals only with a restricted set of basic parameters (which characterise the design) and additional parameters (which specify the design in detail) are used only in latter stages of the design.

Parameters can be bound to some values. Every parameter is associated with a set of possible values which can be assigned to this parameter. Of course, only values of active parameters are considered. Values of inactive parameters (if they are assigned to them) are irrelevant, not up-to-date, or meaningless. From the point of view of assignment, parameters can be divided into two groups:

- independent;
- dependent.

The value which can be assigned to an independent parameter does not depend on the values of other parameters. It can be an arbitrary value from the set of all possible values.

The value of a dependent parameter depends on the values of other parameters (these parameters should be active — if the value of a dependent parameter depends on the value of an inactive parameter then this dependent parameter should be inactive too). The dependency can be *tight* when the value of a dependent parameter can be determined unambiguously from the values of parameters on which the initialized parameter depends (i.e. it is defined by a function). Another case stands for the *loose* dependency when the values of some parameters can restrict the possible values of a dependent parameter. The first type of dependency should be implemented by dependent parameters. The latter type can be implemented by the use of dependent parameters (with changeable set of possible values) or by independent parameters (the consistency is guaranteed by a particular constraint).

From the point of view of implementation it means that a value can be assigned to an independent parameter (directly from its set of possible values) and the value of a dependent parameter is computed from the values of other parameters. In other words, it is needed first to determine values of independent parameters and only after that the values of dependent parameters.

In general, the designer does not determine the values of parameters in a random order but uses a predefined order of parameter processing (although this order can depends on the current stage of the solution). That is why a parameter *priority* is sometimes introduced. The higher priority a parameter has, the higher chance to assign value to this parameter in an early stage. The solution always starts with the initialization of the independent parameter with the highest priority.

3.2 Constraints

The required functionality of a designed artefact is guaranteed by a set of *constraints*. The constraints represent relations defined on the values of parameters, expressing dependencies among these parameters. Every constraint is associated with a n-tuple of parameters. Exactly one parameter from these parameters is constrained by this constraint. Its value is dependent on the values of the other parameters from this n-tuple. On the other hand, one constraint can restrict values of more parameters — each of them is limited by one instance of the constraint.

Constraints (similar to parameters) can be both *active* or *inactive*. The active constraints define the conditions which should be met in order to ensure the required functionality of the constructed artefact. These conditions represent a set of necessary conditions at the current stage of the design process — if they are not satisfied the artefact is not acceptable. The inactive constraints are not important at the current design stage. Thus there is no need to check them, the functionality of the artefact can be achieved despite their violations.

Only those constraints can be active which are imposed solely on active parameters. If one of these parameters becomes inactive, the constraint is inactivated too. A constraint may be inactivated during the design process (e.g. in order to simplify the solution or as a result of an activation of another constraint etc.). On the other hand, the activation of an inactive parameter can cause the transfer of one or several constraints from the set of passive constraints into the set of active constraints. In this way the number of active constraints depends on the fact which parameters are currently active and on the history of the solution.

In order to achieve an acceptable solution it is necessary to satisfy all active constraints. Only seldom it is done in a random or arbitrary order. A designer usually distinguishes between more and less important constraints. The *priority* of the constraints can express designer's intention. First a constraint with higher priority is processed and then constraints with lower priority.

If some constraint is violated, the solution must be modified in a proper way in order to remove this violation. A *fix* is looked for — the correction to a solution capable to reestablish the validity of the constraint. This can change the value of one or more parameters (it can modify not only the constrained parameter but a constraining one too). Some constraints have associated fixes and some have not. If there is no fix available to correct a constraint, in the case of the violations of this constraint the solution cannot be modified in order to remove this violation. On the other hand, some constraints can have more fixes which can differ from each other by its priority. This priority can be dependent on the current stage of the solution.

If a constraint cannot be satisfied using a fix associated with this constraint, then the next step depends on the fact whether this violated constraint stands for a necessary or desirable condition. The solution cannot be constructed and the design process fails if the constraint represents a necessary condition (then the only output is information about the cause of the failure). If the constraint does not represent a necessary condition and the artefact can be designed despite the violation of this constraint (the quality of the solution falls a bit but remains acceptable), then the constraint can be *relaxed* — it becomes inactive and it is not checked in the next stage of the process.

Constraints can influence each other. The relaxation of one constraint can cause the activation or inactivation of other constraints. Similarly, fixing a constraint can violate other constraints or can require the inactivation of those constraints.

4 Solution of the configuration design problem

To solve the configuration design problem means to assign a value to each active parameter in such a way that no active constraint is violated. The design process can be represented by a trajectory in two dimensional *design state space* [1] where the

x-axis represents the number of active unassigned parameters

y-axis represents the number of violated active constraints

(the constraint validity of which cannot be checked because this constraint is defined over an unassigned parameter is not considered to be violated).

The search for the solution can be depicted in the design state space as the transition from the *initial* state to the *goal* state. At the beginning of the search all parameters are unbounded and no constraint is violated. As a result, the initial point of the trajectory is located on the x-axis and its distance to the origin is equal to the number of parameters which are active at the beginning of the search. Since the successful solution stands for the case when all active parameters are assigned and no active constraint is violated, the trajectory in this case ends in the origin of the system of coordinates. If the trajectory ends above the x-axis, not all constraints are satisfied and the search fails. The final point of the trajectory on the x-axis but not in the origin indicates that some parameters remain unbound (and they are not important for the final solution or there is no knowledge available how to assign a value to them).

There are many *strategies* how to solve configuration design problems. Every strategy can produce the trajectory of a characteristic shape in the design state space. In general, the solution consists of various design steps. There are several different basic types of steps:

- The *assignment* step assigns one parameter. This assignment can violate some constraints. The step is represented in the state space by an arrow pointing to the left (it is parallel to the x-axis if the step does not violate any constraint).
- The *corrective* step modifies the values of some parameters in order to remove a constraint violation. It is represented by an arrow parallel to the y-axis (if the number of violated constraint decreases, the arrow is pointing downwards).
- The *activation* step activates one or more parameters and puts them into the current solution. Since activated parameters are unbound, the step is represented in the state space by an arrow parallel to the x-axis and pointing to the right.
- The *inactivation* step inactivates one or more parameters and deletes them from the current solution. The step can induce the inactivation of some constraints associated with inactivated parameters. It is represented in the state space by an arrow pointing to the left (it is parallel to the x-axis if no constraint is inactivated).
- The *relaxation* step inactivates some constraints and activates others. The step is represented in the design state space by an arrow parallel to the y-axis.

Various strategies can be constructed using these steps. One of them is the *Complete-Revise-Optimize* (CRO) strategy which is closely related to the strategy *Complete Model and Revise* [1]. It is based on the idea of dividing the design process into three phases:

1. *Initial* solution. In this phase some values are assigned to all active parameters. This initial solution is very often unsatisfactory because it does not achieve the required functionality of the designed artefact (the initial assignment is not required to take into consideration constraints imposed on the values of parameters). The aim is to obtain some solution (maybe erroneous but with all parameters assigned) as soon as possible.

2. *Acceptable* solution. This phase modifies the solution in order to adapt it to the available (active) constraints. The result of this step is the solution (arbitrary one from the set of all acceptable solutions) which does not violate any active constraint. The values of parameters are only modified — the only exception is the initialization of activated parameters.

3. *Optimal* solution. The solution is modified in such a way that minimizes or maximizes some feature of the designed artefact (at the same time the acceptability of the solution has to be preserved).

First some (possible erroneous) solution is obtained which can be viewed by a designer as a whole and not only as a part of the future solution. It is useful only in the case when domain knowledge guarantees that the process does not deviate too far away from the solution. The designer has the possibility first to create a rough solution which corresponds to basic constraints. In the next stage this solution can be corrected from more detailed point of view. The designer modifies the rough solution in order to meet all constraints imposed on the solution. This approach enables processing in a hierarchical way where only main parameters are processed during the initialization phase and details of the solution are represented by other parameters which are activated in latter phases of the design. Another advantage of dealing with the whole solution is that the designer has an overview of all violated constraints at the same time. The designer can select the best strategy to remove these violations by the use of available fixes whilst taking into account the priority of violated constraints.

The third phase (which is not mandatory and can be skipped) corresponds to the phase of the smooth sensitive refinement of the solution when both older and newer solution are acceptable. The difference is only in the quality of the solution. This difference must not influence the functionality of the designed artefact. It can represent e.g. lower price or lower complexity of the artefact. This phase is related to the fact that the designer is not interested in any acceptable solution but only in the best one.

If there is no need to use either the activation step or the inactivation step (so that the number of active parameters remains the same), typical trajectory in the design state space has the form depicted in Figure 1.

The first phase uses only the assignment step. The trajectory travels from its initial point pointing to the left and ends on the y-axis. Since the number of violated constraints increases, the trajectory increases too. During the second phase the trajectory falls down into the origin. The last phase is represented only by the origin because the trajectory does not leave it. If the relaxation step occurs, the decrease of the trajectory can be temporary replaced by its increase. While correcting the violated constraints their number can increase. The reason

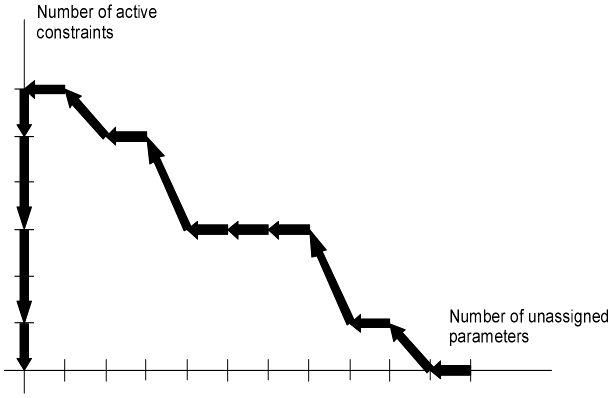

Figure 1: A typical trajectory in the design state space.

is that the designer fixes the violated constraint with the highest priority and the correction of this constraint can cause several violations of constraints with lower priority.

The strategy *Complete-Revise-Optimize* consists of several interconnected tasks as it is depicted in Figure 2.

The first task is the *test for solution existence* which tests the possibility of finding a solution. It is not a detailed test taking into account all constraints. It is usually a rough heuristic which can detect the impossibility of finding a solution under particular circumstances. The test represents the necessary condition of finding a solution. If the test does not reject the possibility of finding a solution, this solution may or may not be found. The aim of the task is to filter out those cases when it is obvious that the solution does not exist. The task is not mandatory — it can be skipped.

The task *initialize* is the first phase of the design process. It is a compulsory task which assigns a value to each active parameter from the given set of parameters. The initialization can be done in two ways — one approach is based on the domain design knowledge and the other one uses an implicit domain independent strategy. It is an advantage if designer is able to provide special knowledge how to initialize particular parameters in order not to violate some of the constraints related to these parameters. Such an initialization can decrease the number of fixes used in the second stage of the design. An implicit initialization can be performed for instance by random choice of a value from the set of possible values for the given parameter. Of course, the combination of both approaches can be used — first some parameters are initialized using domain knowledge and the others are initialized in an implicit way. It is beneficial if the task contains domain knowledge (the explicit

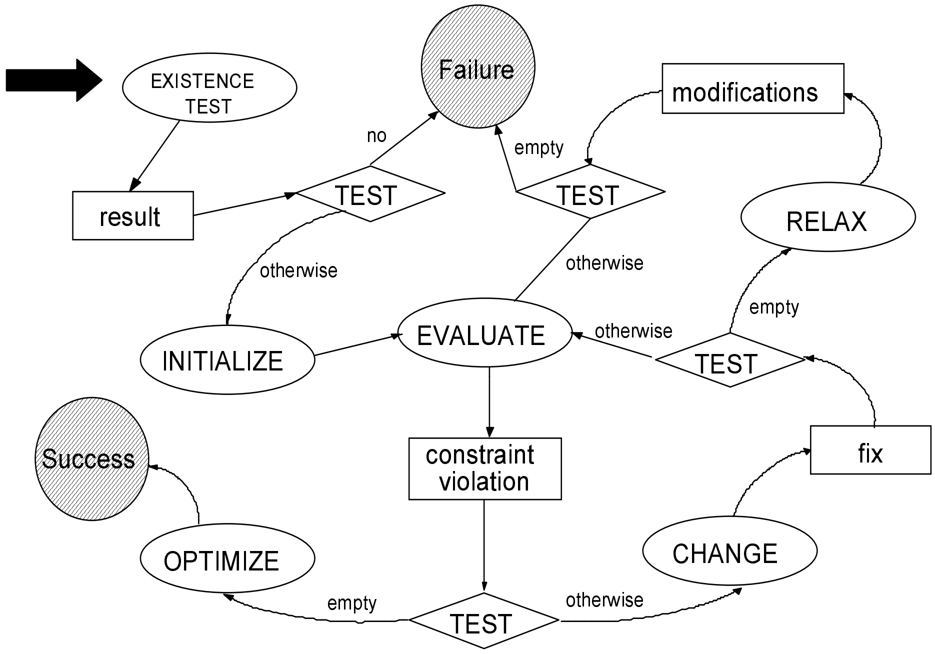

Figure 2: The CRO strategy.

initialization can be performed), but it is not necessary (all parameters can be initialized by implicit initialization).

The second phase consists of the net of three tasks (*evaluate*, *change*, and *relax*) and typically consists of several cycles of these three tasks. In each loop the system tries to remove only one constraint violation (but it may happen that the correction simultaneously corrects other violations too). The number of loops depends on the violated constraints. The task *evaluate* detects changes of constraint violations (which constraint is marked to be violated but currently it is satisfied or on the contrary which constraint is currently violated despite the fact that it is marked as valid) and in accordance with the result of this detection it updates the trajectory in the design state space. The outcome of the task is the list of all violated constraints together with all parameters which violate these constraints. The task is based exclusively on domain knowledge.

If at least one parameter is violated, the activity is passed to the task *change*. The task *change* then selects one constraint from the set of violated constraints to be corrected. The choice can be done either in a dynamic way according to the current stage of the solution (the combination of violated constraints) or in predefined order (e.g. given by the priority of particular constraints). The dynamic choice enables minimization of the number of used corrections (one fix can correct more constraints at the same time), but it requires additional domain knowledge.

When a constraint to be corrected is selected, a fix to perform the correction is looked for. The fixes are represented by domain design knowledge. An appropriate fix is executed and the activity is transferred back to the task *evaluate*. If more different fixes are applicable, only one can be used (the problem of selection this fix is similar to the selection of one violated constraint which should be corrected).

If for the selected constraint violation no fix is available, the solution cannot be modified in order to remove this processed violation. In this case the task *relax* becomes active. This task investigates whether this violated constraint has an influence on the basic functionality of the artefact or only on some desired feature of the artefact. If the constraint does not represent a necessary feature, it can be relaxed. The task *relax* is based on two types of domain knowledge — knowledge about the importance of particular constraints and consequences of their violations, and knowledge how to relax those constraints which can be relaxed.

If the task *evaluate* does not detect the violation of any active constraint, the second phase of the design (the creation of an acceptable solution) ends. The current form of the solution can be used as the final solution or it can be further *optimized* in the third phase of the design process. The optimization can be performed only using of domain knowledge.

In the case of the impossibility of creating a final solution (detected by the task *test for solution existence* before the design starts or as the consequence of the impossibility of correcting a constraint which cannot be relaxed) the system is able to provide the reason for this failure.

5 Implementation of configuration design problem in CLIPS

CLIPS[1] has been used for the development of numerous knowledge based applications. It is a *production system* language with a *rule base* and a *working memory* (database of facts and instances). Rules consists of three parts:

- initial declaration section (optional);
- condition part (antecedent of the rule);
- action part (consequent of the rule).

The condition part consists of a number of conditional elements. All these elements must be satisfied by the current state of the database in order to activate a rule. The action part consists of an ordered sequence of actions. Every rule has the following form

[1]CLIPS was created and is currently maintained by NASA (Software Technology Branch, Johnson Space Center).

```
(defrule <rule name>
   (declare (   ...   ))     - declaration part
      ⋮                       - condition part
=>
      ⋮                       - action part
)
```

In addition to the rule based programming, CLIPS enables the use of *object oriented* and *functional* programming. More information about CLIPS can be found in [2].

5.1 Task control

The *control activity* is transferred from task to task during the search for a solution of the problem. This transfer can be either simple (sequential — task 1 always follows task 2) or more complex when the exact form of the activity transfer depends on the current state of the solution (e.g. on the result of some test). Control transfer is performed in the "background". When a task becomes active, it is active till it returns this activity back to the task control mechanism (in the current state of the solution there is no need for the task to remain active — this stage is detected by the active task). Then the task control passes the activity to the next appropriate task.

Each task consists of a set of rules. While some task is active, only rules of this task and rules dedicated to the activity transfer to the next task can be fired. The rules of the task have higher priority than the activity transfer rules. This enable first to fire all relevant rules of the active task. The rules for the task control (the choice of the next task and the passing the activity to this task) can be fired only when there is no rule of the active task which is relevant in the current situation.

Which fact is currently active is specified by a *control token*. This token is the fact of the following form

```
(job <name of the active task>)
```

which can be found in the database. To divide all rules into groups according to the tasks they belong to, every rule contains the test for the activity of its task at the beginning of its condition part. Thus a general form of rules belonging to a particular task is as follows

```
(defrule <rule name>
      (job <name of a task the rule belongs to>)
      ...
=>
      ...
)
```

The transfer of the activity from one task to another can be implemented in two steps:

1. remove the old control token from the database
2. assert the new control token with the name of the next task into the database

An example of a simple sequential activity transfer is the transfer of the control token from the task *initialize* to the task *evaluate*:

```
(defrule end-initialize-1
   (declare (salience -50))
      ?a <- (job initialize)
=>
      (retract ?a)
      (assert (job evaluate))
)
```

Similar schema is used to activate the task *test for solution existence*. The only difference is that this task is always activated as the first one. The rule belonging to this task has a very simple form

```
(defrule start-1
   (declare (salience -50))
=>
      (assert (job test-exist))
)
```

More complex activity transfer, based on the result of the previous task, consists of several rules. For instance to choose the next task after the task *evaluate* returns the control:

```
(defrule end-evaluate-1
   (declare (salience -50))
      ?a <- (job evaluate)
=>
      (retract ?a)
      (assert (job test1))
)


(defrule end-test1-1
   (declare (salience -50))
      ?a <- (job test1)
      (not (object (is-a constraint)
                   (cons-violation yes)))
=>
      (retract ?a)
      (assert (job optimize))
)


(defrule end-test1-2
   (declare (salience -50))
```

```
        ?a <- (job test1)
        (object (is-a constraint)
                (cons-violation yes))
=>
        (retract ?a)
        (assert (job change))
)
```

where first the simple transfer to the appropriate test is performed. The test always consists of several rules (the number of rules corresponds to the number of different output values of the test). These rules have to contain the appropriate form of the test in their condition parts (in the example above the presence of constraint violations is tested).

The test can be performed as the test of the current state of the solution. Another possibility is to test the result of the activity of the previous task. The previous task could assert its result into the database represented by one of the facts `(result success)` or `(result failure)`. The next task should retract this information from the database. This is true for the control after the tasks *test for solution existence*, *change*, and *relax*. As an example, the part of the task control after the task *change* is finished

```
(defrule end-test2-2
   (declare (salience -50))
        ?a <- (job test2)
        ?b <- (result failure)
=>
        (retract ?a)
        (retract ?b)
        (assert (job relaxation))
)
```

Tasks *success* and *failure* are implemented as special tasks which are able to display the message about the success of the search of the solution and to provide the current state of this solution.

5.2 Representation of basic objects

Basic parts of the configuration design problem structure are:

- the set of parameters (representing the solution of the problem);
- the set of components (expressing resources which are available to the formation of the final solution);
- the set of constraints (defining required functionality of the constructed artefact).

All these parts can be implemented as objects in COOL (the object part of CLIPS). Each part is represented by one or several classes and instances of these classes stand for particular parameters, components, and constraints of the problem being solved. All these classes are defined as subclasses of the parent class USER which represents the root of user defined class hierarchy.

Parameters which form the solution can be implemented by the class `parameter`:

```
(defclass parameter
      (is-a USER)
      (role concrete)
      (pattern-match reactive)
           (slot param-name
                 (access initialize-only)
                 (create-accessor read-write))
           (slot active
                 (allowed-values yes no)
                 (default yes)
                 (access read-write)
                 (create-accessor read-write))
           (slot priority
                 (access read-write)
                 (create-accessor read-write))
           (slot param-type
                 (allowed-values dependent independent)
                 (default independent)
                 (access initialize-only)
                 (create-accessor read-write))
           (slot param-value
                 (access read-write)
                 (create-accessor read-write))
           (multislot param-domain
                 (access initialize-only)
                 (create-accessor read-write))
)
```

Some slots have defined default values. It is not compulsory, but it makes the parameter definition of a particular application easier.

From the point of view of access, there are two types of slots. The first type provides only the possibility of reading the values assigned to the slots. The second type enables to change the values of these slots. For read-only slots it is enough to set their values only once when instances are created. It is possible to do using CLIPS features — setting the `initialize-only` access (which contains reading too) to the slot. If the value of a slot can be changed the access to this slot has to be `read-write`.

The slot param-domain has the form `(param-domain <domain type> <domain>)`. It defines the type of the *domain* and the values which can be assigned to the parameter (in the room allocation problem only enumerative type is allowed). `<domain>` represents one or several values depending on the type of the domain.

The class `component` is dedicated to the representation of components which can form the artefact. But a particular application can use a large number of different components. That is why the class `component` does not represent a particular component (it cannot create instances — it is an *abstract* class), but serves as the parent class for the definition of all possible components. This parent class contains only information which concerns all components (and they can inherit it from the parent class `component`).

```
(defclass component
      (is-a USER)
      (role abstract)
           (slot comp-name
                 (access initialize-only)
                 (create-accessor read-write))
           (slot comp-number
                 (default 1)
                 (access initialize-only)
                 (create-accessor read-write))
)
```

Information which is common to all components is only the name of the component and the number of components of the same name.

Different component types can be defined by subclasses of the parent class `component`. These subclasses have the following general form:

```
(defclass component-<component type>
      (is-a component)
      (role concrete)
      (pattern-match reactive)
      ...
)
```

Every component type is represented by exactly one subclass. In this way the number of these subclasses is equal to the number of different component types which are available. The number and the definition of slots are different for different component types and depend on a particular application.

Since components are predefined (and their features are not changed during the search for the solution), their slots are available only for reading. Every slot (concerning a component) has the form

```
           (slot <slot name>
                 ⋮              - information on one feature
                 (access initialize-only)
                 (create-accessor read-write))
```

Constraints can be defined as instances of the class `constraint` which has the following form:

```
(defclass constraint
      (is-a USER)
```

```
        (role concrete)
        (pattern-match reactive)
             (slot cons-name
                   (access initialize-only)
                   (create-accessor read-write))
             (slot priority
                   (access read-write)
                   (create-accessor read-write))
             (multislot cons-param
                   (access initialize-only)
                   (create-accessor read-write))
             (slot cons-violation
                   (allowed-values yes no)
                   (default no)
                   (access read-write)
                   (create-accessor read-write))
             (multislot viol-param
                   (default (create$ ))
                   (access read-write)
                   (create-accessor read-write))
             (slot active
                   (allowed-values yes no)
                   (default yes)
                   (access read-write)
                   (create-accessor read-write))
)
```

The slots which are used only to read information and for which there is no need to change their values are defined as `initialize-only`. Since the priority of a constraint can vary during the search for the solution depending on the state of other constraints, the slot `priority` enables to change its value.

The slot `cons-param` contains the list of all parameters which are relevant to the constraint. If some of them violate the constraint, the value of the slot `cons-violation` is set to *yes* and this parameter is registered in the `viol-param` slot. If there is no parameter to violate some constraint, the slot `viol-param` of this constraint is empty.

Since the management of the slots `viol-param` is not trivial (modification of a part of the slot content always occurs), it is not possible to use default accessors. New accessors have the form

```
(defmessage-handler constraint del-viol-param primary (?param)
      (bind ?position (member$ ?param ?self:viol-param))
      (bind ?self:viol-param (delete$ ?self:viol-param ?position ?position))
)

(defmessage-handler constraint add-viol-param primary (?param)
      (bind ?self:viol-param (insert$ ?self:viol-param 1 ?param))
)
```

5.3 Task implementation

5.3.1 Task test for solution existence

This task is based on the presence of domain knowledge. Since the task is not mandatory, it is possible to skip it. In order to preserve the task control scheme in the case of skipping over, the task always contains the rule:

```
(defrule test-exist-y
   (declare (salience -10))
      (job test-exist)
      (not (solution ?))
=>
      (assert (solution yes))
)
```

which ensures a proper transfer of the activity to the next task *initialize*. In the case of the presence of domain knowledge for this task, it is represented in the form of rules

```
(defrule test-exist-n-<order number>
      (job test-exist)
      ...
=>
      (assert (solution no))
)
```

which are able to detect the case when the solution cannot be found (in other words, necessary condition for the solution existence does not hold). If this is not the case (and the solution for a particular problem does exist), no rule from these domain rules can be fired. In this case the database does not include the fact related to the non-existence of the solution and the rule `test-exist-y` can be fired.

5.3.2 Task initialize

Only active parameters are initialized during this task. In general, independent parameters should be initialized first and only then the dependent parameters can be processed. Independent parameters can be initialized in two ways:

- explicit (by the use of domain knowledge);
- implicit.

Explicit parameter initialization is preferable to an implicit initialization of this parameter (which is used only if the present explicit initialization knowledge cannot be employed for this parameter for some reason). That is why the rules representing designer's domain initialization knowledge should have higher priority. A typical rule for domain initialization has the form:

```
(defrule init-<parameter name>
   (declare (salience 30))
      (job initialize)
      ?a <- (object   (is-a parameter)
                 (param-name ?b)
                 (active yes)
                 (param-value nil)
                 (param-type independent)
      :   - the calculation of an initialization value ?c
=>
      (send ?a put-param-value ?c)
)
```

The implicit initialization depends on the type of parameter domain of the initialized parameter. If the domain is enumerative, the initialization can use the rule:

```
(defrule init-enum
      (job initialize)
      ?a <- (object   (is-a parameter)
                 (param-value nil)
                 (active yes)
                 (param-type independent)
                 (param-domain enum ?b $?))
=>
      (send ?a put-param-value ?b)
)
```

To initialize dependent parameters rules with domain knowledge (expressing dependence of these parameters on the values of other parameters) have to be given. These rules are different for different problems. Such a rule has to test whether all parameters, on whose values the initialized dependent parameter depends, are already initialized. This test corresponds to the condition of applicability of such a rule.

Since the definition of a parameter can include a priority of this parameter, the initialization can be performed in accordance with the priority of initialized parameters (if more than one parameter can be initialized in some step, then the parameter with the highest priority is preferred). But in the case of the solution strategy which initializes all parameters before dealing with constraints and their violations, the use of the priority in the initialization step does not make any sense.

5.3.3 Task evaluate

This task can be accomplished only by domain dependent rules which correspond to the definitions of particular constraints. Every constraint can be related to two groups of rules:

- rules dedicated to detection whether some parameter violates the constraint which is being checked and at the same time this parameter is not registered in the slot param-viol of this constraint. If there is such a parameter, it is put on the list in this registration slot. The rules have usually the following form:

```
(defrule cons-<constraint name>-y
      (job evaluate)
      (object (is-a constraint)
           (cons-name <constraint name>)
           (active yes))
      (object (is-a parameter)
           (param-name ?a)
           ⋮  - the detection of the constraint violation
                        by some parameter
      (not       (object (is-a constraint)
                 (cons-name <constraint name>)
                 (viol-param $? ?a $?)))
=>
      (send [<constraint name>] add-viol-param ?a)
)
```

- rules detecting whether some parameter is registered in the slot viol-param of the investigated constraint although it does not violate this constraint. If there is such a parameter, it is deleted from this registration slot. These rules have the following general form:

```
(defrule cons-<constraint name>-n
      (job evaluate)
      (object (is-a constraint)
           (cons-name <constraint name>)
           (active yes))
      (object (is-a parameter)
           (param-name ?a)
           ⋮   - the search for a parameter which does not
                      violate the constraint
      (object (is-a constraint)
           (cons-name <constraint name>)
           (viol-param $? ?a $?))
=>
      (send [<constraint name>] del-viol-param ?a)
)
```

If some group consists of more than one rule, the order number of the rule forms the last (hyphenated) part of the name of this rule.

The analysis of some constraint violations is performed only for those parameters which can violate the investigated constraint. The list of these parameters can be found in the slot cons-param of the constraint under investigation or it can be indicated directly in related rules (what is less elegant but it can save one test in the condition part of these rules).

There are two special cases when the last parameter is deleted from the slot viol-param and the slot is empty (thus the constraint is not violated although in its slot cons-violation remains the value *yes*) or a parameter is asserted into the empty slot `viol-param` (thus the constraint is violated although in its slot `cons-violation` remains the value *no*). In order to keep all slots of every constraint consistent, there are rules:

```
(defrule violation-no
   (declare (salience 10))
      ?a  <-  (object (is-a constraint)
                 (cons-violation yes)
                 (viol-param))
=>
      (send ?a put-cons-violation no)
)

(defrule violation-yes
   (declare (salience 10))
      ?a  <-  (object (is-a constraint)
                 (cons-violation no)
                 (viol-param ? $?))
=>
      (send ?a put-cons-violation yes)
)
```

The priority of these rules enables their firing before the activity transfer to the next task.

5.3.4 Task change

The aim of the task is to chose one constraint violation and to correct it. In such a way the cycle consisting of the two tasks *evaluate* and *change* tries to eliminate violations of all active constraints. In every step the most important constraint violation is corrected. This order is based on the priority of constraints. The priority can be defined as a value of the slot `priority` in the constraint definition. Another way (and a simpler one from the point of view of implementation) is to express the priority as the priority of rules related to this task.

If it is not possible to correct a constraint violation with the highest priority (there is no rule applicable in the given situation), the task does not try to correct another constraint violation but ends without success and asserts the name of this non-correctable constraint into the database. Thus the task *relax* can pick up the name of the constraint to be relaxed.

The task consists only of designer's domain knowledge. Every constraint associated with one or more fixes is accompanied by a group of rules. These rules have the following form:

```
(defrule change-<constraint name>-<order number>
   (declare (salience <constraint priority>))
      (job change)
      (not (result ?))
      (object (is-a constraint)
           (cons-name <constraint name>)
```

```
            (cons-violation yes)
            (viol-param ?a $?))
            ...
=>
            ...
        (assert (result success))
)
```

The rule for the correction of some constraint violation can be used only if no constraint with higher priority is violated. The reason is that every used rule in this task asserts the fact `(result <result>)` into the database. In this way the use of the rule with higher priority prevent the use of the rule with lower priority. If in the current situation there is no applicable rule which can remove a constraint violation with the highest priority, the rule

```
(defrule change-<constraint name>-no
   (declare (salience <constraint priority - 1>))
      (job change)
      (not (result ?))
      (object (is-a constraint)
            (cons-name <constraint name>)
            (cons-violation yes))
=>
      (assert (result failure))
      (assert (constraint-to-relax <constraint name>))
)
```

is fired. This rule should have lower priority than any rule for the correction of the same constraint and at the same time higher priority than rules for the correction of constraints with lower priority. Thus the difference between priorities of two arbitrary constraints (with different priorities) should be at least 2.

Every constraint should be accompanied by knowledge of how to satisfy it. This knowledge can be located in the task *initialize* (parameters are initialized in such a way that this constraint is not violated by these parameters). Another possibility is to put knowledge into the task *change* in order to remove the violations of this constraint. The last possibility is to use this knowledge in the task *relax*.

5.3.5 Task relax

This task depends on domain knowledge too. If the task is successful and some constraint has been relaxed, information on this fact is inserted into the database. This information can be used by the test following this task. If there is no proper rule to relax some violated constraint, the task retains the information about this in the database:

```
(defrule relax1
   (declare (salience -10))
      (job relaxation)
```

```
        (not (result success))
=>
        (assert (result failure))
)
```

Since the fact of failure can occur in the database only if the task is not successful, the priority of the rule `relax1` should be lower than the priority of other rules of the task.

Relaxation of some constraint can cause the inactivation of a constraint. If this inactivated constraint is violated, its slots `viol-param` and `cons-violation` should be set properly (because now they have no sense)

```
(defrule non-active-constraint
   (declare (salience 10))
      ?a <- (object (is-a constraint)
                    (active no)
                    (cons-violation yes))
=>
      (send ?a put-cons-violation no)
      (send ?a put-viol-param (create$ ))
)
```

Priority of the rule ensures that these two slots can be changed before the activity is transferred to the next task.

The form of rules dedicated to the relaxation of constraints can vary a lot. Basic pattern of these rules is:

```
(defrule relax-<constraint>-<order number>
      (job relaxation)
      ?a <- (constraint-to-relax <constraint>)
      ...
=>
      ...
      (retract ?a)
      (assert (result success))
)
```

The rule expects information on which constraint is to be relaxed in the database. This information has been asserted to the database by the previous task.

The activities contained in the action part of these rules can vary. They can contain a simple inactivation of the relaxed constraint (which can depend on the current state of the solution), the activation and/or inactivation of other constraints, or the modification of parameter values. Several rules can be associated with the relaxation of one constraint (they can be ordered by their priority).

5.3.6 Task optimize

This task is not compulsory. If there is no rule to optimize the solution, the task is empty. The only rule which is applicable in this case transfers the activity to the task *success*. If knowledge how to replace a particular allowed combination of parameter values by another allowable combination of values of these parameters is available, it can be expressed by the rule:

```
(defrule opti-<order number>
      (job optimize)
            ...
=>
            ...
)
```

which modifies the values of one or several parameters (by the actions contained in its action part).

If more rules are applicable, in general it is possible to use an arbitrary rule from the set of applicable rules. On the other hand, these rules can differ from one another by their priority. Another possibility is to define an additional parameter which guarantees that only one rule from a group of rules can be fired (the used rule sets this parameter in such a way that the condition parts of the other rules from the group are not satisfied any more).

The rules of this task try to maximize or minimize some variable which can be (but it is not a rule) represented by a parameter. Since the value of this parameter does not make sense in the previous tasks, the parameter has not been active until now. Thus, it is time to activate it:

```
(defrule opti-aktiv
   (declare (salience 30))
      (job optimize)
=>
      (send [optim] put-active yes)
)
```

This rule has to be fired as the first rule within the task (it is given by its high priority). To compute the value of the param `optim` there should be a rule which is fired if this parameter is activated or if the solution is changed. Thus this rule has the form:

```
(defrule opti-value
   (declare (salience 40))
      (job optimize)
      (object (is-a parameter)
               (param-name optim)
               (active yes))
      ...
=>
      (send [optim] put-param-value (optim-value ?b ... ))
)
```

which first computes the values of those parameters on which the parameter `optim` depends and then computes the value of the parameter `optim` and assigns this value to the slot `param-value` of the parameter `optim`.

5.4 Initialization of the solution

Before starting the search for the solution by the task *test for solution existence* all particular objects used by the rules have to be prepared. These objects are represented by instances of concrete classes (`parameter`, `constraint`, and `component-<?>`). These instances stand for real objects which are available — the content of these instances is domain dependent. They are defined in the form

```
(definstances of-<class-name>
     (<instance name> of <class name>
          (<slot name> <slot value>)
          ...
          )
     ...
)
```

Every instance has one slot defining name of this instance (e.g. the slot `param-name` of the class `parameter`). Thus, those instances, which do not need to receive messages sent by the `send` statement, can used the function `(gensym*)` to generate `<instance name>`. Values of slots which have default values do not have to be defined if these default values can be used.

5.5 Debugging

It has been proven that the most valuable debugging CLIPS tool is the printout of the current state of the solution (parameters and constraints) in addition to the stepping feature of the CLIPS. The easiest way to print the state of the all parameters is

```
(deffunction dp ()
     (progn$ (?item (find-all-instances ((?q parameter)) TRUE))
          (printout t (send ?item get-param-name) "   "
                      (send ?item get-param-value) crlf)
     )
)
```

This function displays the name and the value of all parameters. The modification of this function to obtain values of other slots is straightforward. The function for the print of constraint is very similar.

5.6 Application structure

Every application consists of two main parts — a *general* part and a *domain dependent* part. Those two parts form program which can be interpreted by CLIPS. Because of the transparency this program is divided into several files which follow the definition mentioned above:

- files with domain dependent information and knowledge (with different contents for different problems). The names of these files are constructed according to the scheme `<system part>-dom.clp`
- files with domain independent parts (they do not change for different problems). The names have the form `<system part>-sys.clp`

The final application consists of up to fourteen files:

cons-sys.clp	-	constraint management
cont-sys.clp	-	task control
deb-sys.clp	-	debugging functions
inic-sys.clp	-	implicit initialization
relx-sys.clp	-	constraint relaxation (system part)
comp-dom.clp	-	component definitions
cons-dom.clp	-	constraint definitions
inic-dom.clp	-	parameter initialization
mode-dom.clp	-	parameter definitions
modi-dom.clp	-	domain knowledge for the task *change*
opti-dom.clp	-	domain knowledge for the task *optimize*
rela-dom.clp	-	relations and supporting functions
relx-dom.clp	-	domain knowledge for the task *relax*
solu-dom.clp	-	domain knowledge for the *test for solution existence*

6 Room allocation problem

6.1 Problem definition

The problem described in this part of the report has been inspired by the *Sisyphus I* Problem [3]. The goal is to assign a group of people to available rooms. The people, who have to be assigned, are characterised by their features. A set of rooms is described by another set of attributes. The problem is to allocate these people within available rooms in such a way that the allocation meets the criteria for the required functionality of the group of people as a whole.

For the illustration of this problem let us consider personnel of a research team. Individuals (members of the team) differ from one another by their position within the team hierarchy,

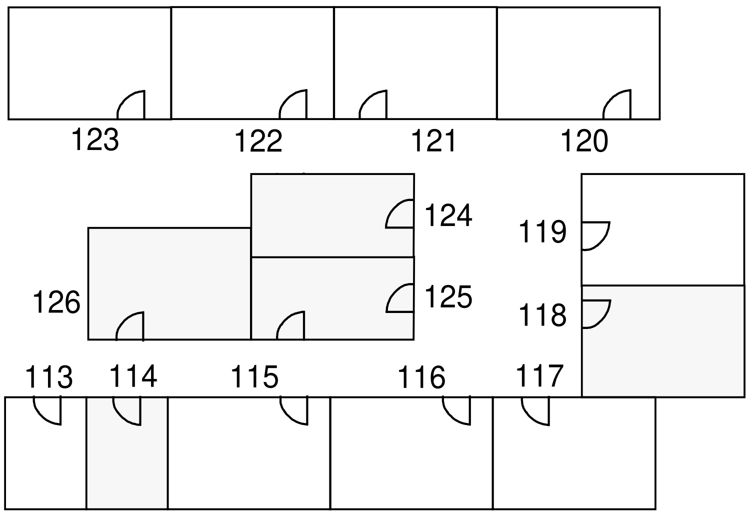

Figure 3: Available rooms.

projects they participate on, and their attitude to smoking and computers. The group consists of fifteen people:

name	*position*	*project*	*smoker*	*hacker*
thomas	head-of-group	mlt		
hans	head-of-project	babylon	yes	
katharina	head-of-project	mlt	yes	yes
joachim	head-of-project	aserti		
werner	researcher	babylon		yes
angi	researcher	babylon		
harry	researcher	babylon		yes
jurgen	researcher	mlt		yes
marc	researcher	aserti		
uwe	researcher	mlt	yes	yes
andy	researcher	mlt	yes	
michael	researcher	mlt		yes
monika	secretary			
ulrike	secretary			
eva	manager			

These persons have to be assigned to new rooms according to Figure 3.

Rooms of different sizes are available. Let us consider only two types of rooms — small rooms (113, 114, 124, and 125) and large rooms. Only one person can be assigned to a small room. In a large room two persons can be llocated simultaneously. Marked rooms (114, 118, 124, 125, and 126) are not available — they cannot be used. Rooms 115, 116, 117, 118, 119, 124 and 125 are central.

The fact that only one person is allowed to be assigned to a small room and maximum two person to a large one can be expressed in the form of two constraints:

```
general1:
(all x y v)      (anybody(x) & in-room(x v) & anybody(y) & in-room(y v)
                 ==> eq(x y))
general2:
(all x y z v)    (anybody(x) & in-room(x v) & anybody(y) & in-room(y v)
                 & anybody(z) & in-room(z v)
                 ==> eq(x y) v eq(x z) v eq(y z))
```

These two constraints hold for any person. The other constraints are relevant only for a group of people (there are separate sets of constraints for people of different position within the hierarchy):

HEAD-OF-GROUP

Head-of-group can be allocated in a large central room. At the same time, he cannot share this room with anybody else.

```
head-of-group1:
(all x y)        (head-of-group(x) & in-room(x y)
                 ==> central(y))
head-of-group2:
(all x y)        (head-of-group(x) & in-room(x y)
                 ==> large(y))
head-of-group3:
(all x y z)      (head-of-group(x) & in-room(x z) & anybody(y) & in-room(y z)
                 ==> eq(x y))
```

SECRETARIES

They share a large room close to the room of the head-of-group.

```
sec1:
(all x y)        (secretary(x) & in-room(x y)
                 ==> large(y))
sec2:
(all x y v w)    (secretary(x) & secretary(y) & not eq(x y)
                 & in-room(x v) & in-room(y w)
                 ==> eq(v w))
sec3:
(all x y v w)    (secretary(x) & in-room(x v) & head-of-group(y) & in-room(y w)
                 ==> close(v w))
```

MANAGER

Her office should be large and central. Moreover, her room has to be as close as possible (to have maximum access to) to the head- of-group and secretariat. She cannot share her room.

```
man1:
(all x y)           (manager(x) & in-room(x y)
                    ==> central(y))
man2:
(all x y)           (manager(x) & in-room(x y)
                    ==> large(y))
man3:
(all x y z v p q)   (manager(x) & in-room(x v) & head-of-group(y) & in-room(y p)
                    & secretary(z) & in-room(z q)
                    ==> maximum-access(v p q))
man4:
(all x y z)         (manager(x) & in-room(x z) & anybody(y) & in-room(y z)
                    ==> eq(x y))
```

HEAD-OF-PROJECTS

Preferably they do not share the room but they can share the room with the person of the same position. The room of a head-of-project is small (if it is possible).

```
head-of-project1:
(all x y)              (head-of-project(x) & in-room(x y)
                       ==> small(y))
head-of-project2:
(all x y z)            (head-of-project(x) & in-room(x z) & anybody(y)
                       & in-room(y z)
                       ==> eq(x y))
head-of-project3:
(all x y v)            (head-of-project(x) & in-room(x v) & anybody(y)
                       & in-room(y v) & not eq(x y)
                       ==> head-of-project(y))
```

RESEARCHERS

Smokers and non-smokers should be in different rooms. Two researchers can share the room if both of them work on the same project and only one of them is a hacker.

`res1:`

(all x y z v w) (researcher(x) & researcher(y) & not eq(x y)
& in-room(x z) & in-room(y z) & smokes(x v) & smokes(y w)
==> eq(v w))

`res2:`

(all x y z v w) (researcher(x) & researcher(y) & not eq(x y)
& in-room(x z) & in-room(y z) & project(x v) & project(y w)
==> eq(v w))

`res3:`

(all x y z v w) (researcher(x) & researcher(y) & not eq(x y)
& in-room(x z) & in-room(y z) & hacker(x v) & hacker(y w)
==> not eq(v w))

6.2 Problem implementation

Both persons and rooms can be considered as the components from which the final artefact (the assignment of people to rooms) can be constructed. But these two types of components are different from the point of view of their participation in the final solution. The final solution should assign all people to some room (no one can remain unassigned). On the other hand, it is not necessary to use each available room. These two types of components can be defined by the use of two separate classes which form subclasses of the parent class component.

The people can be represented by the class `component-person`:

```
(defclass component-person
      (is-a component)
      (role concrete)
      (pattern-match reactive)
           (slot position
                 (allowed-values researcher secretary
                         head-of-project head-of-group manager)
                 (access initialize-only)
                 (create-accessor read-write))
           (slot project
                 (type SYMBOL)
                 (access initialize-only)
                 (create-accessor read-write))
           (slot smoker
                 (allowed-values yes no)
                 (default no)
                 (access initialize-only)
                 (create-accessor read-write))
           (slot hacker
                 (allowed-values yes no)
                 (default no)
                 (access initialize-only)
                 (create-accessor read-write))
)
```

This class contains two slots (features) with default values (`hacker` and `smoker`). According to the definition above, hans (head-of-project, project babylon, smoker, and non-hacker) can be for example defined as an instance of this class as follows

```
(definstances of-component-person
      ((gensym*) of component-person
           (comp-name hans)
           (position head-of-project)
           (project babylon)
           (smoker yes))
        ...
)
```

Rooms can be represented by the class `component-room` of the form

```
(defclass component-room
      (is-a component)
      (role concrete)
      (pattern-match reactive)
           (slot usable
                 (allowed-values yes no)
                 (default yes)
                 (access initialize-only)
                 (create-accessor read-write) )
           (multislot next-to
                 (access initialize-only)
                 (create-accessor read-write) )
           (slot central
                 (allowed-values yes no)
                 (default no)
                 (access initialize-only)
                 (create-accessor read-write) )
           (slot size
                 (type SYMBOL)
                 (allowed-values large small)
                 (access initialize-only)
                 (create-accessor read-write) )
)
```

This class contains two slots with defaults values too (in this case slots `usable` and `central`). The room 114 (small, non-central, not available) can be defined as follows

```
(definstances of-component-room
      ((gensym*) of component-room
           (comp-name 114)
           (next-to 113 115 126)
           (usable no)
           (size small))
       ...
)
```

The solution of the room allocation problem consists of a set of parameters. Each of them represents one pair person-room. Since all components of the type `component-person` should be utilized to create the solution, each component can be associated with one person. The value of such a parameter is simply the room to which the corresponding person is assigned. In this case all parameters are independent. Thus for instance the parameter defining the location of thomas can be defined as:

```
(definstances of-parameter
      ((gensym*) of parameter
           (param-name thomas)
           (param-domain (create$ enum 123 122 ... 117))
           ))
       ...
)
```

In the previous chapter 18 constraints were defined. At the beginning of the search for the solution only 17 constraints are active. The only inactive constraint is `head-of-project3` because this constraint has no sense while the constraint `head-of-project2` is active. On the other hand, if the constraint `head-of-project2` becomes inactive, the constraint `head-of-project3` should be activated. The definition of a particular constraint has the form

```
(definstances of-constraint
  ;;; head-of-project can share a room with another head-of-project
      (head-of-project3 of constraint
           (cons-name head-of-project3)
           (cons-param (create$ hans joachim katharina))
           (active no))
       ...
)
```

The priority of constraints (in the descending order) is as follows:

```
       general1 general2
         head-of-group2
         head-of-group1
         head-of-group3
              sec1
              sec2
              sec3
              man2
              man1
              man3
              man4
        head-of-project1
head-of-project2 head-of-project3
              res1
              res2
              res3
```

where the constraints in one row have the same priority. The priority of constraints is not indicated directly in the definition of these constraints but it is used in the form of the priority of rules dedicated to the change of the assignment in order to eliminate constraint violations.

The terms 'close' (constraint `sec3`) and 'maximum access' (`man3`) are a bit 'fuzzy'. For the sake of simplicity, two rooms are considered to be close to each other if they are adjacent or they are separated only by one room. The access from a given room to a set of rooms is computed as a sum of distances from the given room to each room from this set of rooms, and the maximum access is defined as a minimum sum of those distances.

The test for the solution existence can be performed by a simple comparison of the number of members of the research team and the number of people who can be allocated in all available rooms. If the number of people is greater than the number of free places in available rooms, the solution does not exist.

```
(defrule exist-test
      (job test-exist)
      (test (< (+ (* (number-of-rooms large) 2)
                  (number-of-rooms small))
               (number-of-parameters)))
=>
      (assert (solution no))
)
```

where the `number-of-rooms` and the `number-of-parameters` are functions with obvious functionality.

Since some constraints are simple, it is possible to initialize some parameters in such a way that some constraints can be met (e.g. large room, central room, room sharing). To illustrate this possibility an explicit initialization of head-of-group (thomas) and head-of-project (joachim, hans, katharina) has been implemented. The initialization of head-of-group can serve as the example.

```
(defrule init-head-of-group
   (declare (salience 30))
      (job initialize)
      ?a <- (object  (is-a parameter)
                (param-name ?b)
                (active yes)
                (param-value nil)
                (param-type independent))
      (object (is-a component-person)
           (comp-name ?b)
           (position head-of-group))
      (object (is-a component-room)
           (usable yes)
           (size large)
           (central yes)
           (comp-name ?c))
```

```
        (not        (object (is-a parameter)
                    (param-value ?c)))
=>
        (send ?a put-param-value ?c)
)
```

where the value which meets all constraints placed on the assigned parameter is found.

Other parameters (without explicit initialization or whose explicit initialization failed) can be initialized in an implicit way. Since their domain is enumerative, one item (in this case a room) from their domain can be chosen and assigned to them.

Each constraint is associated with a set of rules to detect the presence or absence of its violation. The rule

```
(defrule cons-sec3-y
        (job evaluate)
        (object (is-a constraint)
             (cons-name sec3)
             (active yes))
        (object (is-a parameter)
             (param-name ?a&monika|ulrike)
             (param-value ?b) )
        (object (is-a parameter)
             (param-name thomas)
             (param-value ?c) )
        (test       (not        (close-to ?b ?c)))
        (not        (object (is-a constraint)
                    (cons-name sec3)
                    (viol-param $? ?a $?)))
=>
        (send [sec3] add-viol-param ?a)
)
```

can detect the constraint violation (in this particular case the violation of the constraint `sec3`) by some parameter although this violation is not registered.

The example of the rule with an inverse function is as follows:

```
(defrule cons-man3-n
        (job evaluate)
        (object (is-a constraint)
             (cons-name man3)
             (active yes))
        (object (is-a parameter)
             (param-name ?a&eva)
             (param-value ?b) )
        (object (is-a component-room)
             (comp-name ?b) )
        (object (is-a parameter)
             (param-name thomas)
             (param-value ?f) )
```

```
      (object (is-a parameter)
           (param-name monika)
           (param-value ?g) )
      (object (is-a parameter)
           (param-name ulrike)
           (param-value ?h) )
      (not ( and (object (is-a component-room)
                     (comp-name ?c&~?b&~?f&~?g&~?h)
                     (usable yes)
                     (size large)
                     (central yes))
                (test (> (distance ?b (create$ ?f ?g ?h))
                         (distance ?c  (create$ ?f ?g ?h)) ))))
      (object (is-a constraint)
           (cons-name man3)
           (viol-param $? ?a $?))
=>
      (send [man3] del-viol-param ?a)
)
```

which detects the fact that the constraint holds for some parameter although this constraint is registered to be violated by this parameter.

Each constraint is associated with rules whose aim is to modify the solution in order to fix the constraint (thus for each constraint there is a (fixable) situation in which this constraint is violated — but not every situation is fixable). Different constraints are connected with a different number of these rules. For instance three rules are attached to the constraint `res1`. The simplest case is (if it is of course possible) to move the person who violates the constraint `res1` into an empty room. A more complicated rule is:

```
(defrule change-res1-1
   (declare (salience 11))
      (job change)
      (not (result ?))
      (object (is-a constraint)
           (cons-name res1)
           (cons-violation yes)
           (viol-param ?a $? ?x $?))
      (object (is-a component-person)
           (comp-name ?a)
           (smoker ?as))
      ?c <- (object   (is-a parameter)
                (param-name ?a)
                (param-value ?b))
      (object (is-a component-person)
           (comp-name ?x)
           (smoker ~?as))
      ?d <- (object   (is-a parameter)
                (param-name ?x)
                (param-value ?y&~?b))
=>
      (send ?c put-param-value ?y)
```

```
        (send ?d put-param-value ?b)
        (assert (result success))
  )
```

Rules fixing the same constraint violation have the same priority. The priority of the rules divides the rules belonging to different constraint violations.

Not every constraint can be relaxed if it cannot be fixed (e.g. constraints `general1` and `general2` cannot be relaxed). Only the following constraints

```
head-of-group1, head-of-group2, head-of-group3, sec1, sec3,
man1, man2, head-of-project1, head-of-project2, res2, and res3.
```

can be relaxed. This relaxation can be done in several ways, for example as simple inactivation of the violated constraint:

```
(defrule relax-head-of-group1-1
      (job relaxation)
      ?a <- (constraint-to-relax head-of-group1)
      ?b <- (object   (is-a constraint)
                  (cons-name head-of-group1))
=>
      (send ?b put-active no)
      (retract ?a)
      (assert (result success))
)
```

or as a more complicated modification of the current solution and the activity of constraints related to the violated one:

```
(defrule relax-head-of-group3-1
   (declare (salience 10))
      (job relaxation)
      ?a <- (constraint-to-relax head-of-group3)
      ?b <- (object   (is-a parameter)
                  (param-name thomas)
                  (param-value ?c))
      (object (is-a component-room)
            (comp-name ?d)
            (size small)
            (usable yes)
            (central yes))
      ?e <- (object   (is-a parameter)
                  (param-value ?d))
      ?g <- (object   (is-a constraint)
                  (cons-name head-of-group2))
      ?h <- (object   (is-a constraint)
                  (cons-name head-of-group1))
=>
      (send ?b put-param-value ?d)
```

```
        (send ?e put-param-value ?c)
        (send ?g put-active no)
        (send ?h put-active yes)
        (retract ?a)
        (assert (result success))
)
```

If a constraint can be relaxed in more ways, the priority of rules represents the priority of different methods to relax this constraint.

There is no special parameter used for the purpose of the optimization. In this case to optimize the solution means to optimize the fuzzy terms 'close' and 'maximum access'. As an example can serve the rule which moves secretaries, head-of-group, and manager into new rooms if in this new position manager has better 'maximum access'.

```
(defrule opti1
        (job optimize)
        (object (is-a parameter)
             (param-name thomas)
             (param-value ?a))
        (object (is-a component-room)
             (comp-name ?a)
             (size ?as)
             (central ?ac))
        (object (is-a parameter)
             (param-name monika)
             (param-value ?b))
        (object (is-a parameter)
             (param-name ulrike)
             (param-value ?b))
        (object (is-a component-room)
             (comp-name ?b)
             (size ?bs)
             (next-to $?next))
        (object (is-a parameter)
             (param-name eva)
             (param-value ?c))
        (object (is-a component-room)
             (comp-name ?c)
             (size ?cs)
             (central ?cc))
        (test (not (and (member$ ?a ?next)
                        (member$ ?c ?next)) ))
        (object (is-a component-room)
             (comp-name ?x)
             (size ?as)
             (central ?ac)
             (usable yes))
        (object (is-a component-room)
             (comp-name ?y&~?x)
             (size ?bs)
             (usable yes))
```

```
      (object (is-a component-room)
           (comp-name ?z&~?x&~?y)
           (size ?cs)
           (central ?cc)
           (usable yes))
      (test  (close-to ?x ?y))
      (test  (<  (distance ?z (create$ ?x ?y))
                 (distance ?c (create$ ?a ?b))))
=>
      (change-rooms ?a ?x ?b ?y ?c ?z)
)
```

Rules grouped into different tasks call various functions in order to obtain information (e.g. functions `close-to`, `number-of-rooms` etc.) or to modify the solution (e.g. the function `change-rooms`). The example computes the distance between one room and a set of rooms (it is based on the computation of the shortest distance between two rooms):

```
(deffunction distance (?room ?rooms)

      (bind ?dist 0)
      (progn$ (?field ?rooms)
                  (bind ?dist (+ ?dist (path ?room ?field))))
      ?dist
)


(deffunction path (?room1 ?room2)

      (bind ?open (create$ ?room1 0))
      (bind ?close (create$ ))

      (while     (not       (member$ ?room2 ?open))
           (bind ?x (nth$ 1 ?open)) (bind ?px (nth$ 2 ?open))
           (bind ?open (delete$ ?open 1 2))
           (bind ?close (insert$ ?close 1 ?x))

           (bind ?exp
                 (send (nth$ 1 (find-instance ((?inst component-room))
                                (eq ?inst:comp-name ?x)))
                      get-next-to))

           (progn$ (?field ?exp)
                 (if  (and (not (member$ ?field ?open))
                           (not (member$ ?field ?close)))
                 then
                   (bind ?open
                      (insert$ ?open (+ (length$ ?open) 1) ?field))
                   (bind ?open
                      (insert$ ?open (+ (length$ ?open) 1) (+ ?px 1))))))

      return (nth$ (+ (member$ ?room2 ?open) 1) ?open)
)
```

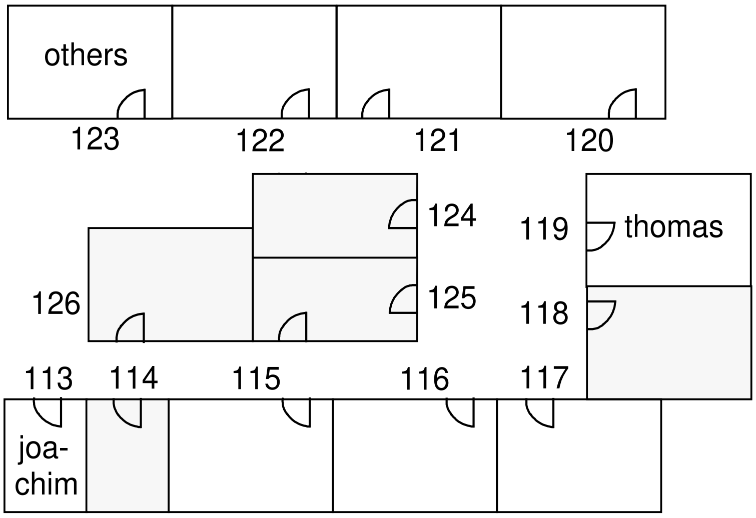

Figure 4:

6.3 Test example

Fifteen persons should be allocated to one small and eight large rooms. Altogether here are seventeen free positions in all available rooms. Thus the test for the solution existence succeeds and the search for the final solution can progress to the initialization task.

First, explicit initialization is done. Since only knowledge for the initialization of head-of-group and head-of-projects has been implemented, the explicit initialization can be done only for four parameters (representing thomas, hans, katharina, and joachim). Thomas can be assigned to a large central room. On the other hand, only one head-of-project can be initialized in such a way (e.g. hans), because only one small room is available. Parameters which stand for katharina and joachim cannot be initialized by the use of domain knowledge. These two parameters together with the others have to be initialized in an implicit way. The first values from the corresponding domain sets are assigned to these parameters. Since each person can be assigned to an arbitrary available room, the parameters have the same domain set. As a result of this, thirteen people are assigned to the same room (see Figure 4).

In this case not only the parameters concerning the required functionality of the final solution are violated but also the parameters which guarantee the solution to be physically possible (`general1` and `general2`) are violated too.

`general1`	active: yes	viol: no	()
`general2`	active: yes	viol: yes	(hans katharina werner angi harry jurgen marc uwe andy michael monika ulrike eva)

The constraint `general2` (only two people can share a large room) is violated by thirteen parameters. For this reason the system moves the people from the room 123 into empty rooms until only two people remain in this room (and the constraint `general2` holds). Now the solution has the form according to Figure 5.

Although the present state of the solution represents a real applicable solution there are still some constraints violated. The state of the constraint violations is as follows:

`head-of-group1`	active: yes	viol: no	()
`head-of-group2`	active: yes	viol: no	()
`head-of-group3`	active: yes	viol: yes	(thomas)
`sec1`	active: yes	viol: no	()
`sec2`	active: yes	viol: yes	(ulrike monika)
`sec3`	active: yes	viol: yes	(monika ulrike)
`man1`	active: yes	viol: yes	(eva)
`man2`	active: yes	viol: no	()
`man3`	active: yes	viol: yes	(eva)
`man4`	active: yes	viol: yes	(eva)
`head-of-project1`	active: yes	viol: yes	(hans katharina)
`head-of-project2`	active: yes	viol: yes	(hans katharina)
`head-of-project3`	active: no	viol: no	()
`res1`	active: yes	viol: no	()
`res2`	active: yes	viol: yes	(michael harry jurgen marc)
`res3`	active: yes	viol: yes	(harry jurgen marc)
`general1`	active: yes	viol: no	()
`general2`	active: yes	viol: no	()

The constraint with the highest priority among violated constraints is the one concerning the head-of-group (this constraint was satisfied after the initialization step but became violated as a result of removing the violation of the constraint `general2` with higher priority). For this reason marc is moved from 119 into the empty room 115 (see Figure 6).

The previous change results not only in fixing the violation of the constraint `head-of-group3` but has another positive side-effect — marc does not violate the constraints `res2` and `res3` any more.

Now the violated constraints with the highest priority are those concerning secretaries. Two moves (the allocation both secretaries into the same large room and the replacement of the room which is allocated close to the head-of-group's room by the secretaries' room) generate the solution represented in Figure 7.

At this stage of the solution all constraints concerning secretaries hold. Since as a result of relocation of andy and uwe, the large central room is empty, the constraints concerning manager are no longer in a state of violation (see Figure 8).

The present state of the constraints is as follows:

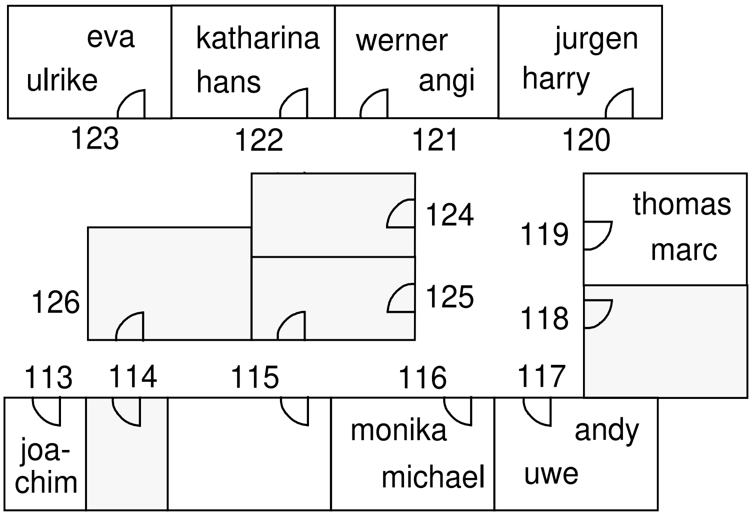

Figure 5:

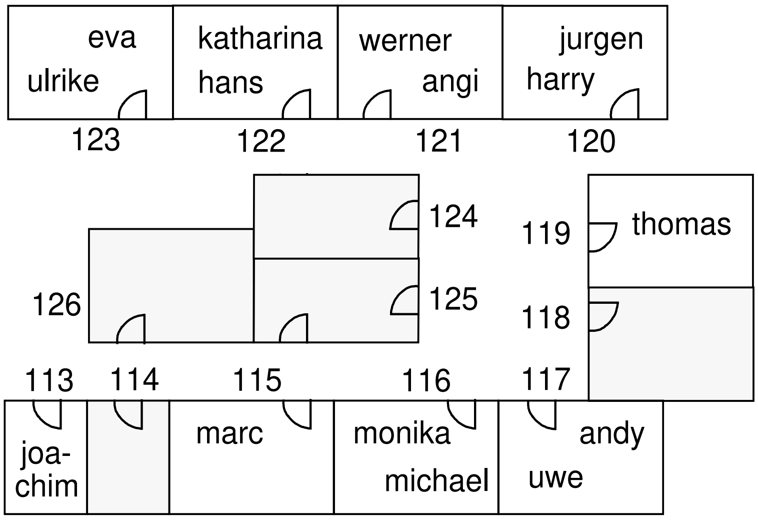

Figure 6:

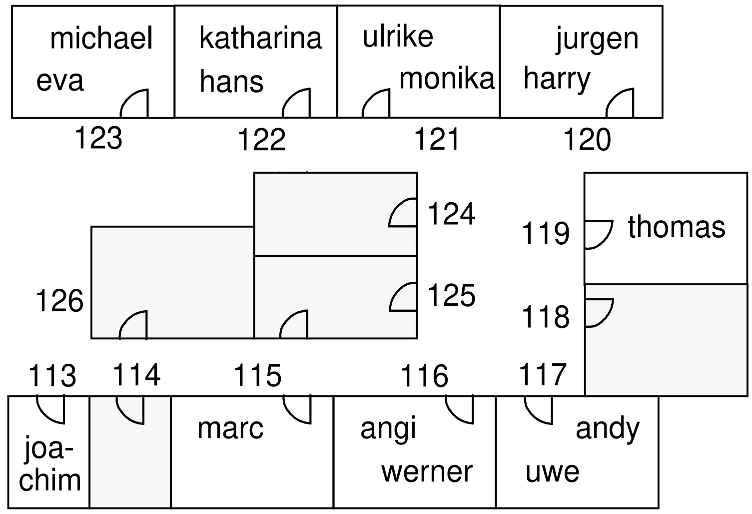

Figure 7:

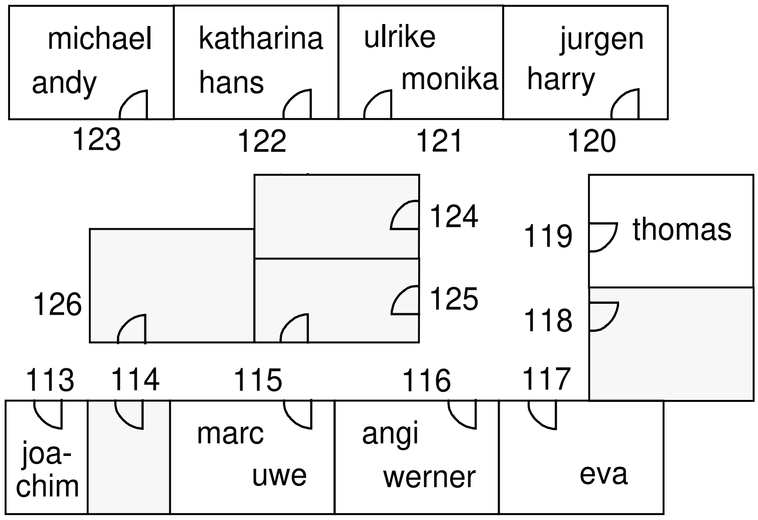

Figure 8:

`head-of-group1`	active: yes	viol: no	()
`head-of-group2`	active: yes	viol: no	()
`head-of-group3`	active: yes	viol: no	()
`sec1`	active: yes	viol: no	()
`sec2`	active: yes	viol: no	()
`sec3`	active: yes	viol: no	()
`man1`	active: yes	viol: no	()
`man2`	active: yes	viol: no	()
`man3`	active: yes	viol: no	()
`man4`	active: yes	viol: no	()
`head-of-project1`	active: yes	viol: yes	(hans katharina)
`head-of-project2`	active: yes	viol: yes	(hans katharina)
`head-of-project3`	active: no	viol: no	()
`res1`	active: yes	viol: yes	(marc uwe michael andy)
`res2`	active: yes	viol: yes	(marc uwe harry jurgen)
`res3`	active: yes	viol: yes	(harry jurgen)
`general1`	active: yes	viol: no	()
`general2`	active: yes	viol: no	()

It is not possible to ensure that each head-of-project occupies a small room (the constraint `head-of-project1`), because only one small room is available. Fortunately, the constraint `head-of-project1` can be relaxed — it can be inactivated. The consequence of this inactivation is that head-of-project can be assigned to a large room. But there is not enough rooms to ensure that head-of- project does not share his/her room with anybody else. For this reason the constraint `head-of-project2` should be inactivated too. Since now head-of-projects can share rooms, the constraint `head-of-project3` has became meaningful and should be activated. By chance this constraint holds and there is no need to modify the allocation of people. The constraints are:

`head-of-project1`	active: no	viol: no	()
`head-of-project2`	active: no	viol: no	()
`head-of-project3`	active: yes	viol: no	()

All constraints concerning researchers are violated:

`res1`	active: yes	viol: yes	(marc uwe michael andy)
`res2`	active: yes	viol: yes	(marc uwe harry jurgen)
`res3`	active: yes	viol: yes	(harry jurgen)

A few moves can produce the solution of the form depicted in Figure 9.

where the states of the constraints `res1`, `res2`, and `res3` are:

`res1`	active: yes	viol: no	()
`res2`	active: yes	viol: yes	(marc harry)
`res3`	active: yes	viol: yes	(michael jurgen)

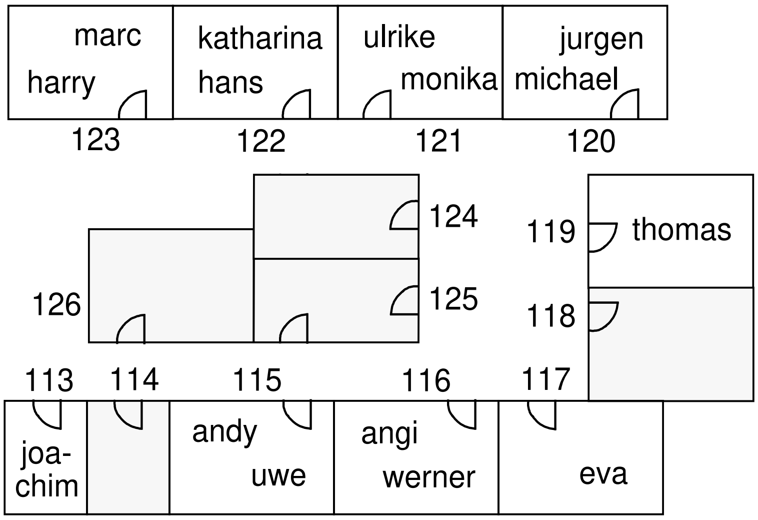

Figure 9:

Different numbers of researchers working on different projects lead to the violation of `res2` which cannot be removed. That is why this constraint should be relaxed. The same is true for the constraint `res3`.

Having done this, all active constraints hold and the search for the final solution of the room allocation problem moves from the task *evaluate* to the task *optimize*. Within this task rooms are swapped in order to optimize the terms 'close' (the constraint `sec3`) and 'maximum access' (the constraint `man3`). The final solution is depicted in Figure 10.

In order to represent the search for the solution of the test example as a trajectory in the design state space, the trajectory should start from the point (15,0) (15 active unassigned parameters and no constraint violation). Since the search was successful the trajectory ends in the origin (0,0). This trajectory can be divided into three parts related to three basic phases of the CRO strategy used to solve the configuration design problem:

1. Phase *Complete* (creating an initial complete solution)

   The trajectory points to the left. It is rising in accordance with the number of constraint violations. At the end of this phase nine constraints are violated (some of them are violated by several parameters). In all there are 45 constraint violations detected. The trajectory ends this phase at the point (0,45).

2. Phase *Revise* (looking for an acceptable solution)

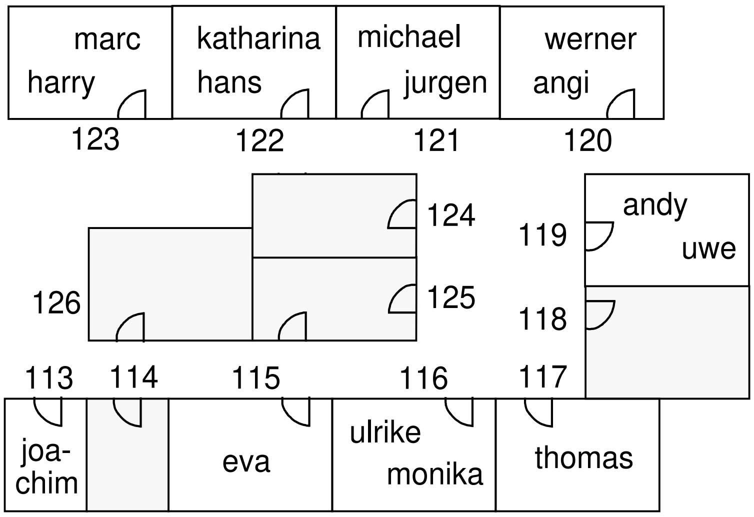

Figure 10:

The trajectory starts from the point (0,45) and ends at the point (0,0). It is parallel to the y-axis but it does not decrease in a monotone way. Its profile (the number of constraint violations after different correction steps) is depicted in Figure 11.

3. Phase *Optimize* (optimization of the acceptable solution)

   The trajectory remains in the point (0,0).

7 Discussion

The room allocation problem (in the used definition) is not a complex problem. It can be characterised by:

- Domain size of the problem is 9^{15} (15 persons have to be assigned and each of them can be placed into 9 available rooms).
- 18 constraints are and most of them can be violated by more than one parameter. Altogether 76 different constraint violations are possible (but some of them are incompatible).
- The illustrative run of the test problem solution requires 15 initialization steps within the Complete phase and 21 correction steps within the Revise phase.

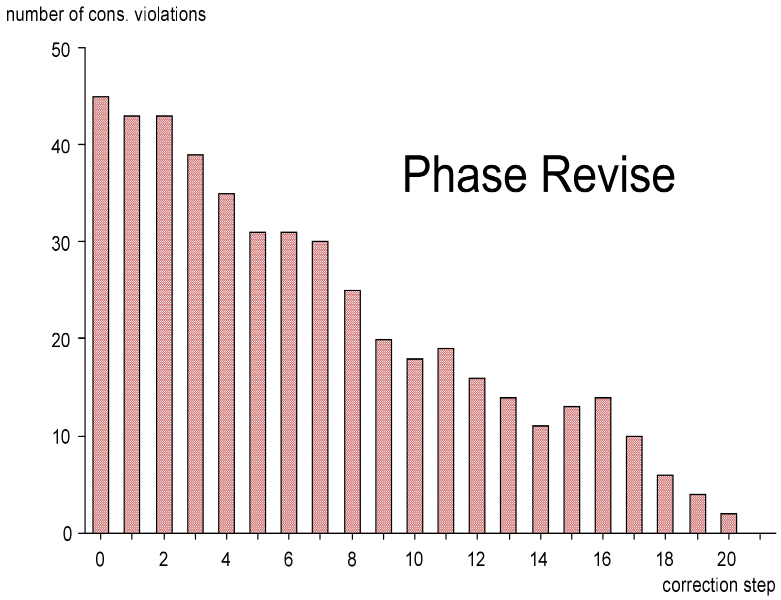

Figure 11: The second phase of the test example.

Our implementation has used all three programming paradigms offered by the CLIPS system:

- rule based programming (139 rules);
- procedural programming (11 functions);
- object oriented programming (63 objects).

The aim of the implementation was to build an experimental application for the solution of the configuration design problem in the *simplest* way in order to verify the applicability of CLIPS to solve configuration design problems. Our attempt has proven that CLIPS offers a wide range of means to implement such problems in an easy and comfortable way.

The implementation does not concern:

- the speed of the developed application

  The speed of every system based on domain knowledge heavily depends on the quality of used knowledge. Since we did not try to optimize either the quality of knowledge being used or the system performance, the execution time of the solution (PC 386 sx / 25 MHz - 243 sec and PC 586 / 75 MHz -

9 sec) has only illustrative meaning. In spite of this, it indicates that CLIPS requires more powerful computing facilities.

- economic use of system resources

  Some means of CLIPS were used in a considerably careless way trifling away CLIPS resources (e.g. excessive number of activations in agenda although only one rule was fired and the other activations were removed — for instance when the activity was transferred to the task *evaluate* for the first time there were 1490 activations on the agenda).

- consistent use of CLIPS means

  A lot of CLIPS means able to make the final implementation more effective were not used in our test implementation, especially those dedicated to handle complex programs (e.g. the possibility of partitioning the program into several independent modules with the focus only on one module at the same time).

The program was written in an incremental way. First, the CLIPS system version 6.03 in the form of 16 bit application running under WfW 3.11 was used. At a certain stage of the program development the used CLIPS version kept crashing (with the obligatory general fault protection message) and was not able to find a solution. In the next stage the CLIPS system version 6.03 in the form of 32 bit application running under DOS 5.0 has been used.

This experience can indicate that the CLIPS resources are not unlimited and that the consistent use of all means offered by CLIPS to decrease the resource consumption is necessary, especially for the implementation of complex problems.

References

[1] Z. ZDRAHAL, E. MOTTA, and A. STUTT. Optimising Revision Procedures for Configuration Design. Technical Report TR-112, Open University.

[2] CLIPS Reference Manual. Volume I and II. 1993, Software Technology Branch, Lyndon B. Johnson Space Center.

[3] E. MOTTA, K. O'HARA, and N. SHADBOLT. Grounding GDMs: A Structured Case Study. Technical Report TR-91, Open University.